Lab Report on:

Software Debugging

SWE 4802: Software Maintenance Lab

Lutfun Nahar Lota

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology

Submitted By:

Abrar Mahabub (200042103)

Mashrur Ahsan (200042115)

Nafisa Maliyat (200042133)

Shanta Maria (200042172)

Table of Contents

1. Introduction

This report is based on a debugging-focused task where a complex ICPC problem was selected, solved, and then improved through a structured debugging process. The chosen problem is the ICPC 2018 Problem F (Go with the Flow). After writing the initial solution, some issues arose during testing, which were resolved through careful analysis. The goal was to understand and document how errors were identified and fixed step by step. This report highlights the problem we worked on and explains the debugging process we took.

2. Problem

ICPC 2018 Problem F - Go with the Flow

In typesetting, a “river” is a string of spaces formed by gaps between words that extends down several lines of text. For instance, Figure F.1 shows several examples of rivers highlighted in red (text is intentionally blurred to make the rivers more visible).

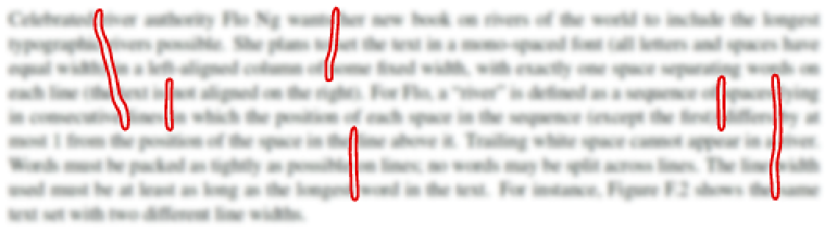
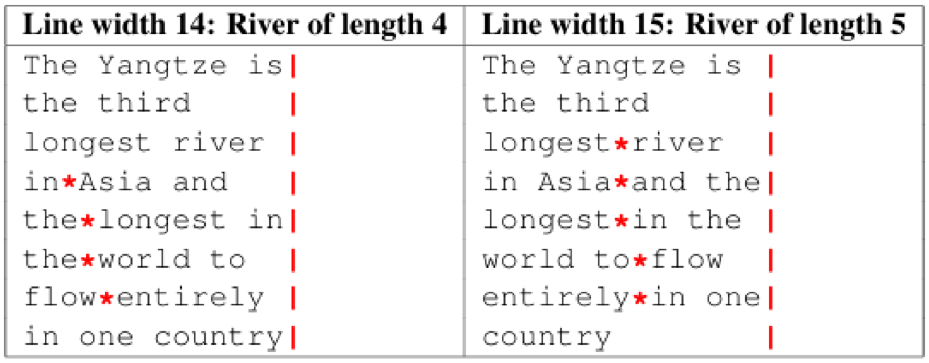


Figure F.1: Examples of rivers in typeset text.

Celebrated river authority Flo Ng wants her new book on rivers of the world to include the longest typographic rivers possible. She plans to set the text in a mono-spaced font (all letters and spaces have equal width) in a left-aligned column of some fixed width, with exactly one space separating words on each line (the text is not aligned on the right). For Flo, a “river” is defined as a sequence of spaces lying in consecutive lines in which the position of each space in the sequence (except the first) differs by at most 1 from the position of the space in the line above it. Trailing white space cannot appear in a river. Words must be packed as tightly as possible on lines; no words may be split across lines. The line width used must be at least as long as the longest word in the text. For instance, Figure F.2 shows the same text set with two different line widths.



Line width 14: River of length 4	Line width 15: River of length 5
`The Yangtze is`	`The Yangtze is`
`the third`	`the third`
`longest river`	`longest*river`
`in*Asia and`	`in Asia*and the`
`the*longest in`	`longest*in the`
`the*world to`	`world to*flow`
`flow*entirely`	`entirely*in one`
`in one country`	`country`

Figure F.2: Longest rivers (*) for two different line widths.

Given a text, you have been tasked with determining the line width that produces the longest river of spaces for that text.

Input

The first line of input contains an integer n (2 ≤ n ≤ 2500) specifying the number of words in the text. The following lines of input contain the words of text. Each word consists only of lowercase and uppercase letters, and words on the same line are separated by a single space. No word exceeds 80 characters.

Output

Display the line width for which the input text contains the longest possible river, followed by the length of the longest river. If more than one line width yields this maximum, display the shortest such line width.

Sample Input 1

```
21
The Yangtze is the third longest
river in Asia and the longest in
the world to flow
entirely in one country
```

Sample Output 1

```
15 5
```

Sample Input 2

```
25
When two or more rivers meet at
a confluence other than the sea
the resulting merged river takes
the name of one of those rivers
```

Sample Output 2

```
21 6
```

3 Solution

3.1 Initial Attempt

```python
import sys

def solve_river_flow():
    lines = sys.stdin.read().strip().split('\n')
    n = int(lines[0])

    words = []
    for i in range(1, len(lines)):
        words.extend(lines[i].split())

    words = words[:n]

    min_width = max(len(word) for word in words)

    total_chars = sum(len(word) for word in words) + len(words) - 1
    max_width = total_chars

    best_width = min_width
    best_river_length = 0

    for width in range(min_width, max_width + 1):
        lines_formed = create_lines(words, width)

        if len(lines_formed) <= 1:
            break

        if len(lines_formed) <= best_river_length:
            continue

        river_length = find_longest_river(lines_formed)
        if river_length > best_river_length:
            best_river_length = river_length
            best_width = width

    print(best_width, best_river_length)


def create_lines(words, width):
    lines = []
    current_line = ""

    for word in words:
        if current_line == "":
            current_line = word
        else:
            if len(current_line) + 1 + len(word) <= width:
                current_line += " " + word
            else:
                lines.append(current_line)
                current_line = word

    if current_line:
        lines.append(current_line)

    return lines


def find_longest_river(lines):
    if len(lines) <= 1:
        return 0

    space_positions = []
    for line in lines:
        positions = []
        for i, char in enumerate(line):
            if char == ' ':
                positions.append(i)
        space_positions.append(positions)

    max_river_length = 0

    for start_pos in space_positions[0]:
        river_length = find_river_from_position(space_positions, 0,
start_pos)
        max_river_length = max(max_river_length, river_length)

    return max_river_length


def find_river_from_position(space_positions, start_line, start_pos):
    river_length = 1
    current_pos = start_pos

    for line_idx in range(start_line + 1, len(space_positions)):
        found_continuation = False

        for pos in space_positions[line_idx]:
            if abs(pos - current_pos) <= 1:
                river_length += 1
                current_pos = pos
                found_continuation = True
                break
        if not found_continuation:
            break

    return river_length

if __name__ == "__main__":
    solve_river_flow()
```

3.2 Final Solution

```
import sys

def longest_river_and_linecount(words, width):
    lines_spaces = []
    current_len = 0
    current_spaces = []
    for w in words:
        if current_len == 0:
            current_len = len(w)
        else:
            if current_len + 1 + len(w) <= width:
                current_spaces.append(current_len)
                current_len += 1 + len(w)
            else:
                lines_spaces.append(current_spaces)
                current_spaces = []
                current_len = len(w)
    lines_spaces.append(current_spaces)

    best_river = 0
    prev_dp = {}
    first_line = True
    for spaces in lines_spaces:
        dp = {}
        for p in spaces:
            length = 1
            if not first_line:
                for d in (-1, 0, 1):
                    if (p + d) in prev_dp:
                        length = max(length, prev_dp[p + d] + 1)
            dp[p] = length
            best_river = max(best_river, length)
        prev_dp = dp
        first_line = False

    return best_river, len(lines_spaces)
```

```python
def find_next_width(words, width):
    next_w = None
    current_len = 0
    for w in words:
        lw = len(w)
        if current_len == 0:
            current_len = lw
        else:
            if current_len + 1 + lw <= width:
                current_len += 1 + lw
            else:
                threshold = current_len + 1 + lw
                if threshold > width:
                    if next_w is None or threshold < next_w:
                        next_w = threshold
                current_len = lw
    return next_w if next_w is not None else width + 1


def find_best_width(words):
    min_width = max(len(w) for w in words)
    max_width = sum(len(w) for w in words) + len(words) - 1

    best_width = min_width
    best_length = 0
    width = min_width

    while width <= max_width:
        river_len, num_lines = longest_river_and_linecount(words, width)
        if river_len > best_length:
            best_length = river_len
            best_width = width
        if river_len >= num_lines:
            break
        width = find_next_width(words, width)
    return best_width, best_length



def main():
    input_lines = sys.stdin.read().splitlines()
    n = int(input_lines[0])
    words = []
    for line in input_lines[1:]:
        words.extend(line.strip().split())
    assert len(words) == n, f"Expected {n} words, got {len(words)}"

    width, length = find_best_width(words)
    print(f"{width} {length}")
```

```python
if __name__ == "__main__":
    main()
```

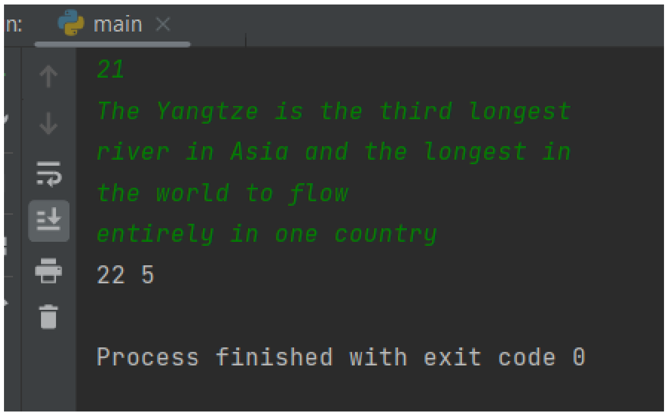
4 Debugging

When a submitted solution gives the status ‘not accepted’, the system detects and displays the error - either the answer is wrong for a specific test case or the time limit was exceeded. In our initial solution, we faced issues where the code was giving wrong answers on some test cases. The debugging process was then conducted in a step-by-step manner in order to fix the underlying issue.

4.1 Examining Failing Test Case

The first step is to identify the test case that caused the submitted solution not to be accepted. This involves comparing the answer with the expected result in order to compare how the logic works for that test case.

This is the test case that resulted in a wrong answer with our initial attempt:



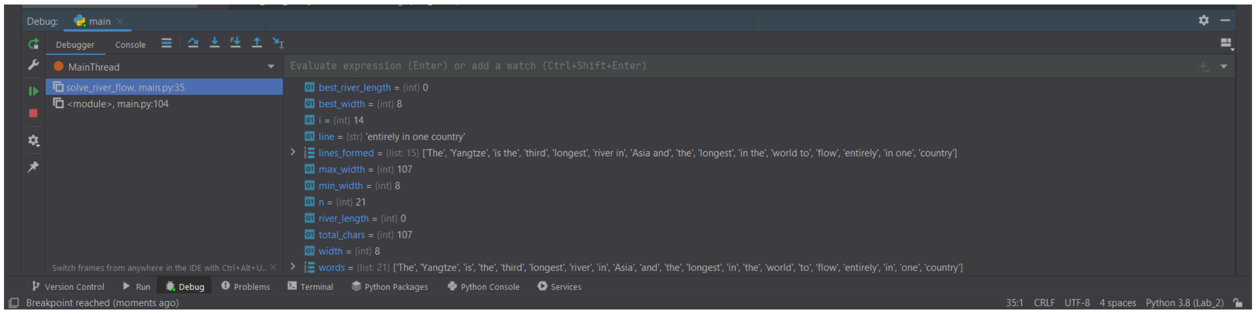
```
21
The Yangtze is the third longest
river in Asia and the longest in
the world to flow
entirely in one country
22 5

Process finished with exit code 0
```

We recreated these test cases locally in a debugging environment to observe how the logic flow occurs for the specific inputs. It is also helpful to go through the code for that input on paper to visualize where the logic might be wrong and correct the specific lines of code. This results in the identification of logic errors more easily. Often, it is also helpful to run additional test cases around the failing input to grasp the pattern of failure better.

4.2 Using Debugging Tools

Modern IDEs come equipped with built-in debugging tools that make the process of identifying and resolving errors in the code easier. They allow developers to execute the program step by step and keep track of how the values of the variables change by setting breakpoints.

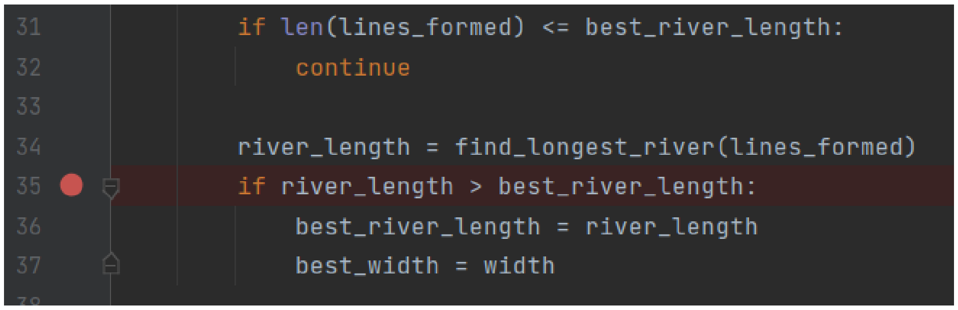
Debugging panel in PyCharm is as below:



```
solve_river_flow, main.py:35
<module>, main.py:104
Switch frames from anywhere in the IDE with Ctrl+Alt+U..
```

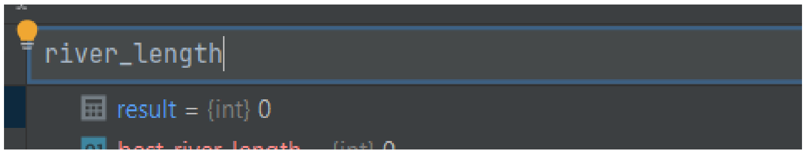
```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
best_river_length = {int} 0
best_width = {int} 8
i = {int} 14
line = {str} 'entirely in one country'
lines_formed = {list: 15} ['The', 'Yangtze', 'is the', 'third', 'longest', 'river in', 'Asia and', 'the', 'longest', 'in the', 'world to', 'flow', 'entirely', 'in one', 'country']
max_width = {int} 107
min_width = {int} 8
n = {int} 21
river_length = {int} 0
total_chars = {int} 107
width = {int} 8
words = {list: 21} ['The', 'Yangtze', 'is', 'the', 'third', 'longest', 'river', 'in', 'Asia', 'and', 'the', 'longest', 'in', 'the', 'world', 'to', 'flow', 'entirely', 'in', 'one', 'country']
Breakpoint reached (moments ago)
```

For example, here we used a breakpoint to check the final value of the variable *river_length*:



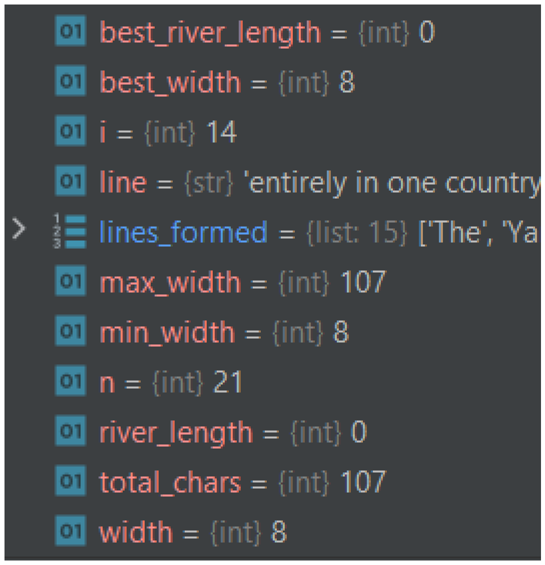
```
        if len(lines_formed) <= best_river_length:
            continue

        river_length = find_longest_river(lines_formed)
        if river_length > best_river_length:
            best_river_length = river_length
            best_width = width
```

The value of the *river_length* variable can be found in the debugging panel by searching it as such:



```
river_length
    result = {int} 0
```

We simulated the program using the failing test case inputs and using the debugging tool in PyCharm, introducing breakpoints in the program in order to keep track of how the variables are being updated. This helped to find out what the variable value should have been and its actual value at that point of execution, and pinpoint which logic flow error might have resulted in the process.]

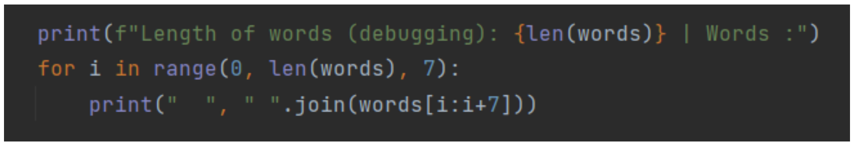


```
best_river_length = {int} 0
best_width = {int} 8
i = {int} 14
line = {str} 'entirely in one country
> lines_formed = {list: 15} ['The', 'Ya
max_width = {int} 107
min_width = {int} 8
n = {int} 21
river_length = {int} 0
total_chars = {int} 107
width = {int} 8
```

4.3 Using print() Statements

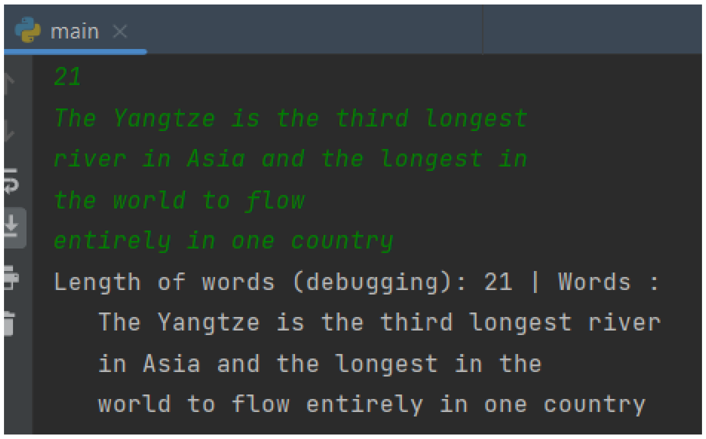
While this is a basic method, placing print() statements at intervals is a very valuable method of debugging, specifically for quick verification of logic in specific parts of the code. We used multiple print statements to track how variables are changed and verify that our control flow paths are accurate. Debugging tools achieve similar results, but this is a faster method that consists of running the code in seconds to observe the terminal for displaying intermediate values. Specific messages were added to ensure we know which step is producing the print statements.

When any print statement showed a different value being displayed than the one we had anticipated, the portion of the code was isolated for fixing.



```
print(f"Length of words (debugging): {len(words)} | Words :")
for i in range(0, len(words), 7):
    print("  ", " ".join(words[i:i+7]))
```

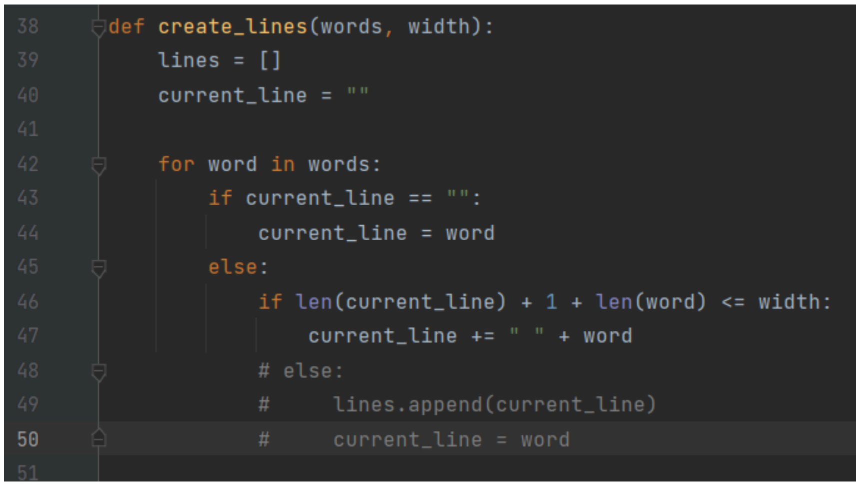
For example, here we used print to check if the words are being sliced properly otherwise the whole layout would be wrong:



```
21
The Yangtze is the third longest
river in Asia and the longest in
the world to flow
entirely in one country
Length of words (debugging): 21 | Words :
   The Yangtze is the third longest river
   in Asia and the longest in the
   world to flow entirely in one country
```

4.4 Selectively Isolating Code

This can help to effectively narrow down the sections of code that can be responsible for giving the wrong answers. Isolating code involves commenting out parts of the code in order to test whether the error lies in the remaining code. Usually, it is done in a step-by-step manner where the first part of the code is tested and verified to ensure that it does not contain any errors. The next parts are commented during that execution, so we can verify that a specific code segment works as intended. The next part is then commented iteratively to check for the part of the code that might be responsible for the wrong answer.

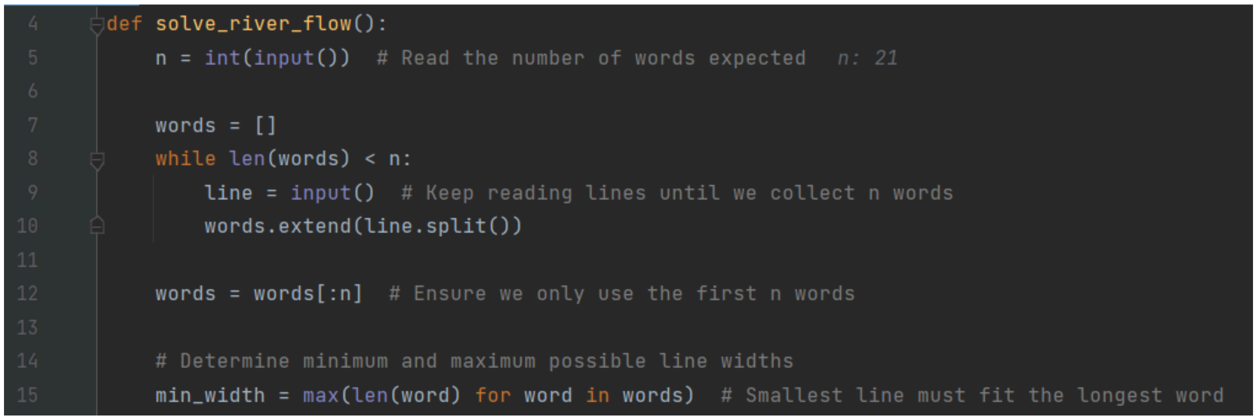


```python
def create_lines(words, width):
    lines = []
    current_line = ""

    for word in words:
        if current_line == "":
            current_line = word
        else:
            if len(current_line) + 1 + len(word) <= width:
                current_line += " " + word
            # else:
            #     lines.append(current_line)
            #     current_line = word
```

Commenting out the *else* part of the code helps us understand if the correct input is being recognized by the *if* statement and if the *if* condition statements are being executed correctly.

4.5 Rubber Duck Debugging

Rubber duck debugging is a technique where we verbally explained the code functionality line by line in order to determine whether the written code reflects the solution we have thought of. The code was explained to a fellow teammate to articulate the logic explicitly which helps identify any oversights that were not considered during the original problem-solving steps. This can be extremely helpful for very subtle logical errors that usually are not caught by other debugging methods.



```python
def solve_river_flow():
    n = int(input())  # Read the number of words expected   n: 21

    words = []
    while len(words) < n:
        line = input()  # Keep reading lines until we collect n words
        words.extend(line.split())

    words = words[:n]  # Ensure we only use the first n words

    # Determine minimum and maximum possible line widths
    min_width = max(len(word) for word in words)  # Smallest line must fit the longest word
```

The comments here can help explain the logic to a fellow programmer hence performing the Rubber Duck debugging.

4.6 Additional Research

To avoid the Time Limit Exceeded (TLE) error, we searched for any small similar problems other programmers might have encountered during the solution process using the same type of data structure, solution process, or input-taking methods. These also introduced some standard ways of optimization that sped up the execution time for large test cases, like using a more search efficient data structure. Sources of research included basic Google searches and exploration of different helpful links to try out optimization methods. Using trial and error with different suggestions, the TLE error was dealt with.

We mainly used Google and StackOverflow:

We found some suggestions regarding common problems of TLE in Python:

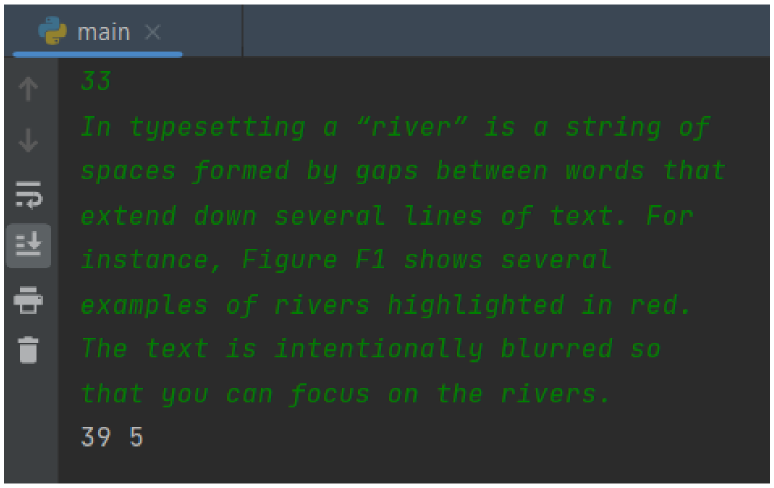
len(arr) has complexity of O(N), and you are doing it N times, so overall complexity is N^2, perhaps this complexity does not meet the required one. – Asif Mohammed Aug 29, 2022 at 14:00

your function is empty. put some code inside it or `pass` if you want to leave it empty – user2261062 Aug 29, 2022 at 14:00

Add a comment

4.7 Dry Running with Custom Inputs

Dry running is a process of manually running the code with custom inputs to see how the logic works at each stage. This method aids in comprehending the control flow and confirming that the logic is as intended. We used custom inputs to observe how lines were made and how rivers were forming. We were able to detect any mismatch between expected and actual behavior, which is helpful for the early debugging stage.



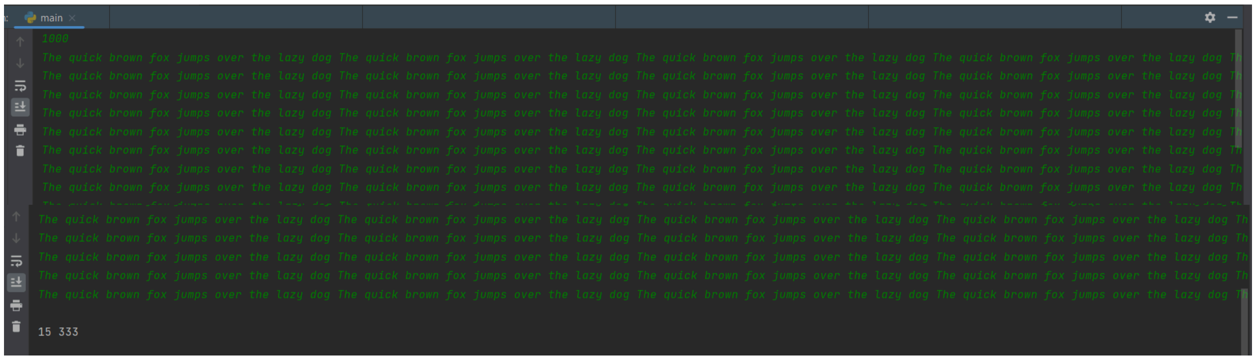
```
33
In typesetting a “river” is a string of
spaces formed by gaps between words that
extend down several lines of text. For
instance, Figure F1 shows several
examples of rivers highlighted in red.
The text is intentionally blurred so
that you can focus on the rivers.
39 5
```

4.8 Checking Editorials or Hints

After struggling with issues for a while, we decided to check the editorials and online discussions. We got an idea of what direction we might be missing. That is when we realized our code was checking too many line widths unnecessarily. And the editorials pointed out some tricks that were ignored by us and made us adjust our problem-solving approach.

4.9 Testing With Large Inputs

We executed the program with large test cases of different lengths. Running these test cases locally helped us replicate the issues we were seeing.



```
1000
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th
The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog The quick brown fox jumps over the lazy dog Th

15 333
```

We noted how long the code was taking and whether our changes made any difference. It was a reliable way to measure improvement and gave a better sense of what kind of inputs the code needed to handle efficiently.