Bit Happens

Problem: Bit Happens

You are a digital forensics expert tasked with uncovering secrets buried deep within obscure file formats. Each case challenges you to decode hidden patterns, analyze packet captures, and simulate the execution of enigmatic bytecode left behind by an unknown system.

Each test case consists of a packet capture file and a **MysticLang** bytecode file. However, there is only **one shared bitmap file**, which may contain clues relevant to all cases.

The base64 encoded data of the shared bitmap file **common.bmp** is shown below


```
Qk02AwAAAAAADYAAAAoAAAAEAAAABAAAAABABgAAAAAAAADAAATCwAAEwsAAAAAAAAAAAAAzOvv+fL9+Mz02vLa1s/Z2OH8ysz109Hw8u/R1
+nS+uva8Obhzufd2tzW09DT5PPiyObN+fzK9Oj14tbpztvJ7OP/9+DZ8dHM7trn+dPx29zU49fY2fnOztT02ejx2/ji2ur1/tDoy8jz3/bS6P
zQ+OzSyuHY//Xr/sr30NLf/erM49re+NTv8PHs0u/k6dDy4Nfb2tXL4+XZ49PN4tjh3/br4uHQ6ujn1tzp/vHu//Tt+cjz9eXL0O3X/PDS8Pn
2/PDZ6uHr4+vl9/3Vyu3l6/bS+eTly93o6u3W3tjJ68j1/97m3fr9+vPIzNzk08rq/Org+/fwl+T0+d/z1e/e39fV+u/K1+jbzPzlzM7w6tz3
8NPiyPf2+vn90Nbx3OPn2NLN8OTi9PHl2ubL6P/51tDk9tDz3O/n79D0///W88741fji1/jR7+3t1dnuyNPr7OHj7M3x0+ro4P3w5eL45+vu5
d/s9OHW0dPU0N3e4ufQ3uTR7tvk5s3h//D3z/fO+uPd7vbl/v3m7eHj0uT34OnY+Nbs9eH609XX79bLyunNy8zM0N3qy83O5PrS5Pjl14vvS39
v//ePM6vTv6e7Jy8/u5cnW9/Dt/8jj4PDj9/Dt0uj52ebR8P/9z8nc/fLb5f/h2OLI6+br6t3U8O3T9srU1Pf07dbS1Of30/7i2Nz93uvY1fL
s6Ov03c3f0OLM7ePQ2eDd/Pz55uDK+8nczt3l2/Xj/8nl88rxytT/zfjN09bK0N3P0svt7PDo2fXv3tf04+ba4uzjz93Q39/268vO1NTV+Ov2
4OjQ+Prv+N3P0eHl3O35z97Z0ePk687i6ebl0dvm2vrs0NHs9+DI6M/079zW7O3Y4f789ezV4ej77fPi5/XU5sjk2d7k2ena8sjl5Oj599zv3
ufZ5tblytTY0d/b5+vd583zztLr3vfr+u/j2PrZ0vf3yOTy4NLx8ej/69zS5N7w8srV9//k6Or54NLs++ff38/R383i79Dn+OTy2ubS0e7S39
n159Xs
```

Input

A **single base64-encoded string** representing a **JSON object** with the following structure:

JSON Structure (Before Base64 Encoding)

Sample input

```
{
    "n": 1,
    "data": [
        {
            "pcap": "bm90IGEgdmFsaWQgcGNhcA==",
            "myst": "bm90IGEgdmFsaWQgbXlzdGxhbmcgaGV4Y29kZQ==",
            "memory_address": 2
        },
        ...
    ]
}
```

- `n` is the total number of test cases (length of `<data>` array)
- `<memory_address>` is an integer between `0` and `255` (inclusive)
- `<myst>` A base64-encoded string of data pertinent to the "myst" subtask.
- `<pcap>` A base64-encoded string of the PCAP file content relevant to the challenge

For each test case `i` (1-based index):

- `base64` decode the pcap and myst data to arrive at files `i.pcap`, `i.myst`
- Note: You don't need to actually create these files for cracking the challenge. This step is just to help you understand the context for the remainder of the problem.
- If the input is not a valid base64 encoded text, output 3 space separated 0s.

Subtasks

Each case involves the following three parts:

Subtask 1: Bitmap Extraction (Shared File)

- Search the shared bitmap file `common.bmp` for an ASCII-encoded string of the form: `ABC{<number>}` e.g. `ABC{12}`.
- Extract `<number>` (consisting only of digits).
- If the pattern is not found, the number is treated as `0`.
- The base64-encoded contents of `common.bmp` have already been provided above. Since this input is shared across all test cases, the expected output remains the same for each test.
- Techniques such as inspecting the image with hexdump, analyzing differences between color channels or extracting data from a single channel (e.g. red) might be good places to start with.
- For example, the BMP image data (base64 encoded below): `Qk1OAAAAAAAAADYAAAAoAAAABAAAAAIAAAABABgAAAAAABgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQUJDezg4fQAAA AAAAAAAAAAAAAAAAAA` contains the hidden pattern `ABC{88}`. The algorithm to uncover the secret in `common.bmp` may not necessarily be the same.
- Your solution should contain a function `bitmap_extraction` that takes a single string argument (base64-encoded text) and returns the extracted integer. It should work for the base64-encoded content of `common.bmp`. **Submissions with hardcoded answers will be rejected, regardless of the score shown on the HackerRank portal**.

Subtask 2: Packet Pattern Search

- Inspect the contents of the pcap file `i.pcap`.
- There could be multiple data packets exchanged between two IP addresses.
- Depending on the protocol, a single data payload could be fragmented over multiple packets.
- Look for a pattern of the form `ABC{<number>}` e.g. `ABC{123}`.
- Extract `<number>` and compute: `(number % 10007) + 3`
- If the pattern is not found, this part's result is `0` (do not add `3` to it).
- The extracted number could be very large necessitating the use of big-integer math.
- In case there're multiple matches, report the answer for the first hit.
- Note: You can run wireshark/tcpdump to collect network data at home and validate your parsing logic.

Subtask 3: MysticLang Memory Read

- Simulate the MysticLang bytecode file using the spec below.
- Just **before the `halt (FF)` instruction is executed**, read the **value at the specified memory address** `<memory_address>`.
- `<memory_address>` is a decimal integer.
- Interpret the memory value as an unsigned 8-bit integer.
- Return the value at memory_address

Final Output

For each test case `i` (where `1 ≤ i ≤ n`), output a single line containing three space-separated values representing the answers to subtasks `1, 2, and 3`, in that order.

MysticLang CPU Instruction Set

==========

Registers

- **%r0 to %r15** (16 general-purpose registers)
- Registers are **4-bit addressed** (0–15).
- **Encoding Note**:
  Although register IDs are encoded as **full bytes** (0–255), **only values 0–15 are valid**.
  Values above 15 are **invalid**.

Memory

- **256 bytes**
- **Byte-addressed (0–255)**

Flags

- **Z (Zero Flag)**: Set if the result of `sub` is zero.

Stack

- **Stack Pointer**: %r15 (register 15)
- **Stack grows DOWNWARD**:
  - **Push**: Decrement %r15, store value at memory[%r15]
  - **Pop**: Load from memory[%r15], increment %r15

Instruction Encoding Format

Each **instruction** begins with **1 byte opcode**,
followed by **0 or more operand bytes**, depending on the instruction.

- **Opcode**: Always **1 byte** (0x01–0xFF)
- **Operands**: Additional bytes depending on the opcode

Instruction Set

Opcode	Syntax	Operand Bytes	Meaning
0x01	`set imm8, %rX`	imm8, reg4	Set 8-bit immediate into %rX
0x02	`sum %rX, %rY`	reg4, reg4	%rY = %rY + %rX
0x03	`sub %rX, %rY`	reg4, reg4	%rY = %rY - %rX (updates Z if zero)
0x04	`goto addr8`	addr8	Unconditional jump to instruction index addr8 (0 indexed)
0x05	`ifzero addr8`	addr8	Jump if Z flag is set
0x06	`load addr8, %rX`	addr8, reg4	Load memory[addr8] into %rX

`0x07`	`store %rX, addr8`	reg4, addr8	Store %rX into memory[addr8]
`0x08`	`call addr8`	addr8	Push return address, jump to addr8
`0x09`	`ret`	none	Pop return address, jump to it
`0xF`	`halt`	none	Stop execution

Example 1: `set 5, %r0`

```
0x01 0x05 0x00
```

- `0x01`: Opcode `set`
- `0x05`: Immediate value 5
- `0x00`: Destination register %r0

Example 2: `sum %r1, %r2`

```
0x02 0x01 0x02
```

- `0x02`: Opcode `sum`
- `0x01`: Source register %r1
- `0x02`: Destination register %r2
- **Operation**: `%r2 = %r2 + %r1`

- All values are unsigned
- Arithmetic wraps around at 16 bits
- Instruction pointer (IP) moves forward unless jumped

Example

Say `MHgwMSAweDAxIDB4MDEKMHgwMSAweDEwIDB4MDEKMHhGRgo=` represents the base64 encoded version of the hex code for the mystlang program.

Post the decode, you end up with the following program

```
0x01 0x01 0x01
0x01 0x10 0x01
0xFF
```

Both instructions executed before the halt (`FF`) only modify register `r1`, first setting it to `1` and later to `16`, without accessing or modifying memory. Therefore, reading from any valid memory address should return `0`.

Note: The '0x' prefix may be skipped for some of the hexcode input data.

```
01 01 01
01 10 01
FF
```

is the same as the decoded program illustrated above.

Input Format

Input

A **single base64-encoded string** representing a **JSON object** with the following structure:

```
{
    "n": 1,
    "data": [
        {
            "pcap": "bm90IGEgdmFsaWQgcGNhcA==",
            "myst": "bm90IGEgdmFsaWQgbXlzdGxhbmcgaGV4Y29kZQ==",
            "memory_address": 2
        },
        ...
    ]
}
```

- `n` is the total number of test cases (length of `<data>` array)
- `<memory_address>` is an integer between `0` and `255` (inclusive)
- `<myst>` represents the base64 encoded mystlang hexcode data.
- `<pcap>` represents the base64 encoded pcap data

Constraints

`1 ≤ n ≤ 100`

Output Format

For each test case `i` from `1` to `n`, output three space separated values on each line. If the input is not a valid base64-encoded text, output 3 space separated 0s.

Sample Input 0

```
QUJ===$
```

Sample Output 0

```
0 0 0
```

Explanation 0

The input is an invalid base64 encoded text, so, the output has 3 zeros in a single line.