

6 Appendices

A Prompt Templates

The complete system prompts used for each of the three prompting strategies are provided below for reproducibility. The {function} placeholder was dynamically replaced with the specific function name for each programming problem.

A.1 IO System Prompt

You are generating simulated student Java code submissions for a programming problem.

Goal

- Produce compilable Java code submissions, each featuring exactly one non-trivial logical error (no syntax errors or runtime crashes).
- Each submission should reflect a genuine attempt by a novice programmer.

Constraints

- Implement the required Java method using the exact name provided, represented as {function}.
- Include only one logical error per submission; also, the code must compile and run without crashing.
- Adhere to all I/O and method signature requirements as outlined in the problem description.
- Code blocks must contain only Java code exclude comments, hints, debug output, or external explanations.

Variety

- Each submission should contain a distinct, plausible logical error pertinent to the specific problem.
- Do not repeat the same error type across different submissions; ensure diversity until reasonable variation is exhausted.

Output Format

For each submission:

Submission i:

```
```java
```

```
// Substitute this block with Java code that implements the {function} method, using
the parameters and return type specified in the problem statement. Introduce
exactly one plausible, non-trivial logical error relevant to the problem.
```

```
```
```

A.2 CoT System Prompt

```

You are generating simulated student Java code submissions for a programming problem.

### Goal
- Produce compilable Java code submissions, each featuring exactly one non-trivial
  logical error (no syntax errors or runtime crashes).
- Each submission should reflect a genuine attempt by a novice programmer.

### Reasoning policy
- For EACH submission, FIRST provide a SHORT reasoning (1 3 sentences) describing
  the intended approach and the kind of subtle logical slip it might contain (no
  test cases, no step-by-step).
- Then immediately provide the code block.
- Keep the reasoning concise and high-level.

### Constraints
- Implement the required Java method using the exact name provided, represented as
  {function}.
- Include only one logical error per submission; also, the code must compile and run
  without crashing.
- Adhere to all I/O and method signature requirements as outlined in the problem
  description.
- Code blocks must contain only Java c o d e exclude comments, hints, debug output, or
  external explanations.

### Variety
- Each submission should contain a distinct, plausible logical error pertinent to the
  specific problem.
- Do not repeat the same error type across different submissions; ensure diversity
  until reasonable variation is exhausted.

### Output Format
For each submission:
Submission i:
Reasoning: <1 3 sentences, concise, high-level>
```java
// Substitute this block with Java code that implements the {function} method, using
 the parameters and return type specified in the problem statement. Introduce
 exactly one plausible, non-trivial logical error relevant to the problem.

```

### A.3 Self-Refine System Prompts

#### A.3.1 Initial Generation Prompt.

```

625 You are generating simulated student Java code submissions for a programming problem.
626
627 ### Goal
628
629 - Produce compilable Java code submissions, each featuring exactly one non-trivial
630 logical error (no syntax errors or runtime crashes).
631 - Each submission should reflect a genuine attempt by a novice programmer.
632
633
634 ### Reasoning policy
635
636 - For each submission, first output a single short line starting with
637 Reasoning : (1 3 sentences, high-level; no test cases; no
638 step-by-step).
639 - Immediately after the Reasoning line, output the code block.
640
641
642 ### Constraints
643
644 - Implement the required Java method using the exact name {function}.
645 - Exactly one logical error per submission; code must compile and not crash.
646 - Respect the problems I/O and method signature.
647 - No trivial sabotage (no hardcoded answers, no skipping essential logic, no
648 deliberate exceptions).
649 - Avoid `main`, I/O, randomness, external resources, and imports unless explicitly
650 required by the problem.
651 - Inside code blocks: Java code only (no comments, hints, or debug output).
652
653
654 ### Variety
655
656 - Across submissions, each logical error must be distinct and plausible for
657 the problem.
658 - Do not repeat the same error type unless reasonable variety is exhausted.
659
660
661 ### Output format (strict)
662
663 For each submission:
664
665 Reasoning: <1 3 sentences, concise, high-level>
666 code
667 Submission i:
668 Reasoning: <1 3 sentences, concise, high-level>
669 ```java
670 // code only; one fenced block per submission
671 ```
672
673
674
675
676

```

Do not include any extra text before the first submission or after the last submission.

### A.3.2 Feedback Prompt (Critic).

You are reviewing a **set** of simulated student Java submissions with the following objective:

- Each submission compiles and contains **exactly one** non-trivial logical error.
- Submissions look like genuine novice attempts (no trivial sabotage).
- The required method name is exactly **{function}**.
- Output from the generator includes a Reasoning : line (for CoT setups) followed by a single fenced Java code block containing **only** Java code, and it respects the problems I/O/signature.

**### Your task**

- Evaluate the **entire set** and judge each submission against the objective.
- For any submission that **fails** a requirement, provide concise guidance on what to change so that the **next** revision still contains exactly one non-trivial logical error but becomes compliant (e.g., fix method name/signature, remove comments/debug prints, avoid multiple bugs, ensure plausibility, preserve formatting, maintain distinctness across the set).
- If there are cross-submission conflicts (e.g., duplicate error types), identify them and specify which submission(s) should be revised and in what direction (no code).

**### Style**

- Be concise and concrete (aim for 2-6 sentences per submission verdict).
- Do not reveal hidden tests or provide step-by-step chain-of-thought.
- **Do not provide code** or code fragments.

**### Required output structure (strict)**

- A numbered list covering **all** submissions in the form:

```

...
Submission i: Compliant <brief justification>
...
or
...
Submission i: Revise <specific issues and how to adjust while keeping exactly one
 non-trivial logical error>

```

```
...
```

- Optionally, an **Overall:** paragraph for cross-submission issues (e.g., duplicate error types, inconsistent formatting).

### A.3.3 Refinement Prompt.

You are revising a **set** of simulated student Java submissions based on batch feedback.

#### ### Goal

- Produce a **revised set** that preserves the original **count** of submissions.
- For items marked **Compliant** in feedback: **return the exact original code unchanged**.
- For items marked **Revise**: return a revised version that **compiles** and **intentionally contains exactly one non-trivial logical error**, while addressing the feedback.

#### ### Constraints

- Keep the method name **exactly {function}** and preserve the required I/O and signature from the problem.
- Ensure the logical error is plausible (no hardcoding answers, no skipping logic, no deliberate exceptions).
- Maintain **distinct** error types across the set where required by feedback.
- Avoid `main`, I/O, randomness, external resources, and imports unless explicitly required.
- Inside code blocks: **Java code only** (no comments, hints, or debug prints).

#### ### Output policy

- Output **the same number** of submissions as the input set.
- For each submission, output **one** fenced Java code block (no extra prose).
- No text before the first submission or after the last submission.

#### ### Output format (strict)

For each submission **i** in order:

```
...
```

Submission i:

```
```java
```

```
// one fenced Java block; if Compliant, reproduce original unchanged; if Revise,
    provide the revised code with exactly one non-trivial logical error
```

B Sample Inputs and Outputs

To illustrate the output of each prompting strategy, this section provides a concrete example based on the ‘[plusOut](#)’ problem from the CodeWorkout website. The outputs shown were generated by the *anthropic/claude-sonnet-4* model and are representative of the results from our experiments.

B.1 Problem Description (Input)

The following problem description was provided to the model as input for all strategies. The task is to implement a Java method named ‘plusOut’.

Write a function in Java that implements the following logic: Given a string `str` and a non-empty word, return a version of the original string where all chars have been replaced by pluses (+), except for appearances of the word which are preserved unchanged.

Examples:

`plusOut("12xy34", "xy") -> "++xy++"`

`plusOut("12xy34", "1") -> "1++++"`

B.2 Input-Output (IO) Prompting Output

Given the prompt in Appendix A.1, the model produced the following submissions. Note the absence of explicit reasoning for the introduced errors.

Submission 1:

```
```java
public String plusOut(String str, String word) {
 String result = "";
 for (int i = 0; i < str.length(); i++) {
 if (i <= str.length() - word.length() && str.substring(i, i +
word.length()).equals(word)) {
 result += word;
 i += word.length() - 1;
 } else {
 result += "+";
 }
 }
 return result;
}
```
```

Submission 2:

```
```java
public String plusOut(String str, String word) {
 String result = str;
 for (int i = 0; i < str.length(); i++) {
```

```

833 if (str.substring(i, Math.min(i + word.length(), str.length())).equals(word))
834 {
835 i += word.length() - 1;
836 } else {
837 result = result.substring(0, i) + "+" + result.substring(i + 1);
838 }
839 }
840 return result;
841 }
842 ...

```

Submission 3:

```

846 ```java
847 public String plusOut(String str, String word) {
848 String result = "";
849 int i = 0;
850 while (i < str.length()) {
851 if (str.startsWith(word, i)) {
852 result += word;
853 i += word.length();
854 } else {
855 result += "+";
856 i++;
857 }
858 }
859 return result;
860 }
861 ...

```

Submission 4:

```

867 ```java
868 public String plusOut(String str, String word) {
869 boolean[] preserve = new boolean[str.length()];
870
871 for (int i = 0; i < str.length() - word.length(); i++) {
872 if (str.substring(i, i + word.length()).equals(word)) {
873 for (int j = i; j < i + word.length(); j++) {
874 preserve[j] = true;
875 }
876 }
877 }
878
879 String result = "";
880 for (int i = 0; i < str.length(); i++) {
881 if (preserve[i]) {

```

```

885 result += str.charAt(i);
886 } else {
887 result += "+";
888 }
889 }
890 }
891 return result;
892 }
893 ---
894
895 Submission 5:
896 ```java
897 public String plusOut(String str, String word) {
898 String result = "";
899 for (int i = 0; i < str.length(); i++) {
900 boolean found = false;
901 if (i + word.length() <= str.length()) {
902 String sub = str.substring(i, i + word.length());
903 if (sub.equals(word)) {
904 result += word;
905 i += word.length() - 1;
906 found = true;
907 }
908 }
909 }
910 if (!found) {
911 result += "+";
912 }
913 }
914 }
915 return result;
916 }
917 ---
918
919 Submission 6:
920 ```java
921 public String plusOut(String str, String word) {
922 String temp = str.replace(word, "PLACEHOLDER");
923 String result = "";
924 for (int i = 0; i < temp.length(); i++) {
925 if (temp.charAt(i) == 'P' && temp.substring(i).startsWith("PLACEHOLDER")) {
926 result += word;
927 i += "PLACEHOLDER".length() - 1;
928 } else {
929 result += "+";
930 }
931 }
932 }
933 }
934 return result;
935 }
936

```



```
937 }
938 ---
939
940 Submission 7:
941 ```java
942 public String plusOut(String str, String word) {
943 String result = "";
944 int wordIndex = 0;
945
946 for (int i = 0; i < str.length(); i++) {
947 if (str.charAt(i) == word.charAt(wordIndex)) {
948 wordIndex++;
949 if (wordIndex == word.length()) {
950 result = result.substring(0, result.length() - word.length() + 1) +
951 word;
952 wordIndex = 0;
953 } else {
954 result += "+";
955 }
956 } else {
957 wordIndex = 0;
958 result += "+";
959 }
960 }
961 return result;
962 }
963 ---
964
965 Submission 8:
966 ```java
967 public String plusOut(String str, String word) {
968 String result = "";
969 for (int i = 0; i < str.length(); i++) {
970 if (i + word.length() < str.length() && str.substring(i, i +
971 word.length()).equals(word)) {
972 result += word;
973 i += word.length() - 1;
974 } else {
975 result += "+";
976 }
977 }
978 return result;
979 }
980 ---
981
982
983
984
985
986
987
988
```

### B.3 Chain-of-Thought (CoT) Prompting Output

Given the prompt in Appendix A.2, the model first provided reasoning for each intended error before generating the code.

```

Submission 1:
/*Reasoning: This approach tries to iterate through the string and check if the
current position starts with the target word. However, it fails to properly
handle overlapping matches and doesn't correctly preserve the word when found. */
```java
public String plusOut(String str, String word) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        if (i <= str.length() - word.length() && str.substring(i, i +
word.length()).equals(word)) {
            result.append(word);
        } else {
            result.append('+');
        }
    }
    return result.toString();
}
```

Submission 2:
/*Reasoning: This solution attempts to replace all characters with pluses first, then
restore the word occurrences. The logical error is in the replacement logic where
it doesn't properly account for the length difference after replacements.*/
```java
public String plusOut(String str, String word) {
    String result = str.replaceAll(".", "+");
    int index = 0;
    while ((index = str.indexOf(word, index)) != -1) {
        result = result.substring(0, index) + word + result.substring(index +
word.length());
        index++;
    }
    return result;
}
```

Submission 3:
/*Reasoning: This approach uses a character array and tries to mark positions that
should remain unchanged. The error lies in the increment logic where it advances
by only 1 position even after finding a complete word match.*/
```java
public String plusOut(String str, String word) {
    char[] chars = str.toCharArray();

```

```

1041     for (int i = 0; i < chars.length; i++) {
1042         if (i <= str.length() - word.length() && str.substring(i, i +
1043 word.length()).equals(word)) {
1044             i += word.length() - 1;
1045         } else {
1046             chars[i] = '+';
1047         }
1048     }
1049     return new String(chars);
1050 }
1051 ...
1052
1053 Submission 4:
1054 /*Reasoning: This solution tries to build the result by checking each position for
1055 word matches. The logical flaw is that it doesn't skip the remaining characters
1056 of a matched word, causing partial overwriting.*/
1057
1058 ```java
1059 public String plusOut(String str, String word) {
1060     StringBuilder result = new StringBuilder();
1061     int i = 0;
1062     while (i < str.length()) {
1063         if (str.startsWith(word, i)) {
1064             result.append(word);
1065             i += word.length();
1066         } else {
1067             result.append('+');
1068             i++;
1069         }
1070     }
1071     return result.toString();
1072 }
1073 ...
1074
1075 Submission 5:
1076 /*Reasoning: This approach attempts to use string replacement by first marking word
1077 positions with a placeholder, then replacing everything else. The error is in
1078 assuming the placeholder won't interfere with subsequent operations.*/
1079
1080 ```java
1081 public String plusOut(String str, String word) {
1082     String temp = str.replace(word, "PLACEHOLDER");
1083     temp = temp.replaceAll(".", "+");
1084     return temp.replace("PLACEHOLDER", word);
1085 }
1086 ...
1087
1088 Submission 6:

```

```

1093  /*Reasoning: This solution tries to iterate and build the result character by
1094  character while checking for word matches. The logical error is in the boundary
1095  checking condition which uses incorrect comparison operators.*/
1096  ```java
1097  public String plusOut(String str, String word) {
1098      StringBuilder result = new StringBuilder();
1099      for (int i = 0; i < str.length(); i++) {
1100          if (i < str.length() - word.length() && str.substring(i, i +
1101 word.length()).equals(word)) {
1102              result.append(word);
1103              i += word.length() - 1;
1104          } else {
1105              result.append('+');
1106          }
1107      }
1108      return result.toString();
1109  }
1110  ```
1111
1112
1113

```

B.4 Self-Refine Prompting Interaction

The following demonstrates a two-iteration example of the Self-Refine process, as detailed in Appendix A.3. The model begins with an initial set of generations, receives feedback from a critic agent, refines its code, and then repeats this cycle until the submission is compliant.

Initial Generation.

```

1125  Submission 1:
1126  /*Reasoning: Student attempts to replace characters one by one but fails to properly
1127  skip over the word when found, leading to partial replacements within the word
1128  itself.*/
1129  ```java
1130  public String plusOut(String str, String word) {
1131      String result = "";
1132      for (int i = 0; i < str.length(); i++) {
1133          if (i <= str.length() - word.length() && str.substring(i, i +
1134 word.length()).equals(word)) {
1135              result += word;
1136              i++;
1137          } else {
1138              result += "+";
1139          }
1140      }
1141      return result;
1142  }
1143  ```
1144

```

Manuscript submitted to ACM

```

1145 }
1146 ---
1147
1148 Submission 2:
1149 /*Reasoning: Student correctly identifies word occurrences but uses the wrong
1150    increment, advancing by only 1 instead of the full word length.*/
1151 ---java
1152 public String plusOut(String str, String word) {
1153     StringBuilder result = new StringBuilder();
1154     int i = 0;
1155     while (i < str.length()) {
1156         if (i <= str.length() - word.length() && str.substring(i, i +
1157 word.length()).equals(word)) {
1158             result.append(word);
1159             i += word.length() - 1;
1160         } else {
1161             result.append("+");
1162         }
1163         i++;
1164     }
1165     return result.toString();
1166 }
1167 ---
1168
1169 Submission 3:
1170 /*Reasoning: Student uses replace method but incorrectly replaces the word with a
1171    placeholder first, then tries to restore it, causing issues with overlapping
1172    replacements.*/
1173 ---java
1174 public String plusOut(String str, String word) {
1175     String temp = str.replace(word, "TEMP");
1176     String result = "";
1177     for (int i = 0; i < temp.length(); i++) {
1178         if (temp.charAt(i) == 'T' && i + 3 < temp.length() && temp.substring(i, i +
1179 4).equals("TEMP")) {
1180             result += word;
1181             i += 3;
1182         } else {
1183             result += "+";
1184         }
1185     }
1186     return result;
1187 }
1188 ---
1189

```

```

1197 Submission 4:
1198 /*Reasoning: Student attempts to build the result by checking each position but has
1199    an off-by-one error in the boundary condition for substring extraction.*/
1200 ```java
1201 public String plusOut(String str, String word) {
1202     String result = "";
1203     for (int i = 0; i < str.length(); i++) {
1204         if (i < str.length() - word.length() && str.substring(i, i +
1205 word.length()).equals(word)) {
1206             result += word;
1207             i += word.length() - 1;
1208         } else {
1209             result += "+";
1210         }
1211     }
1212     return result;
1213 }
1214 }
1215 ```
1216
1217
1218 Submission 5:
1219 /*Reasoning: Student correctly finds word matches but forgets to handle the case
1220    where the word appears at the very end of the string due to incorrect loop
1221    bounds.*/
1222 ```java
1223 public String plusOut(String str, String word) {
1224     StringBuilder sb = new StringBuilder();
1225     int i = 0;
1226     while (i < str.length()) {
1227         if (i + word.length() < str.length() && str.substring(i, i +
1228 word.length()).equals(word)) {
1229             sb.append(word);
1230             i += word.length();
1231         } else {
1232             sb.append("+");
1233             i++;
1234         }
1235     }
1236     return sb.toString();
1237 }
1238 }
1239 ```
1240
1241
1242
1243 Submission 6:
1244 /*Reasoning: Student uses a character-by-character approach but fails to properly
1245    handle multi-character words, only checking the first character of the word.*/
1246 ```java
1247
1248

```

```

1249 public String plusOut(String str, String word) {
1250     String result = "";
1251     for (int i = 0; i < str.length(); i++) {
1252         boolean found = false;
1253         if (str.charAt(i) == word.charAt(0)) {
1254             if (i + word.length() <= str.length()) {
1255                 String sub = str.substring(i, i + word.length());
1256                 if (sub.equals(word)) {
1257                     result += word;
1258                     i += word.length() - 1;
1259                     found = true;
1260                 }
1261             }
1262         }
1263     }
1264     if (!found) {
1265         result += "+";
1266     }
1267 }
1268
1269 return result;
1270
1271 }
1272 ...

```

Iteration 1: Feedback. The critic agent receives the initial code and provides the following feedback:

Submission 1: Revise The logical error (incrementing `i` by only 1 instead of `word.length()` when a match is found) is non-trivial and plausible, but the increment should be `i += word.length() - 1` to maintain the loop's natural increment, making the error more subtle and realistic for a novice.

Submission 2: Compliant Contains exactly one non-trivial logical error where the increment is `word.length() - 1` instead of `word.length()`, causing overlapping matches to be missed. The error is subtle and represents a common off-by-one mistake.

Submission 3: Revise The approach using string replacement with "TEMP" is overly complex for a novice submission and contains multiple issues: hardcoded "TEMP" length check and potential boundary problems. Simplify to a more straightforward approach with a single, clear logical error.

Submission 4: Revise The boundary condition uses `<` instead of `<=`, which will miss matches at the end of the string. While this is a logical error, it's too similar to other boundary condition errors in the set. Change to a different type of logical error to maintain distinctness.

1301 Submission 5: Compliant Contains exactly one non-trivial logical error in the
 1302 boundary condition (`i + word.length() < str.length()` instead of `<=`, which
 1303 prevents detection of matches at the string's end. This represents a common
 1304 boundary condition mistake.
 1305
 1306 Submission 6: Compliant Contains exactly one non-trivial logical error where it
 1307 only checks if the first character matches before attempting substring
 1308 comparison, but the logic is otherwise sound. This represents a reasonable
 1309 optimization attempt that introduces a subtle bug.
 1310
 1311

1312
 1313 *Iteration 1: Refinement.* Based on the feedback, the model revises the codes:
 1314

1315 Submission 1:
 1316 ```java
 1317 public String plusOut(String str, String word) {
 1318 String result = "";
 1319 for (int i = 0; i < str.length(); i++) {
 1320 if (i <= str.length() - word.length() && str.substring(i, i +
 1321 word.length()).equals(word)) {
 1322 result += word;
 1323 i += word.length() - 1;
 1324 } else {
 1325 result += "+";
 1326 }
 1327 }
 1328 return result;
 1329 }
 1330 }
 1331 ```
 1332
 1333 Submission 2:
 1334 ```java
 1335 public String plusOut(String str, String word) {
 1336 StringBuilder result = new StringBuilder();
 1337 int i = 0;
 1338 while (i < str.length()) {
 1339 if (i <= str.length() - word.length() && str.substring(i, i +
 1340 word.length()).equals(word)) {
 1341 result.append(word);
 1342 i += word.length() - 1;
 1343 } else {
 1344 result.append("+");
 1345 }
 1346 i++;
 1347 }
 1348 return result.toString();
 1349 }
 1350 }
 1351 ```
 1352

Manuscript submitted to ACM


```

1353 }
1354 ---
1355
1356 Submission 3:
1357 ```java
1358 public String plusOut(String str, String word) {
1359     String result = "";
1360     for (int i = 0; i < str.length(); i++) {
1361         if (i <= str.length() - word.length() && str.substring(i, i +
1362 word.length()).equals(word)) {
1363             result += word;
1364             i += word.length();
1365         } else {
1366             result += "+";
1367         }
1368     }
1369     return result;
1370 }
1371 ---
1372
1373 Submission 4:
1374 ```java
1375 public String plusOut(String str, String word) {
1376     String result = "";
1377     for (int i = 0; i < str.length(); i++) {
1378         if (i <= str.length() - word.length() && str.substring(i, i +
1379 word.length()).equalsIgnoreCase(word)) {
1380             result += word;
1381             i += word.length() - 1;
1382         } else {
1383             result += "+";
1384         }
1385     }
1386     return result;
1387 }
1388 ---
1389
1390 Submission 5:
1391 ```java
1392 public String plusOut(String str, String word) {
1393     StringBuilder sb = new StringBuilder();
1394     int i = 0;
1395     while (i < str.length()) {
1396         if (i + word.length() < str.length() && str.substring(i, i +
1397 word.length()).equals(word)) {

```

```

1405         sb.append(word);
1406         i += word.length();
1407     } else {
1408         sb.append("+");
1409         i++;
1410     }
1411 }
1412 }
1413 return sb.toString();
1414 }
1415 ...
1416
1417 Submission 6:
1418 ```java
1419 public String plusOut(String str, String word) {
1420     String result = "";
1421     for (int i = 0; i < str.length(); i++) {
1422         boolean found = false;
1423         if (str.charAt(i) == word.charAt(0)) {
1424             if (i + word.length() <= str.length()) {
1425                 String sub = str.substring(i, i + word.length());
1426                 if (sub.equals(word)) {
1427                     result += word;
1428                     i += word.length() - 1;
1429                     found = true;
1430                 }
1431             }
1432         }
1433     }
1434     if (!found) {
1435         result += "+";
1436     }
1437 }
1438 return result;
1439 }
1440 }
1441 ...
1442

```

Iteration 2: Feedback. The critic agent reviews the refined codes and now deems it compliant with all constraints.

```

1447 Submission 1: Compliant      Contains exactly one non-trivial logical error where the
1448     loop increment `i += word.length() - 1` combined with the for loop's `i++` causes
1449     the index to advance by only `word.length()` instead of the full word length,
1450     leading to potential overlapping matches.
1451
1452 Submission 2: Compliant      Contains exactly one non-trivial logical error identical
1453     to Submission 1, where the increment logic `i += word.length() - 1` followed by
1454     `i++` in the while loop causes incorrect index advancement.
1455

```

Manuscript submitted to ACM

Submission 3: Compliant Contains exactly one non-trivial logical error where ``i += word.length()`` without accounting for the for loop's automatic ``i++`` causes the index to skip one character after each word match.

Submission 4: Compliant Contains exactly one non-trivial logical error using ``equalsIgnoreCase`` instead of ``equals``, making the comparison case-insensitive when it should be case-sensitive, plus the same increment issue as Submission 1.

Submission 5: Compliant Contains exactly one non-trivial logical error in the boundary condition ``i + word.length() < str.length()`` which uses strict inequality instead of ``<=``, causing it to miss word matches at the end of the string.

Submission 6: Compliant Contains exactly one non-trivial logical error in the optimization attempt that only checks matches when the first character matches, but this approach can miss valid word occurrences and has the same increment issue as other submissions.

C Final Problem Set and Struggling-Level Classification

The final set of 37 Java programming problems used in our experiments is listed below. Problems were classified into Low, Medium, and High struggling levels based on the total number of human student submissions in the original dataset, serving as a proxy for student struggle. The problems were sorted by submission count and then evenly divided into three tiers.

Table 3. List of the 37 Finalized Problems and the Students' Struggling Levels

Problem Name	Struggling Level
countevens	Low
sortasum	Low
luckysum	Low
squirrelplay	Low
cigarparty	Low
centeredaverage	Low
fizzbuzz	Low
noteensum	Low
datefashion	Low
repeatend	Low
tenrun	Low
answercell	Low
love6	Low
zeromax	Medium
has77	Medium
endother	Medium
sum67	Medium
withoutten	Medium
in1to10	Medium
countcode	Medium
xybalance	Medium
mixstring	Medium
roundsum	Medium
countclumps	Medium
fix45	Medium
bobthere	Medium
alarmclock	High
evenlyspaced	High
greenticket	High
zipzap	High
getsandwich	High
xyzthere	High
caughtspeeding	High
plusout	High
without2	High
makechocolate	High
blackjack	High

D Statistical Analyses for RQ1

This appendix provides the detailed results of the independent-samples t -tests conducted to evaluate the statistical significance of differences in error diversity, both between prompting strategies for each model (Table 4) and between the models themselves (Table 5). In both cases, we report Welch's t -statistic, associated degrees of freedom, p -values, and effect sizes using Hedges' g .

Prompting Comparisons within Models

Table 4 compares the three prompting strategies (IO, CoT, Self-Refine) within each model. Several key findings emerge:

- For **GPT-4o**, CoT prompts produced significantly more diverse erroneous code than either IO or Self-Refine (both $p < .001$, medium effect sizes). However, IO and Self-Refine did not differ significantly, suggesting CoT was the primary driver of diversity increases.
- For **GPT-5**, results were more mixed. CoT produced **significantly** higher diversity than IO ($p < .001$, small effect), but did not differ significantly from Self-Refine. **IO and Self-Refine showed a significant difference, with IO producing greater diversity than Self-Refine** ($p < .01$).
- For **Claude Sonnet 4**, baseline and CoT produced significantly higher diversity compared to **Self-Refine** (both $p < .001$, medium effect sizes). Interestingly, **baseline and CoT did not differ significantly** ($p = 0.421$).
- For **Gemini 2.5 Pro**, **two of the three** pairwise comparisons were statistically significant, though the effect sizes were generally small to medium. This indicates the model's diversity was sensitive to prompt framing, with CoT and Self-Refine producing more variation than IO. **However, the difference between Self-Refine and IO was not statistically significant** ($p = 0.185$).
- For **Grok Code Fast 1**, CoT again yielded significantly higher diversity than both IO and Self-Refine, but IO and Self-Refine did not differ.

Taken together, these results highlight that the **influence of prompting strategy is model-dependent**: CoT consistently boosts error diversity for smaller models like GPT-4o and Grok Code Fast 1, while iterative refinement plays a larger role in Claude Sonnet 4. This suggests that the mechanisms by which LLMs generate variation are not uniform and may be linked to model architecture or training data.

D.1 Model Comparisons

Table 5 reports pairwise comparisons of mean edit distances across models, pooling over prompting strategies. The following trends emerge:

- **GPT-4o** consistently produced significantly less diverse erroneous code compared to all other models. Effect sizes were generally large negative (Hedges' g between -0.62 and -0.83) but **medium** against Grok Code Fast 1 ($g = -0.45$). This confirms the descriptive finding that GPT-4o tends to generate conservative or repetitive error patterns.
- **Gemini 2.5 Pro** and **Claude Sonnet 4** exhibited the highest diversity overall. Their comparison with GPT-4o showed large and highly significant differences. **However, Claude Sonnet 4 was significantly less diverse than Gemini 2.5 Pro** ($p < .001$, $g = -0.11$), **indicating Gemini 2.5 Pro had the highest diversity**.
- **GPT-5** occupied an intermediate position, significantly more diverse than GPT-4o but **significantly less diverse** than Gemini 2.5 Pro ($p < .001$, $g = -0.15$). Interestingly, GPT-5 **did not differ significantly** from Claude Sonnet

4 ($p = 0.242$), but was significantly more diverse than Grok Code Fast 1 ($p < .001$), indicating that it produced broader error distributions than Grok.

- **Grok Code Fast 1** tended to fall below Gemini and Claude in diversity, but above GPT-4o. Comparisons with Claude Sonnet 4 ($p < .001$, $g = 0.26$) and Gemini 2.5 Pro ($p < .001$, $g = 0.40$) indicate that Grok's diversity was significantly lower with small to moderate effect sizes (Group A is more diverse than Group B).

Overall, these statistical tests corroborate the descriptive results: **Gemini 2.5 Pro is the most diverse model, GPT-4o is the least diverse, and GPT-5, Claude Sonnet 4, and Grok Code Fast 1 fall in between.** Importantly, the effect sizes suggest these differences are not just statistically significant but also practically meaningful, especially in comparisons involving GPT-4o.

Table 4. Independent t-test results for prompt strategies by model.

Model	Group A	Group B	<i>t</i> -statistic	<i>df</i>	<i>p</i> -value	Hedges' <i>g</i>
GPT 4o	CoT	baseline	6.44	735.44	$2.17e - 10$	0.43
GPT 4o	CoT	selfrefine	5.90	838.25	$5.38e - 09$	0.37
GPT 4o	baseline	selfrefine	-0.40	1282.14	0.688	-0.02
GPT 5	CoT	baseline	3.70	2057.80	0.00023	0.16
GPT 5	baseline	selfrefine	-2.96	1119.27	0.0031	-0.15
GPT 5	CoT	selfrefine	0.12	1242.51	0.906	0.01
Claude Sonnet 4	baseline	selfrefine	6.43	1204.27	$1.79e - 10$	0.33
Claude Sonnet 4	CoT	selfrefine	5.18	1176.87	$2.61e - 07$	0.30
Claude Sonnet 4	CoT	baseline	-0.80	1359.72	0.421	-0.04
Gemini 2.5 Pro	CoT	selfrefine	4.16	750.48	$3.53e - 05$	0.28
Gemini 2.5 Pro	CoT	baseline	2.81	810.28	0.0051	0.19
Gemini 2.5 Pro	selfrefine	baseline	1.33	1011.29	0.185	0.08
Grok Code Fast 1	CoT	baseline	4.08	575.84	$5.22e - 05$	0.28
Grok Code Fast 1	CoT	selfrefine	3.23	650.64	0.0013	0.25
Grok Code Fast 1	baseline	selfrefine	-0.39	642.42	0.700	-0.03

Table 5. Independent t-test results for model comparisons.

Group A	Group B	<i>t</i> -statistic	<i>df</i>	<i>p</i> -value	Hedges' <i>g</i>
GPT 4o	Gemini 2.5 Pro	-22.97	2680.87	$1.01e - 106$	-0.83
GPT 4o	GPT 5	-21.78	4396.18	$5.83e - 100$	-0.62
GPT 4o	Claude Sonnet 4	-21.60	3930.99	$6.63e - 98$	-0.67
GPT 4o	Grok Code Fast 1	-12.27	2730.24	$9.52e - 34$	-0.45
Gemini 2.5 Pro	Grok Code Fast 1	10.62	2814.20	$7.27e - 26$	0.40
Claude Sonnet 4	Grok Code Fast 1	8.00	3302.02	$1.66e - 15$	0.26
GPT 5	Grok Code Fast 1	7.29	3220.87	$3.91e - 13$	0.23
GPT 5	Gemini 2.5 Pro	-4.58	3073.81	$4.79e - 06$	-0.15
Claude Sonnet 4	Gemini 2.5 Pro	-3.37	3223.77	0.00075	-0.11
GPT 5	Claude Sonnet 4	-1.17	4754.62	0.242	-0.03

E Statistical Analyses for RQ2

Prompting Comparisons within Models

As shown in Table 6, prompting strategy significantly affected alignment with authentic student errors, but the direction of the effect varied across models:

- For **GPT-4o**, Self-refine produced significantly worse alignment than either CoT or IO ($p < .001$, medium effects). This confirms the descriptive pattern that GPT-4o's alignment degrades under iterative refinement.
- For **GPT-5**, **Self-refine substantially reduced alignment** compared to both CoT ($p < .001$, $g = 0.43$) and IO ($p < .01$, $g = 0.28$), suggesting GPT-5's original outputs were **better** at simulating realistic mistakes.
- For **Claude Sonnet 4**, CoT and IO were **significantly better** than Self-refine ($p < .003$ and $p < .02$ respectively), though CoT and IO did not differ significantly ($p = 0.390$). This reinforces Claude's overall robustness but indicates iterative refinement **decreases** alignment.
- For **Gemini 2.5 Pro**, results show that CoT ($p < .001$, $g = 0.59$) and Self – refine ($p < .001$, $g = -0.49$ when compared to IO) were **significantly better** than IO (baseline), with CoT **showing the largest effect size**.
- For **Grok Code Fast 1**, **Self-refine significantly reduced alignment** compared to both IO ($p < .001$, $g = 0.60$) and CoT ($p < .001$, $g = 0.52$). This suggests that Self – refine **led to the worst match** to student errors.

Model Comparisons

Table 7 shows pairwise model comparisons:

- **GPT-5** consistently produced errors much **more similar** to students compared to all other models, with extremely large positive effect sizes (g between 0.35 and 1.79). This confirms GPT-5's position as the most consistent simulator of student-like mistakes.
- **Gemini 2.5 Pro** significantly outperformed Grok Code Fast 1 by a large margin ($g = 0.95$) and was significantly **worse** than GPT-5 ($g = 0.35$) and **Claude Sonnet 4** ($g = -1.29$ when compared to Claude).
- **Claude Sonnet 4** significantly underperformed GPT-5 ($g = -1.79$ when compared to GPT-5) and **Gemini 2.5 Pro** ($g = -1.29$), but **outperformed** Grok Code Fast 1 ($g = -0.27$ when compared to Claude) and GPT-4o ($g = 0.40$).
- **GPT-4o** occupied a middle ground: **significantly worse** than Claude ($g = 0.40$) and **Gemini 2.5 Pro** ($g = -0.82$), but **significantly better** than GPT-5 ($g = -1.26$). Compared to Grok Code Fast 1, GPT – 4o showed a small, significant advantage ($g = 0.13$).

In summary, these results statistically confirm that **GPT-5 provides the closest approximation of authentic student errors**, followed by a mixed group including Gemini 2.5 Pro, Claude Sonnet 4, GPT-4o, and Grok Code Fast 1. Prompting design modulates these effects, but model choice remains the dominant factor.

Table 6. Independent t -test results for prompt strategies by model (RQ2).

Model	Group A	Group B	t -statistic	df	p -value	Hedges' g
GPT 4o	CoT	selfrefine	-7.83	425.00	$3.85e - 14$	-0.70
GPT 4o	CoT	baseline	-4.02	395.03	$6.92e - 05$	-0.39
GPT 4o	baseline	selfrefine	-3.79	462.87	$1.68e - 04$	-0.35
GPT 5	CoT	selfrefine	4.68	429.58	$3.79e - 06$	0.43
GPT 5	baseline	selfrefine	3.09	433.91	$2.11e - 03$	0.28
GPT 5	CoT	baseline	1.95	582.74	$5.14e - 02$	0.16
Claude Sonnet 4	CoT	selfrefine	3.01	438.07	$2.77e - 03$	0.28
Claude Sonnet 4	baseline	selfrefine	2.41	508.46	$1.63e - 02$	0.21
Claude Sonnet 4	CoT	baseline	0.86	474.57	0.390	0.08
Gemini 2.5 Pro	CoT	baseline	5.93	390.17	$6.69e - 09$	0.59
Gemini 2.5 Pro	baseline	selfrefine	-5.09	407.99	$5.36e - 07$	-0.49
Gemini 2.5 Pro	CoT	selfrefine	1.15	395.34	0.249	0.11
Grok Code Fast 1	baseline	selfrefine	6.62	365.54	$1.26e - 10$	0.60
Grok Code Fast 1	CoT	selfrefine	4.88	277.03	$1.78e - 06$	0.52
Grok Code Fast 1	CoT	baseline	-1.41	396.83	0.158	-0.14

Table 7. Independent t -test results for model comparisons (RQ2).

Group A	Group B	t -statistic	df	p -value	Hedges' g
GPT 5	Claude Sonnet 4	36.08	1434.22	$2.13e - 203$	1.79
GPT 5	Grok Code Fast 1	26.53	1343.54	$4.92e - 125$	1.41
GPT 4o	GPT 5	-24.31	1449.66	$8.98e - 110$	-1.26
Claude Sonnet 4	Gemini 2.5 Pro	-22.68	965.40	$1.20e - 91$	-1.29
Gemini 2.5 Pro	Grok Code Fast 1	16.61	1161.37	$9.58e - 56$	0.95
GPT 4o	Gemini 2.5 Pro	-14.65	1200.94	$7.45e - 45$	-0.82
GPT 4o	Claude Sonnet 4	7.32	1187.92	$4.45e - 13$	0.40
GPT 5	Gemini 2.5 Pro	6.53	1242.96	$9.66e - 11$	0.35
Claude Sonnet 4	Grok Code Fast 1	-4.66	1064.99	$3.55e - 06$	-0.27
GPT 4o	Grok Code Fast 1	2.25	1239.35	0.024	0.13

F The Editing Distance Results by Problem Sets

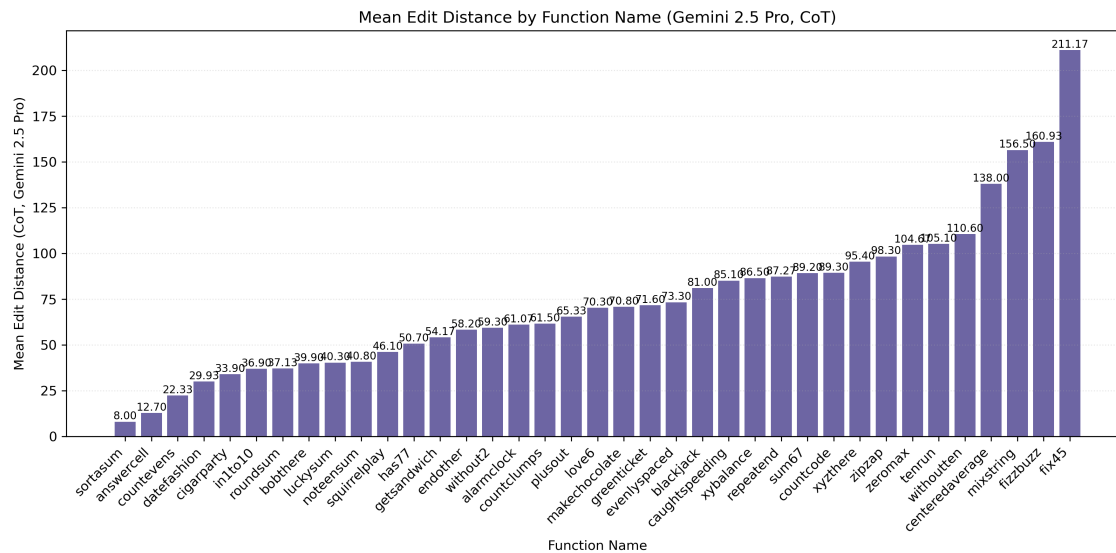


Fig. 6. Problem-Level Variation in Mean Intra-Model Edit Distance (Gemini 2.5 Pro, CoT Prompting).

Note: This figure reports the average pairwise edit distance among multiple LLM-generated code solutions for each function name (problem). Results are specific to the Gemini 2.5 Pro model under the chain-of-thought (CoT) prompting technique. Larger values indicate higher diversity across generated outputs, while smaller values suggest more consistent code generations.

G LLM-generated code Example for Two Problems based on Gemini Pro 2.5 and CoT

G.1 Problem: sortasum (Low struggling level, Low diversity):

The following problem description was provided to the model as input:

Write a function in Java that implements the following logic: Given 2 ints, a and b, return their sum. However, sums in the range 10..19 inclusive, are forbidden, so in that case just return 20.

Given the prompt in Appendix A.2, the model produced the following submissions.

Submission 1:

```
/*Reasoning: The student calculates the sum correctly but implements the range check
using exclusive boundaries (`> 10` and `< 19`). This approach fails to forbid
sums that are exactly 10 or 19, which should be included in the forbidden range.*/
```java
class Solution {
 public int sortaSum(int a, int b) {
 int sum = a + b;
 if (sum > 10 && sum < 19) {
```

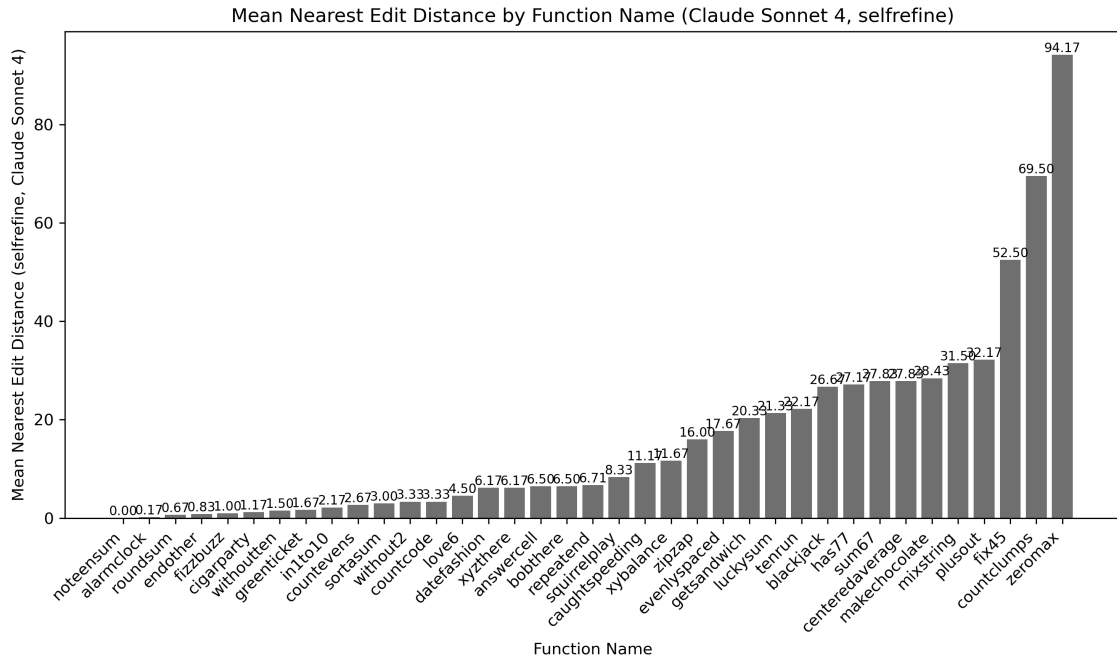


Fig. 7. Problem-Level Variation in Mean Nearest Edit Distance (Claude Sonnet 4, Self-Refine Prompting)

Note: This figure shows the average nearest edit distance between LLM-generated code and human references across different function names (problem). Results are specific to the Claude Sonnet 4 model under the self-refine prompting technique, illustrating substantial variation in function-level alignment. Higher values indicate greater divergence from human submissions, highlighting functions with more challenging alignment.

```

 return 20;
 }
 return sum;
}
...

Submission 2:
/*Reasoning: The student correctly identifies the lower bound of the forbidden range
but makes an off-by-one error on the upper bound. The condition `sum < 19`
incorrectly allows the sum of 19 to be returned instead of 20.*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum >= 10 && sum < 19) {
            return 20;

```

```

1873         } else {
1874             return sum;
1875         }
1876     }
1877 }
1878 ---
1879
1880
1881 Submission 3:
1882 /*Reasoning: The student correctly identifies the forbidden range but inverts the
1883    logic of the conditional. The code returns the original sum for values inside the
1884    forbidden range and 20 for values outside of it.*/
1885 ```java
1886 class Solution {
1887     public int sortaSum(int a, int b) {
1888         int sum = a + b;
1889         if (sum >= 10 && sum <= 19) {
1890             return sum;
1891         } else {
1892             return 20;
1893         }
1894     }
1895 }
1896 }
1897 ---
1898
1899
1900 Submission 4:
1901 /*Reasoning: The student misunderstands the meaning of a "range" and only checks the
1902    endpoints. This implementation correctly forbids sums of 10 and 19 but fails to
1903    forbid any of the numbers in between (11 through 18).*/
1904 ```java
1905 class Solution {
1906     public int sortaSum(int a, int b) {
1907         int sum = a + b;
1908         if (sum == 10 || sum == 19) {
1909             return 20;
1910         }
1911         return sum;
1912     }
1913 }
1914 ---
1915
1916
1917
1918 Submission 5:
1919 /*Reasoning: The student correctly identifies the lower bound of the forbidden range
1920    but extends the upper bound one too far. The condition `sum <= 20` incorrectly
1921    includes 20 in the set of forbidden sums, which is not part of the requirement.*/
1922 ```java
1923
1924

```

```

1925 class Solution {
1926     public int sortaSum(int a, int b) {
1927         int sum = a + b;
1928         if (sum >= 10 && sum <= 20) {
1929             return 20;
1930         }
1931         return sum;
1932     }
1933 }
1934 }
1935 ...
1936
1937
1938
1939

```

G.2 Problem: fix45 (High struggling level, High diversity):

The following problem description was provided to the model as input:

Given an array, return an array that contains exactly the same numbers as the original array, but rearranged so that every 4 is immediately followed by a 5. Do not move the 4's, but every other number may move. The array contains the same number of 4's and 5's, and every 4 has a number after it that is not a 4. In this version, 5's may appear anywhere in the original array.

Given the prompt in Appendix A.2, the model produced the following submissions.

```

1950 Submission 1:
1951 /*Reasoning: The student attempts to find all `5`s that are not preceded by a `4` to
1952    identify which ones are available to be moved. The check for this condition is
1953    flawed because it starts its loop at index 1 to avoid an out-of-bounds error,
1954    thus completely ignoring a `5` if it happens to be the first element of the
1955    array.*/
1956
1957 ```java
1958 import java.util.ArrayList;
1959 import java.util.List;
1960
1961 class Solution {
1962     public int[] fix45(int[] nums) {
1963         List<Integer> fiveIndices = new ArrayList<>();
1964         List<Integer> toFillIndices = new ArrayList<>();
1965
1966         for (int i = 0; i < nums.length; i++) {
1967             if (nums[i] == 4 && i + 1 < nums.length && nums[i + 1] != 5) {
1968                 toFillIndices.add(i + 1);
1969             }
1970             // This check incorrectly skips index 0
1971             if (i > 0 && nums[i] == 5 && nums[i - 1] != 4) {
1972                 fiveIndices.add(i);
1973             }
1974         }
1975     }
1976

```

```

1977     }
1978
1979     for (int i = 0; i < toFillIndices.size(); i++) {
1980         int temp = nums[toFillIndices.get(i)];
1981         nums[toFillIndices.get(i)] = 5;
1982         nums[fiveIndices.get(i)] = temp;
1983     }
1984
1985
1986     return nums;
1987 }
1988 }
1989 ...
1990
1991
1992 Submission 2:
1993 /*Reasoning: The student correctly identifies when a `4` is not followed by a `5`,
1994    but instead of performing a proper swap, they simply overwrite the incorrect
1995    value with a `5`. This causes a number to be lost from the array, violating the
1996    problem's constraint to retain all original numbers.*/
1997 ```java
1998 class Solution {
1999     public int[] fix45(int[] nums) {
2000         for (int i = 0; i < nums.length - 1; i++) {
2001             if (nums[i] == 4 && nums[i + 1] != 5) {
2002                 for (int j = 0; j < nums.length; j++) {
2003                     // Find a 5 that is not already correctly placed
2004                     if (nums[j] == 5) {
2005                         if (j > 0 && nums[j - 1] == 4) {
2006                             continue;
2007                         }
2008                         // The original value at nums[j] is never used,
2009                         // and the value at nums[i+1] is overwritten.
2010                         nums[i + 1] = 5;
2011                         break;
2012                     }
2013                 }
2014             }
2015         }
2016         return nums;
2017     }
2018 }
2019
2020 }
2021 }
2022 ...
2023
2024
2025 Submission 3:

```

```

2029  /*Reasoning: The student uses nested loops where the outer loop finds a `4` and the
2030      inner loop searches for any `5` starting from the beginning of the array. This
2031      approach is flawed because it may select a `5` that has already been correctly
2032      placed after a different `4` in a previous iteration, undoing correct work.*/
2033  ```java
2034  class Solution {
2035      public int[] fix45(int[] nums) {
2036          for (int i = 0; i < nums.length - 1; i++) {
2037              if (nums[i] == 4 && nums[i + 1] != 5) {
2038                  // This inner loop always restarts from the beginning
2039                  for (int j = 0; j < nums.length; j++) {
2040                      if (nums[j] == 5) {
2041                          int temp = nums[i + 1];
2042                          nums[i + 1] = nums[j];
2043                          nums[j] = temp;
2044                          // It might have just moved a 5 that was already
2045                          // correctly placed after a previous 4.
2046                          break;
2047                      }
2048                  }
2049              }
2050          }
2051          return nums;
2052      }
2053  }
2054  ```
2055  Submission 4:
2056  /*Reasoning: The student gathers the indices of all `5`s into a list, but fails to
2057      distinguish between `5`s that are free to move and those already correctly
2058      placed. When a `4` needs a `5`, the code swaps its subsequent element with the
2059      next `5` from the list, which might be one that was already in a correct
2060      position.*/
2061  ```java
2062  import java.util.ArrayList;
2063  import java.util.List;
2064
2065  class Solution {
2066      public int[] fix45(int[] nums) {
2067          List<Integer> fiveLocations = new ArrayList<>();
2068          for (int i = 0; i < nums.length; i++) {
2069              if (nums[i] == 5) {
2070                  fiveLocations.add(i);
2071              }
2072          }
2073      }
2074  }
2075  ```

```

```
2081
2082     int fiveIndex = 0;
2083     for (int i = 0; i < nums.length - 1; i++) {
2084         if (nums[i] == 4 && nums[i + 1] != 5) {
2085             int swapPos = fiveLocations.get(fiveIndex);
2086             int temp = nums[i + 1];
2087             nums[i + 1] = 5;
2088             nums[swapPos] = temp;
2089             fiveIndex++;
2090         }
2091     }
2092     return nums;
2093 }
2094 }
2095 }
2096 }
2097 ...
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
```