1: Pattern 11 â€" Alternating 1-0 Triangle

1. Understand the Problem

- **Read & Identify:** Given an integer n, print n lines where line i (1-indexed) contains i numbers, alternating 1 and 0, starting with 1 on odd-numbered lines and 0 on even-numbered lines
- **Goal:** Recreate the displayed pattern exactly for any n.
- **Paraphrase:** Paraphrase: For each row i from 1 to n, print i values alternating between 1 and 0; if the row number is odd, start with 1, otherwise start with 0.

---

2. Input, Output, & Constraints

- **Input:** Single integer n (number of rows).
- **Output:** n lines, line i containing i space-separated digits (1/0) forming the alternating pattern.

Constraints:

- 1 â‰¤ n â‰¤ 10^5 (practical limits for printing depend on environment; very large n will be I/O heavy)
- Time complexity target: O(nÂ²) is acceptable because output size is Î˜(nÂ²).

---

3. Examples & Edge Cases

Example(n = 5):

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

Edge Case Checklist:

- n = 1 â†' prints 1
- small n values (2,3)
- large n â†' ensure efficient printing (use buffered output)
- check behavior for n = 0 (problem typically assumes n â‰¥ 1

---

4. Approaches

Approach 1:Direct Pattern Generation (Simple & Clear)

Idea: For each row i from 1..n: Determine starting value start = (i % 2 == 1) ? 1 : 0. Print i values, toggling (val = 1 - val) after each printed number.

Pseudocode:

```
for i from 1 to n:
    if i is odd:
        val = 1
    else:
        val = 0
    for j from 1 to i:
        print val (with space if needed)
        val = 1 - val
    print newline
```

Complexity:

- Time: O(nÂ²) â€" you must print O(nÂ²) numbers (1 + 2 + ... + n).
- Space:O(1) extra (excluding output buffer).

Approach 2: Using Row Index Parity and j Parity (Alternative formulation)

Idea: You can compute the value at position j in row i as: value = (i + j) % 2 == 0 ? 1 : 0 if you want to use an arithmetic formula (check indexing convention). This avoids explicit toggling, though performance is equivalent.

Pseudocode:

```
for i from 1 to n:
    for j from 1 to i:
        val = ((i + j) % 2 == 0) ? 1 : 0
        print val
    newline
```

Complexity:

- Time:O(nÂ²)
- Space: O(1)

---

6. Justification / Proof of Optimality

- Printing every required number is necessary, total output size is Î˜(nÂ²) (sum of 1..n). Any correct solution must produce that many tokens, so O(nÂ²) time is optimal up to constant factors for this problem.

- Both approaches produce correct alternating values; the toggle method is straightforward and avoids repeated arithmetic, while the formula method is compact and declarative.

7. Variants / Follow-Ups

- Change separators (no spaces, commas).
- Start each row with the opposite bit (i.e., always start with 0).
- Print a similar pattern in a matrix/2D grid shape.
- Convert to characters (A/B or X/O) instead of 1/0.

Q10: Pattern 21 -Hollow Square Pattern

1. Input, Output, & Constraints

- **Input:**

```
5
```

- **Output:**

```
*****
*   *
*   *
*   *
*****
```

Constraints:

- 1 â‰¤ n â‰¤ 26 (English alphabets)

---

2. Examples & Edge Cases

Example 1 (edge case): Input:

```
2
```

Output:

```
**
**
```

---

3. Approaches

Approach 1: Using Nested Loops

- **Idea:**
  - Loop through each row
  - If row is first or last â†’ print all *
  - Otherwise â†’ print * at first and last column, spaces in between

Pseudocode:

```
function printHollowSquare(n):
    for i in range(1, n+1):
        for j in range(1, n+1):
            if i == 1 or i == n or j == 1 or j == n:
                print("*", end="")
            else:
                print(" ", end="")
        print()   # New line after each row
```

Complexity:

- Time: O(n^2) â†’ Nested loops for n rows Ã— n columns
- Space: O(1) â†’ Only loop variables

Approach 2: String Concatenation (Optional)

- **Idea:**
  - Precompute strings for first/last row and middle rows
  - Print first/last row directly, print middle row n-2 times

Pseudocode:

```
function printHollowSquare(n):
    full_row = "*" * n
    middle_row = "*" + " " * (n-2) + "*" if n > 1 else "*"

    print(full_row)
    for i in range(1, n-1):
        print(middle_row)
    if n > 1:
        print(full_row)
```

Complexity:

- Time: O(n^2) â†’ Still iterating over n rows Ã— n columns
- Space: O(1)â†’ For storing row strings

---

4. Justification / Proof of Optimality

- Optimality: Both approaches are straightforward and efficient for printing a hollow square.
- Comparison:
- Nested loop â†' Easy to understand for beginners, prints directly
- String concatenation â†' Slightly more efficient if row strings are reused

---

5. Variants / Follow-Ups

- Hollow rectangle (rows â‰ columns)
- Hollow triangle, hollow diamond
- Filled border patterns with different characters
- Hollow square with diagonal * inside

Q11: Pattern 22 : Number Square with Decreasing Layers

---

1. Input, Output, & Constraints

- **Input:**

```
5
```

- **Output:**

```
5 5 5 5 5 5 5 5 5
5 4 4 4 4 4 4 4 5
5 4 3 3 3 3 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 2 1 2 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 3 3 3 3 4 5
5 4 4 4 4 4 4 4 5
5 5 5 5 5 5 5 5 5
```

Constraints:

- 1 â‰¤ n â‰¤ 26 (English alphabets)

---

2. Examples & Edge Cases

Example 1 (edge case): Input:

```
2
```

Output:

```
2 2 2
2 1 2
2 2 2
```

3. Approaches

Approach 1: Using Distance from Edges

- **Idea:**
  - For a position (i, j) in the square, the value = n - min(min(i, j), min(size-1-i, size-1-j))
  - Here, size = 2*n - 1

Java Code:

```
public static void printPattern22(int n) {
    int size = 2 * n - 1;  // Total rows and columns

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int top = i;
            int left = j;
            int right = size - 1 - j;
            int bottom = size - 1 - i;

            int minDistance = Math.min(Math.min(top, bottom), Math.min(left,
right));
            int value = n - minDistance;

            System.out.print(value + " ");
        }
        System.out.println();  // Move to next row
    }
}
```

Complexity:

- Time: O(nÂ²) â†' Double loop for (2*n-1) x (2*n-1) elements

- Space: O(1) â†’ Only loop variables

---

4. Justification / Proof of Optimality

- Optimality: Each element is computed in O(1) using distance from edges, so the approach is efficient.
- Symmetry: Works for any n and automatically handles center and layers.

---

5. Variants / Follow-Ups

- Use letters instead of numbers
- Print pattern in hollow style (only borders of layers)
- Diagonal or rotated versions of the pattern

2:Diamond Pattern

1. Understand the Problem

- **Read & Identify:** Given an odd integer N, print a diamond of stars * with height = N.
- **Goal:** The pattern should be symmetric vertically and horizontally
- **Paraphrase:** Print the upper pyramid (increasing stars), then the lower pyramid (decreasing stars), forming a diamond.

---

2. Input, Output, & Constraints

- **Input:** odd integer N (height of diamond)
- **Output:** print the diamond pattern with height N

Constraints:

- 1 â‰¤ T â‰¤ 100
- 1 â‰¤ N â‰¤ 199 (must be odd)
- Printing size ~ O(NÂ²), which is optimal since output itself is Î˜(NÂ²).

---

3. Examples & Edge Cases

Example:
Input: 5 Output:

```
  *
 ***
*****
 ***
  *
```

---

4. Approaches

Approach 1:â€" Direct Simulation with Two Loops

Idea: The diamond can be split into two parts:

- Upper pyramid (1 star â†' N stars)
- Lower pyramid (N-2 stars â†' 1 star)

Each row has spaces first, then stars.

Number of spaces = (N - stars)/2.

Pseudocode:

```
for each test case:
    read N
    mid = N // 2

    // upper half including middle row
    for i from 0 to mid:
        stars = 2 * i + 1
        spaces = (N - stars) / 2
        print spaces + stars

    // lower half
    for i from mid-1 downto 0:
        stars = 2 * i + 1
        spaces = (N - stars) / 2
        print spaces + stars
```

Complexity:

- Time: O(NÂ²) (you must print NÂ²/2 characters)
- Space:O(1) (apart from output buffer)

Approach 2:Unified Formula

Idea: Instead of splitting into two loops, compute stars directly by i row index.

- For row i (0-based, total rows = N):
  - If i â‰¤ mid: stars = 2*i + 1
  - Else: stars = 2*(N-i-1) + 1
- Spaces = (N - stars)/2

Pseudocode:

```
for each test case:
    read N
    mid = N // 2
    for i from 0 to N-1:
        if i <= mid:
            stars = 2*i + 1
        else:
            stars = 2*(N-i-1) + 1
        spaces = (N - stars) / 2
        print spaces + stars
```

Complexity:

- Time:O(nÂ²)
- Space: O(1)

---

6. Justification / Proof of Optimality

- You must print O(NÂ²) characters (â‰ˆ NÂ²/2 stars + NÂ²/2 spaces).
- Both approaches accomplish this in O(NÂ²) time and O(1) space.
- Splitting into halves or using a unified formula is equivalent in complexity; the unified formula is cleaner

7. Variants / Follow-Ups

- Diamond with hollow center (* only on border).
- Diamond of numbers instead of stars.
- Diamond aligned to left/right instead of centered.
- Print multiple diamonds side by side.

Q3: Print Number Pattern 3

---

1. Input, Output, & Constraints

- **Input:**

```
5
```

- **Output:**

```
0
1 1
2 3 5
8 13 21 34
55 89 144 233 377
```

Constraints:

- 1 â‰¤ n â‰¤ 20
- Target time complexity: O(nÂ²)
- Target space complexity: O(1) if generating on the fly

---

2. Examples & Edge Cases

Example 1 (Single Row): Input:

```
1
```

Output:

```
0
```

Example 2 (Two Rows): Input:

```
2
```

Output:

```
0
1 1
```

---

3. Approaches

Approach 1: Generate On the Fly (Optimal)

Idea: Keep track of the last two Fibonacci numbers and generate numbers row by row. Print them immediately or store in a list.

Pseudocode:

```
function printFibonacciTriangle(n):
    a = 0, b = 1
    for row = 1 to n:
        for i = 1 to row:
            print a
            c = a + b
            a = b
            b = c
```

Complexity:

- Time: O(nÂ²)
- Space: O(1) (no extra storage needed)

---

4. Variants / Follow-Ups

- Print the triangle in reverse (largest row first).
- Right-align the triangle for better formatting.
- Generate similar patterns for other sequences (Tribonacci, Lucas numbers).
- Store all numbers in a single-line format for API submission or further processing.

Q9: Pattern 18 â€" Alphabet Pyramid Ending with â€˜Eâ€™

---

1. Input, Output, & Constraints

- **Input:**

```
5
```

- **Output:**

```
E
D E
C D E
B C D E
A B C D E
```

Constraints:

- 1 â‰¤ n â‰¤ 26 (English alphabets)

2. Approaches

Approach 1: Using ASCII Values

- **Idea:**
  - 'A' has ASCII value 65.
  - The last letter is 'A' + n - 1.
  - For row i, start printing from (last_letter - i + 1) up to last_letter.

Pseudocode:

```
function printPattern18(n):
    last_char = 65 + n - 1      # ASCII of last letter
    for row in range(1, n+1):
        start_char = last_char - row + 1
        for col in range(start_char, last_char+1):
            print(chr(col), end=" ")
        print()   # new line after each row
```

Complexity:

- Time: O(n^2) â†’ Each row prints up to n letters
- Space: O(1) â†’ Only loop variables

Approach 2: Using String Arithmetic (Optional)

- **Idea:**
  - Pre-generate "ABCDEFGHIJKLMNOPQRSTUVWXYZ" and use slicing.
  - For row i, slice from n-i to n and print letters.

Pseudocode:

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
function printPattern18(n):
    for row in range(1, n+1):
        start_index = n - row
        end_index = n
        for i in range(start_index, end_index):
            print(alphabet[i], end=" ")
        print()
```

Complexity:

- Time: O(n^2)
- Space: O(1)

3. Justification / Proof of Optimality

- Optimality: ASCII method: direct calculation, no extra memory, simple math.
- String slicing: intuitive and readable, especially for beginners.
- Comparison: Both approaches are O(nÂ^2) in time and O(1) in space.
- Use ASCII for efficiency, string for clarity.

4. Variants / Follow-Ups

- Change the ending letter to a custom letter
- Reverse the pattern (start at 'A', go up)
- Diagonal or mirrored pyramid patterns
- Use lowercase letters or other character sets