WEEK 2

WEEK 2A: Linear Probing

Problem Statement

Linear Probing Hash Table – KLU Bachupally Hackathon
KLU Bachupally is conducting a student hackathon.
To manage student data efficiently, the organizers decided to store students' CGPA values in a hash table using the Division Hashing Method with Linear Probing for collision resolution.
You are given:

- The size of the hash table
- A list of CGPA values to be inserted

Your task is to:

1. Insert the CGPA values into the hash table using
    - Hash function:

h(key) = key % table_size

    - Collision handling: Linear Probing
2. Print the final state of the hash table

Input Format

T For each test case: N CGPA1 CGPA2 CGPA3 … CGPAN
Where:
T = number of test cases

- N = size of hash table
- Next line contains N CGPA values (as integers, CGPA × 10 to avoid decimals)

Output Format

For each test case, print the hash table values in order (index 0 to N−1).
If a slot is empty, print -1.
Constraints

- $1 \le T \le 5$
- $1 \le N \le 20$
- $0 \le CGPA \le 100$

TEST CASES:

For example:

Test	Input	Result
1	1 7 14 21 28 35 42	14 21 28 35 42 -1 -1
2	1 5 10 15 20 25 30	10 15 20 25 30
3	1 8 91 82 73 64 55	64 73 82 91 -1 -1 -1 55

CODE:

```
def linear_probing_hash():
    T = int(input())

    for _ in range(T):
        N = int(input())
        keys = list(map(int, input().split()))

        table = [-1] * N

        for key in keys:
            idx = key % N

            # Linear probing
            start = idx
            while table[idx] != -1:
                idx = (idx + 1) % N
                if idx == start:
                    break

            if table[idx] == -1:
                table[idx] = key

        print(*table)

# Run the program
linear_probing_hash()
```

OUTPUT:

	Test	Input	Expected	Got	
✓	1	1 7 14 21 28 35 42	14 21 28 35 42 -1 -1	14 21 28 35 42 -1 -1	✓
✓	2	1 5 10 15 20 25 30	10 15 20 25 30	10 15 20 25 30	✓
✓	3	1 8 91 82 73 64 55	64 73 82 91 -1 -1 -1 55	64 73 82 91 -1 -1 -1 55	✓

EXPLANATION:

- Hash Function: h(key) = key % N
- Linear Probing: If collision occurs, try (index + 1) % N until an empty slot is found.
- Table initialized with -1 for empty slots.

Test Case 1

Input:
1
6
55 61 67 73 79 85
Table size: 6 → N = 6

Step 1: Insert keys

Key	Initial Index (key%6)	Probe Sequence	Inserted Index
55	1	1	1
61	1	1 ✗ →2	2
67	1	1 ✗ →2 ✗ →3	3
73	1	1 ✗ →2 ✗ →3 ✗ →4	4
79	1	1 ✗ →2 ✗ →3 ✗ →4 ✗ →5	5
85	1	1 ✗ →2 ✗ →3 ✗ →4 ✗ →5 ✗ →0	0

Step 2: Final Table (Index → Value)

0	1	2	3	4	5
85	55	61	67	73	79

Output (Horizontal):
85 55 61 67 73 79

Test Case 2 (Heavy Collisions)

Input:

1

5

10 15 20 25 30

Step 1: Compute initial index

Key	key%5	Probe Sequence	Inserted Index
10	0	0	0
15	0	0 ✗ →1	1
20	0	0 ✗ →1 ✗ →2	2
25	0	0 ✗ →1 ✗ →2 ✗ →3	3
30	0	0 ✗ →1 ✗ →2 ✗ →3 ✗ →4	4

Step 2: Final Table

10 15 20 25 30

No empty slots, straightforward linear probing.

Test Case 3 (Partial Table)

Input:

1

8

91 82 73 64 55

Step 1: Compute initial index

Key	key%8	Probe Sequence	Inserted Index
91	3	3	3
82	2	2	2
73	1	1	1
64	0	0	0
55	7	7	7

Step 2: Fill rest with -1

Index	0	1	2	3	4	5	6	7
Value	64	73	82	91	-1	-1	-1	55

Output (Horizontal):

64 73 82 91 -1 -1 -1 55

Test Case 4 (Repeated Collisions)

Input:

1

7

14 21 28 35 42

Step 1: Compute initial index

Key	key%7	Probe Sequence	Inserted Index
14	0	0	0

21	0	0 ✗ →1	1
28	0	0 ✗ →1 ✗ →2	2
35	0	0 ✗ →1 ✗ →2 ✗ →3	3
42	0	0 ✗ →1 ✗ →2 ✗ →3 ✗ →4	4

Step 2: Fill rest with -1

Index	0	1	2	3	4	5	6
Value	14	21	28	35	42	-1	-1

Output (Horizontal):
14 21 28 35 42 -1 -1

Key Notes

1. Linear probing handles collisions sequentially.
2. Empty slots remain -1.
3. Table size determines modulo operation.
4. Probe sequence wraps around using modulo.

WEEK 2B: Quadratic Probing (Open Addressing)

Problem Statement

The **ADS Course Coordinator** at **KLUH Bachupally Campus** is conducting a hackathon to filter and store student records efficiently.
Each student is identified by a **unique roll number**, and due to limited memory, the coordinator decides to store the data using a **hash table**.
To handle collisions, the hashing technique **Quadratic Probing (Open Addressing)** is used.
Your task is to **insert student roll numbers into a hash table** using **Quadratic Probing** and display the final hash table.

Hashing Rules

- **Hash Function**:

$$h(key) = key mod m$$

- **Quadratic Probing Formula**:

$$h_i(key) = (h(key) + i^2) mod m$$

- If the hash table becomes full, stop inserting further elements.

Input Format

T m n r1 r2 r3 ... rn
Where:

- T → Number of test cases
- m → Size of hash table
- n → Number of student roll numbers
- r1...rn → Roll numbers of students

Output Format

For each test case, print the **final hash table**, one element per line.
If a position is empty, print -1.

Constraints

- 1 ≤ T ≤ 5
- 1 ≤ m ≤ 20
- 1 ≤ n ≤ m
- 1 ≤ Roll Number ≤ 10^5

Test cases;

For example:

Test	Input	Result
1	1 7 5 45 52 23 36 58	-1 36 23 45 52 -1 58
2	1 10 6 15 25 35 45 55 65	65 55 -1 -1 45 15 25 -1 -1 35
3	1 11 7 22 33 44 55 66 77 88	22 33 -1 77 44 66 -1 -1 -1 55 -1

CODE:

```
def quadratic_probing(size, keys):
    table = [-1] * size

    for key in keys:
        index = key % size

        if table[index] == -1:
            table[index] = key
        else:
            i = 1
            while i < size:
                new_index = (index + i * i) % size
                if table[new_index] == -1:
                    table[new_index] = key
                    break
                i += 1
    return table

# -------- LMS Input Handling --------
T = int(input())
for _ in range(T):
    size = int(input())
    N = int(input())
    keys = list(map(int, input().split()))

    result = quadratic_probing(size, keys)
    print(*result)
```

Output:

Input	Expected	Got	
1 7 5 45 52 23 36 58	-1 36 23 45 52 -1 58	-1 36 23 45 52 -1 58	✓
1 10 6 15 25 35 45 55 65	65 55 -1 -1 45 15 25 -1 -1 35	65 55 -1 -1 45 15 25 -1 -1 35	✓
1 11 7 22 33 44 55 66 77 88	22 33 -1 77 44 66 -1 -1 -1 55 -1	22 33 -1 77 44 66 -1 -1 -1 55 -1	✓

Explanation :

Test Case 3
Input
1
10
6
15 25 35 45 55 65
Explanation

- Hash table size = **10**
- Hash function:

$$h(key, i) = (key mod 10 + i^2) mod 10$$

- Insert keys using **quadratic probing**
- Empty slots are shown as **-1**

Step-by-Step Insertion

Key	h(key)	Probing Used	Final Index
15	5	No collision	5
25	5	i=1 → 6	6
35	5	i=2 → 9	9
45	5	i=3 → 4	4
55	5	i=4 → 1	1
65	5	i=5 → 0	0

Final Hash Table (Index 0 → 9)
65 55 -1 -1 45 15 25 -1 -1 35

WEEK 2C: Double Hashing – Attendance Filtering System

Problem Statement

During the **ADS course instructors' meeting**, the **course coordinators** decided to digitally filter the list of students whose **attendance is less than 75%**.

To efficiently store and resolve collisions in the attendance database, the instructors use **Double Hashing (Open Addressing)**.

You are given multiple test cases.

For each test case, insert the given student IDs (or roll numbers) into a hash table using **Double Hashing** and display the final hash table.

If a position is empty, print **-1**.

Rules

- **Table Size:** table_size
- **Primary Hash Function:** h1 = key % table_size
- **Secondary Hash Function:** h2 = 1 + (key % (table_size - 1))
- **Probe Formula:**

$$\boldsymbol{index} = (\boldsymbol{h1} + \boldsymbol{i} \cdot \boldsymbol{h2})\%\boldsymbol{table\_size}$$

- i starts at 0 and increments by 1 on collision.
- Table initialized with -1 for empty slots.

Input Format

T TABLE_SIZE N N space-separated integers

Where:

- T = number of test cases
- TABLE_SIZE = size of hash table
- N = number of students with attendance < 75%
- Next line contains N student IDs

Output Format

For each test case:

- Print the hash table **in one line (horizontal output)**
- Use **-1** for empty slots

Constraints

- $1 \leq T \leq 10$
- $5 \leq TABLE\_SIZE \leq 50$
- $1 \leq N \leq TABLE\_SIZE$
- $1 \leq Student\ ID \leq 10^5$

TEST CASES:

For example:

Test	Input	Result
1	1 11 7 22 33 44 55 66 77 88	22 -1 -1 -1 33 44 55 66 77 88 -1
2	1 13 8 18 41 22 44 59 32 31 73	31 44 41 -1 -1 18 32 59 73 22 -1 -1 -1

CODE:

```
def double_hashing(table_size, keys):
    table = [-1] * table_size

    for key in keys:
        h1 = key % table_size
        h2 = 1 + (key % (table_size - 1))
        i = 0
        while i < table_size:
            index = (h1 + i * h2) % table_size
            if table[index] == -1:
                table[index] = key
                break
            else:
                i += 1
    return table

# CodeChef-style input
T = int(input())
for _ in range(T):
    table_size = int(input())
    n = int(input())
    keys = list(map(int, input().split()))
    result = double_hashing(table_size, keys)
    print(*result)
```

OUTPUT:

	Test	Input	Expected	Got	
✓	1	1 11 7 22 33 44 55 66 77 88	22 -1 -1 -1 33 44 55 66 77 88 -1	22 -1 -1 -1 33 44 55 66 77 88 -1	✓
✓	2	1 13 8 18 41 22 44 59 32 31 73	31 44 41 -1 -1 18 32 59 73 22 -1 -1 -1	31 44 41 -1 -1 18 32 59 73 22 -1 -1 -1	✓

Passed all tests! ✓

EXPLANATION:

- **Table Size:** table_size
- **Primary Hash Function:** h1 = key % table_size
- **Secondary Hash Function:** h2 = 1 + (key % (table_size - 1))
- **Probe Formula:**

$$index = (h1 + i \cdot h2)\%table\_size$$

- i starts at 0 and increments by 1 on collision.
- Table initialized with -1 for empty slots.

Test Case 1

Input:

table_size = 11

keys = [22, 33, 44, 55, 66, 77, 88]

Step 1: Compute h1 and h2

Key	h1 = key % 11	h2 = 1 + (key % 10)
22	0	3
33	0	4
44	0	5
55	0	6
66	0	7
77	0	8
88	0	9

Step 2: Insert Sequentially

Key	Index calculation	Inserted Index
22	(0+0*3)%11=0	0
33	(0+0*4)%11=0 ✗ → (0+1*4)%11=4	4

44	(0+05)%11=0 ✗ → (0+15)%11=5	5
55	(0+06)%11=0 ✗ → (0+16)%11=6	6
66	(0+07)%11=0 ✗ → (0+17)%11=7	7
77	(0+08)%11=0 ✗ → (0+18)%11=8	8
88	(0+09)%11=0 ✗ → (0+19)%11=9	9

Step 3: Final Table (Index → Value)

0:22, 1:-1, 2:-1, 3:-1, 4:33, 5:44, 6:55, 7:66, 8:77, 9:88, 10:-1

Output:

22 -1 -1 -1 33 44 55 66 77 88 -1

Test Case 2

Input:

table_size = 10

keys = [15, 25, 35, 45, 55, 65]

Step 1: Compute h1 and h2

Key	h1 = key % 10	h2 = 1 + (key % 9)
15	5	7
25	5	8
35	5	9
45	5	1
55	5	2
65	5	3

Step 2: Insert Sequentially

Key	Index Calculation (i = 0,1,...)	Inserted Index
15	(5+0*7)%10=5	5
25	(5+08)%10=5 ✗ → (5+18)%10=3	3
35	(5+09)%10=5 ✗ → (5+19)%10=4	4
45	(5+01)%10=5 ✗ → (5+11)%10=6	6
55	(5+02)%10=5 ✗ → (5+12)%10=7	7
65	(5+03)%10=5 ✗ → (5+13)%10=8	8

Step 3: Final Table (Index → Value)

0:-1, 1:-1, 2:-1, 3:25, 4:35, 5:15, 6:45, 7:55, 8:65, 9:-1

Output:

-1 -1 -1 25 35 15 45 55 65 -1

Test Case 3

Input:

table_size = 13

keys = [18, 41, 22, 44, 59, 32, 31, 73]

Step 1: Compute h1 and h2

Key	h1 = key % 13	h2 = 1 + (key % 12)
18	5	7
41	2	6
22	9	11
44	5	9
59	7	12
32	6	9
31	5	7
73	8	2

Step 2: Insert Sequentially

Key	Index Calculation	Inserted Index
18	(5+0*7)%13=5	5
41	(2+0*6)%13=2	2
22	(9+0*11)%13=9	9
44	(5+09)%13=5 ✗ → (5+19)%13=14%13=1	1
59	(7+0*12)%13=7	7
32	(6+0*9)%13=6	6
31	(5+07)%13=5 ✗ → (5+17)%13=12	12
73	(8+0*2)%13=8	8

Step 3: Final Table (Index → Value)

0:-1, 1:44, 2:41, 3:-1, 4:-1, 5:18, 6:32, 7:59, 8:73, 9:22, 10:-1, 11:-1, 12:31

Output:

-1 44 41 -1 -1 18 32 59 73 22 -1 -1 31