Seed Programming

Competitive Programming Basics

Lecture 05: Complete Notes

1 Introduction to CodeChef

CodeChef is an online judge: a platform where programmers write code, submit it, and have it tested automatically against hidden test cases. Unlike running a program in a personal terminal, an online judge:

- feeds input strictly through standard input (`stdin`), without interactive prompts;
- compares the program's exact printed output with an expected output file; and
- reports a wrong answer when extra text appears, even if the calculation itself is correct.

This is the biggest mental shift from *coding in class* to *coding for a judge*. It explains several common mistakes discussed in Sections 3 and 4.

2 Popcorn Movie Ticket Problem

Problem link: CodeChef POPCORN7

2.1 Problem Statement

Chef has `X` money. A movie ticket costs `100`, and each popcorn bucket costs `50`. Find the maximum number of popcorn buckets Chef can buy after purchasing the ticket.

2.2 Explanation and Logic

After buying the ticket, Chef has `X - 100` left. Since each bucket costs `50`, the number of buckets is

$$\left\lfloor \frac{X - 100}{50} \right\rfloor.$$

The floor operation rounds down because Chef cannot buy a fraction of a bucket. Leftover money that cannot pay for another complete bucket is ignored.

In C++, integer division already discards the fractional part. In Python, use `//`; the `/` operator produces a floating-point result.

2.3 Python Solution

```python
X = int(input())
buckets = (X - 100) // 50
print(buckets)
```

2.4 Dry Runs

For `X = 250`:

- Money left: `250 - 100 = 150`.
- Buckets: `150 // 50 = 3`.

Output

```
3
```

For `X = 180`:

- Money left: `180 - 100 = 80`.
- Buckets: `80 // 50 = 1`.
- The remaining `30` is not enough for another bucket.

Output

```
1
```

Important. *If `X < 100`, the formula gives a negative value. Check the problem constraints to see whether `X` is guaranteed to be at least `100`, or whether the answer would need to be limited to zero.*

The time complexity is $O(1)$ per test case because the solution uses a fixed number of arithmetic operations and no loop.

3 POINTCAL: Score-Type Question

Problem link: CodeChef POINTCAL

3.1 The Problem with the Obvious Code

```
A, B, C = map(int, input("Enter three inputs: ").split())
print(A, B, C)
```

This fails on an online judge because:

- `input("Enter three inputs: ")` prints the prompt to standard output;
- the judge expects only the required computed answer; and
- input comes from a prepared test file, so there is no person waiting to read a prompt.

3.2 Corrected Code

```
A, B, C = map(int, input().split())
print(A, B, C)
```

3.3 Understanding `input().split()` and `map()`

Expression	Meaning
`input()`	Reads one complete line from `stdin` as a string.
`.split()`	Splits the string at whitespace. For example, `"3 5 7".split()` produces `['3', '5', '7']`.
`map(int, ...)`	Applies `int()` to every piece, converting the strings to integers.
`A, B, C = ...`	Unpacks the three converted values directly into three variables.

Important. *The golden rule for online judges is to keep `input()` bare. Never place a prompt inside it unless the problem explicitly requires interactive output.*

4 Jumping Frog Problem

Problem link: HackerEarth Jumping Frogs

A frog is trapped in a slippery well of height `H`. On each attempt, it jumps `J` units upward and, unless it has escaped, slips `S` units downward. The goal is to find the required number of jumps.

4.1 Simulation Version

```python
H, J, S = map(int, input().split())

if J <= S and H > J:
    print(-1)          # impossible: no lasting progress
else:
    steps = 0

    while True:
        H -= J
        steps += 1

        if H <= 0:
            break      # escaped; no slip after final jump

        H += S

    print(f"steps: {steps}")
```

The impossible case is checked first. If `J - S` is zero or negative and a single jump cannot clear the well, the frog is stuck forever. Otherwise, each loop iteration subtracts the jump from the remaining height, counts the jump, tests for escape, and adds the slip only if the frog is still inside.

4.2 Dry Run: `H = 10, J = 3, S = 1`

Jump	Before	After jump	Out?	After slip
1	10	7	No	8
2	8	5	No	6
3	6	3	No	4
4	4	1	No	2
5	2	−1	**Yes; break**	—

Output

```
steps:  5
```

4.3 Efficient Version Without Slip

When there is no slippage, compute the ceiling of `H / J` instead of looping:

```python
steps = H // J

if steps * J != H:
    steps += 1

print(f"steps: {steps}")
```

Integer division gives the complete jumps. If there is a remainder, one extra jump is needed.

4.4 Efficient Version with Slip

The slip-aware answer can also be calculated without simulating every jump:

```
H, J, S = map(int, input().split())

if J <= S and H > J:
    print(-1)
elif H <= J:
    print(1)
else:
    remaining = H - J
    net = J - S
    steps = 1 + (remaining + net - 1) // net
    print(steps)
```

The final jump is handled separately because it has no slip. Every earlier jump-and-slip cycle gains only `J - S`. The expression

$$\frac{\texttt{remaining} + \texttt{net} - 1}{\texttt{net}}$$

with integer floor division implements ceiling division.

For `H = 10`, `J = 3`, and `S = 1`:

- `remaining = 10 - 3 = 7`;
- `net = 3 - 1 = 2`; and
- `steps = 1 + ceil(7 / 2) = 1 + 4 = 5`.

Output

```
5
```

5 W3Schools Guide for Python

W3Schools Python is a general self-study resource for reviewing Python syntax and beginner-level concepts. No specific W3Schools program was covered in class for this lecture.

6 Rock Paper Scissors: Built Step by Step

6.1 Stage 1: Simple Version with `input()`

```
P1Name, P2Name = input("Two Names: ").split()

p1 = input(f"{P1Name}, enter your symbol (r/p/s): ")
p2 = input(f"{P2Name}, enter your symbol (r/p/s): ")

if ((p1 == 'r' and p2 == 's') or
        (p1 == 's' and p2 == 'p') or
        (p1 == 'p' and p2 == 'r')):
    print(f'{P1Name} Wins')
elif p2 == p1:
    print('Draw...')
else:
    print(f'{P2Name} wins')
```

This version works when both players enter exactly `r`, `p`, or `s` in lowercase. However, Python string

comparison is case-sensitive. If a player enters `'R'` instead of `'r'`, then `'R' == 'r'` is false. The comparison silently fails, and the game can announce the wrong result without raising an error.

6.2 Stage 2: Normalize Case with `lower()`

```
P1Name, P2Name = input("Two Names: ").split()

p1 = input(f"{P1Name}, enter your symbol (r/p/s): ")
p2 = input(f"{P2Name}, enter your symbol (r/p/s): ")

p1 = p1.lower()
p2 = p2.lower()

if ((p1 == 'r' and p2 == 's') or
        (p1 == 's' and p2 == 'p') or
        (p1 == 'p' and p2 == 'r')):
    print(f'{P1Name} Wins')
elif p2 == p1:
    print('Draw...')
else:
    print(f'{P2Name} wins')
```

The `lower()` method converts both entries to lowercase before they are compared. As a result, `'R'` and `'r'` are treated identically. The `upper()` method could also be used, provided the game conditions compared the normalized values with uppercase symbols.

This fixes the case-sensitivity problem, but a more important flaw remains.

6.3 Stage 3: Player 1's Move Is Visible

With ordinary `input()`, every character Player 1 types is echoed on the screen and remains visible after Enter is pressed. By the time Player 2 chooses a move, Player 2 can already see Player 1's choice.

This breaks the game: Player 2 can always choose the winning response or force a draw. The moves must therefore be collected without displaying the pressed keys. The final version replaces `input()` for the moves with a hidden single-key reader.

6.4 Stage 4: Final Fix with `getch()`

```
import os
import sys

if os.name == 'nt':
    import msvcrt

    def getch():
        char = msvcrt.getch()
        return char.decode('utf-8', errors='ignore')
else:
    import tty
    import termios

    def getch():
        fd = sys.stdin.fileno()
        old_settings = termios.tcgetattr(fd)

        try:
            tty.setraw(sys.stdin.fileno())
            char = sys.stdin.read(1)
```

```
        finally:
            termios.tcsetattr(
                fd, termios.TCSADRAIN, old_settings
            )

        return char

from os import system
system('cls' if os.name == 'nt' else 'clear')

P1Name, P2Name = input("Two Names: ").split()

print('Both of You enter your symbols: ')
p1 = getch()
print('*')
p2 = getch()
print('*')

p1 = p1.lower()
p2 = p2.lower()

if ((p1 == 'r' and p2 == 's') or
        (p1 == 's' and p2 == 'p') or
        (p1 == 'p' and p2 == 'r')):
    print(f'{P1Name} Wins')
elif p2 == p1:
    print('Draw...')
else:
    print(f'{P2Name} wins')
```

The `getch()` function reads one keypress immediately. The player does not need to press Enter, and the selected character is not echoed to the terminal. Each `print('*')` provides confirmation that a key was received without revealing the move.

The program uses `msvcrt.getch()` on Windows. On Linux and macOS, it temporarily places the terminal in raw mode, reads one character, and restores the original terminal settings in the `finally` block.

The screen-clearing command is also platform-aware:

- Windows uses `cls`.
- Linux and macOS use `clear`.

Together, hidden keypresses, case normalization, and cross-platform screen clearing produce a fair two-player terminal game.