Problem H. Hardware Hashing

Input file:	`hardware.in`
Output file:	`hardware.out`
Time limit:	2.5 seconds
Memory limit:	512 mebibytes

Harry is working in Hlink company that is developing routing hardware for Hoogle. Recently he was asked to write a hashing software for routing chip HH-932.

Let us describe the architecture of HH-932. The chip has 4 megabytes $= 4 \times 2^{20}$ of read only flash memory that can contain the program that the chip will execute and the data. The processor of the chip has 256 registers named `r0` through `r255`, each register contains a 32-bit integer. Before the execution of the program each register except `r0` contains 0, and `r0` contains the input to the program.

The chip has one more special register called the *instruction pointer*. The program is stored in the flash memory and initially the instruction pointer is equal to 0. The execution of the program proceeds as follows. The instruction at the address equal to the instruction pointer is parsed and executed. If the instruction doesn't result in a jump, the instruction pointer is increased by the size of the parsed instruction and thus points to the byte in memory following the recently executed instruction.

Harry has a task to create a very fast hashing function for the given set of n integers $A = \{a_1, a_2, \ldots, a_n\}$ to the set of integers from 0 to $2n - 1$. The evaluation of the function must terminate after at most 30 instructions. Hashing function must not have collisions. Formally, it is required to write a program, that will satisfy the following conditions:

- If the program is executed with one of the integers from the set A, it must terminate after executing at most 30 instructions and return the value from 0 to $2n - 1$.
- For any two different values a_i and a_j from A the returned values must be different.

Harry is not strong in low level programming, so he asks you to help. He has sent you a table with possible instructions of HH-932 chip and their hexadecimal opcodes. He didn't give comments, so you have searched through internet for documentation, didn't find it, but found the following additional comments.

- All registers contain 32-bit integers.
- Addition, subtraction, and bitwise operations do not care whether integers are signed or unsigned.
- Conditional jumps interpret registers as signed integers.
- Result of the multiplication or division is a signed 64-bit integer that is saved into two registers. Register that gets the lower part of the result stores it as unsigned integer.
- Division and remainder use 64-bit integers as a dividend, the lower part is interpreted as unsigned, the upper part as signed. Divisor is signed.
- Division and remainder of signed integers is performed in the same way as in x86 architecture.
- All integers in the chip memory are saved lower byte first (little endian).

You have to create the contents of the flash memory for the chip, so that after it is executed as a program, the required hash function is evaluated and returned.

For any input from the set A your program must not address memory outside of 4M range and must not divide by zero. Instruction pointer must not get out of 4M range either.

Instructions table

Instr.	Size	Opcode	What instruction does
nop	1 byte	`00`	Do nothing
add	4 bytes	`01 r1 r2 r3`	`r3 := r1 + r2`
sub	4 bytes	`02 r1 r2 r3`	`r3 := r1 - r2`
mul	5 bytes	`03 r1 r2 r3 r4`	`(r3, r4) := r1 * r2` r3 gets the lower bits, r4 gets the higher bits of the product
div	6 bytes	`04 r1 r2 r3 r4 r5`	`(r3, r4) := (r1, r5) div r2` r1 contains the lower bits, r5 contains the higher bits of the dividend, r3 gets the lower bits, r4 gets the higher bits of the quotient
mod	5 bytes	`05 r1 r2 r3 r4`	`r3 := (r1, r4) mod r2`, r1 contains the lower bits, r4 contains the higher bits of the dividend
and	4 bytes	`10 r1 r2 r3`	`r3 := r1 and r2`
or	4 bytes	`11 r1 r2 r3`	`r3 := r1 or r2`
xor	4 bytes	`12 r1 r2 r3`	`r3 := r1 xor r2`
neg	2 bytes	`20 r1`	`r1 := -r1`
not	2 bytes	`21 r1`	`r1 := ~r1`
load	3 bytes	`30 r1 r2`	`r1 := memory[r2]` 4 bytes starting from address r2 are copied to register r1, lowest byte first
put	6 bytes	`31 r1 b0 b1 b2 b3`	`r1 := (b0, b1, b2, b3)` here b0 is the lower byte of the number put to the register, b3 — the higher byte
jmp	5 bytes	`40 b0 b1 b2 b3`	Jump to instruction at byte (b0, b1, b2, b3)
jz	6 bytes	`41 r1 b0 b1 b2 b3`	If `r1 == 0` jump to instruction at byte (b0, b1, b2, b3)
jnz	6 bytes	`42 r1 b0 b1 b2 b3`	If `r1 != 0` jump to instruction at byte (b0, b1, b2, b3)
jg	6 bytes	`43 r1 b0 b1 b2 b3`	If `r1 > 0` jump to instruction at byte (b0, b1, b2, b3)
jge	6 bytes	`44 r1 b0 b1 b2 b3`	If `r1 >= 0` jump to instruction at byte (b0, b1, b2, b3)
jl	6 bytes	`45 r1 b0 b1 b2 b3`	If `r1 < 0` jump to instruction at byte (b0, b1, b2, b3)
jle	6 bytes	`46 r1 b0 b1 b2 b3`	If `r1 <= 0` jump to instruction at byte (b0, b1, b2, b3)
ret	1 byte	`ff`	Return from program, the returned value is at `r0`

Input

The first line of the input file contains n ($1 \le n \le 100\,000$). The second line contains distinct n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^9$).

Output

Output the contents of flash memory of the chip. You may output any number of bytes from 1 to 4 194 304. The rest of the memory is filled with zeroes. Each byte must be printed as a hexadecimal two-digit number. Do not print spaces.

Please output only the required prefix of the memory for your program, do not output trailing zeroes, to speed up testing.

Examples

hardware.in	hardware.out
3 2 3 9	3101040000000500010003ff
2 6 10	300000ff00000000000001000000

Explanations

In the first example the program is parsed and executed as follows:

Opcode	Instruction	What happens	What's going on
`31 01 04 00 00 00`	`put r1 4`	`r1 := 4`	`r0` = a_i, `r1 = 4`
`05 00 01 00 03`	`mod (r0, r3) r1 r0`	`r0 := (r0, r3) mod r1`	`r0` = a_i `mod 4`, `r1 = 4`
`ff`	`ret`	`return r0`	$a_i \bmod 4$ is returned

In the second example the program is parsed and executed as follows:

Opcode	Instruction	What happens	What's going on
`30 00 00`	`load r0 r0`	`r0 := memory[r0]`	`r0 = 0` if $a_i = 6$, `r0 = 1` if $a_i = 10$
`ff`	`ret`	`return r0`	0 or 1 is returned

Note that in addition to the program code, the memory in the second example contains some additional data which is used as return values.