Programming Test

In this test, you will have four problems. Please solve these problems by implementing the given interfaces or introducing your system design. Write your code or introductions to the answer sheet. For program code, we tolerate simple syntax errors, and all the code for a problem is seen as written in one file. And when writing your design introductions or code comments, you can freely choose English or Chinese without concern about the penalty. If you decide to implement the given interface while having difficulties writing actual code, you can also provide your ideas for solving the problem (with a 20% penalty) or write pseudocode (with a 10% penalty). If you have any questions, feel free to ask invigilators for help. Note that, the problems are NOT ordered by their difficulty.

Problem 1 - Number Game

Brief

Alice is playing a game with a list of numbers. The list has at least 2 numbers and each number is between 1 and 100. She can do the following operation as many times as she wants: select two different numbers *a* and *b* (*a* > *b*) from the list, replace the bigger one with their difference value, *i.e.* a = a - b. The goal is to make the sum of the list as little as possible.

Interface

C: `int NumberGame(int n, const int *list);`
C++: `int NumberGame(std::vector<int> list);`
Java: `int NumberGame(int[] list);`

Sample

Input 1: [1, 2]
Output 1: 2

Input 2: [2, 4, 6]
Output 2: 6

Input 3: [12, 18]
Output 3: 12

Input 4: [45, 12, 27, 30, 18]
Output 4: 15

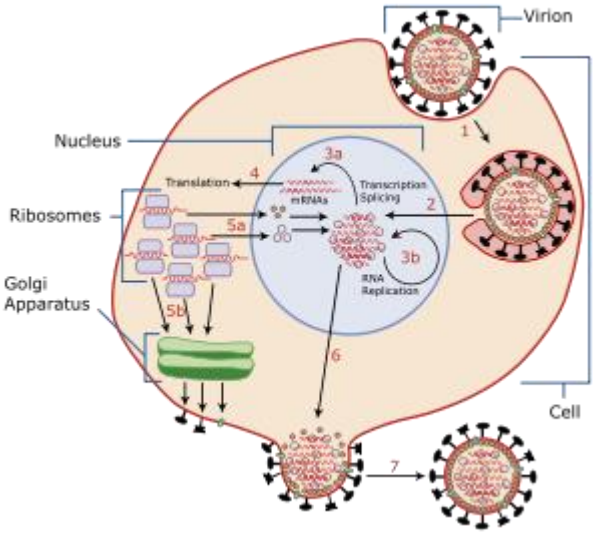
Hints

For input 1, the most optimal operation is to assign x2 = x2 - x1.
For input 2, the most optimal operation is to assign x3 = x3 - x2, then assign x2 = x2 - x1.

Problem 2 - Viral Replication

Brief

Compute how many viruses will be produced after a given time limit.



Viral replication is the formation of biological viruses during the infection process in the target host cells. When a virus infects a cell, it will take *t* timepieces to use itself and the materials in the cell to produce *n* copies, and all these *n* viruses will be released, i.e. one virus becomes *n* viruses after replication. In each cycle, the first timepiece is used to searching for the target cell to infect, and the last timepiece represents the new viruses are released from the infected cell. Between the two timepieces, the viruses are inside the cell's body, and will not be counted. Suppose we have materials to let 4 viruses to replicate, the number of viruses vary as time goes.

Cycle	Cycle 1			Cycle 2		
Timepiece	0	1 ~ *t*-2	*t*-1	*t*	*t*+1 ~ 2*t*-2	2*t*-1
Materials	4	3	3	3	0	0
Viruses	1	0	*n*	*n*	*n*-3	*n*-3+3*n*

Now, we have some materials and one virus in a petri dish. These materials can only supply at most *C* viruses to produce copies. How many viruses will be finally created after a period of *T* timepieces? Tell me the number at the end of the timepiece *T*.

Interfaces

C/C++ (64-bit): `long Replication(long C, long T, int t, int n);`
Java: `long Replication(long C, long T, int t, int n);`

Sample

Input 1: C = 5, T = 24, t = 7, n = 2
Output 1: 6

Input 2: C = 45, T = 28, t = 7, n = 2
Output 2: 16

Input 3: C = 13, T = 24, t = 7, n = 2
Output 3: 2

Hints

The following table presents the relationship between the time (*T*), the materials left (*C*), and the number of viruses (*V*). For simplicity, if there is no change for *C* or *V*, the column will be merged with the column to the left, and *T* for this column is presented as a range.

Case 1:

T	0	1-5	6-7	8-13	14-15	16-20	21-22	23-24
C	5	4	4	2	2	0	0	0
V	1	0	2	0	4	2	6	6

Case 2:

T	0	1-5	6-7	8-13	14-15	16-20	21-22	23-27	28
C	45	44	44	42	42	38	38	30	30
V	1	0	2	0	4	0	8	0	16

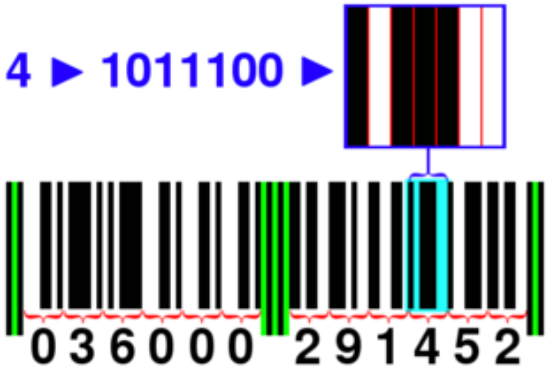
Case 3:

T	0	1-5	6-7	8-13	14-15	16-20	21-22	23-24
C	13	12	12	10	10	6	6	0
V	1	0	2	0	4	0	8	2

Problem 3 - UPC

Brief

Convert the input bitstream of a UPC-A bar code to the corresponding integer array.



A sample UPC-A barcode.

The number-code table of UPC-A barcode.

Number	Left Code	Right Code
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

The Universal Product Code (UPC) is a barcode symbology that is widely used for tracking trade items in stores. Among the UPC barcode standards, the UPC-A barcode is visually represented by strips of bars and spaces that encode the UPC-A 12-digit number. Each digit is represented by a unique pattern of bars and spaces. The bars and spaces have a variable width from 1 to 4 modules wide, representing 1 to 4 one-bits or zero-bits respectively.

A complete UPC-A is 95 modules wide: 7 × 12 = 84 modules for the digits (42 modules each for the left and right sections) combined with 3 + 5 + 3 = 11 modules for the start, middle, and end guard patterns (the green parts). The start and end guard patterns are both 101, and the middle guard pattern is 01010.

The UPC-A's left-hand side digits have odd parity, which means the total width of the black bars is an odd number of modules. On the contrary, the right-hand side digits have even parity. Consequently, a UPC scanner can determine whether it is scanning a symbol from left-to-right or from right-to-left (the symbol is upside-down). The number-code table is provided below.

Your job is to decode a given UPC-A barcode stream to the corresponding number stream. In the input, a leading zero-bit is added to make the input a 12-byte char stream. Read the input char stream from the input parameter, and store the result to the output array.

Interfaces

C: `void UPC(const char *input /*length = 12*/, int *output);`
C++: `std::vector<int> UPC(std::array<char, 12> input);`
Java: `int[] UPC(byte[] input /*length = 12*/);`

Sample

Input stream 1: [83, 36, 222, 163, 98, 189, 81, 36, 116, 229, 155, 101]
Output array 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2]

Input stream 2: [83, 108, 211, 151, 18, 69, 94, 163, 98, 189, 146, 101]
Output array 2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2]

Hints

For sample 1, when we convert the input char array to corresponding binary bytes, the input is `[01010011, 00100100, 11011110, 10100011, 01100010, 10111101, 01010001, 00100100, 01110100, 11100101, 10011011, 01100101]`. And the input can be reformated as: **0** (leading zero-bit) **101** (start section) **0011001** (1L) **0010011** (2L) **0111101** (3L) **0100011** (4L) **0110001** (5L) **0101111** (6L) **01010** (middle section) **1000100** (7R) **1001000** (8R) **1110100** (9R) **1110010** (0R) **1100110** (1R) **1101100** (2R) **101** (end section).

While sample 2 is the reverse case of sample 1. The input byte stream is `[01010011, 01101100, 11010011, 10010111, 00010010, 01000101, 01011110, 10100011, 01100010, 10111101, 10010010, 01100101]`. Since the first 11 bits are `0 101 0011011`, which indicates the first digit has 4 one-bits, the input barcode is upside-down. Then, we need to reverse the input bit stream and reformat it as: **101** (start section) **0011001** (1L) **0010011** (2L) **0111101** (3L) **0100011** (4L) **0110001** (5L) **0101111** (6L) **01010** (middle section) **1000100** (7R) **1001000** (8R) **1110100** (9R) **1110010** (0R) **1100110** (1R) **1101100** (2R) **101** (end section) **0** (leading zero-bit).

For C/C++, when an array of **`char`** type is used as an 8-bit integer array, it can be seen as a byte array of Java **`byte[]`**.

Problem 4 - Interpreter

Brief

You are now asked to implement a simple interpreter for a language called “BF”, please present your system design.

The BF Language Commands

Code	Meaning	C equivalent
`>`	increment the data pointer (to point to the next cell to the right).	`++ptr;`
`<`	decrement the data pointer (to point to the next cell to the left).	`--ptr;`
`+`	increment (increase by one) the byte at the data pointer.	`++*ptr;`
`-`	decrement (decrease by one) the byte at the data pointer.	`--*ptr;`
`.`	output the byte at the data pointer.	`putchar(*ptr);`
`,`	accept one byte of input, storing its value in the byte at the data pointer.	`*ptr=getchar();`
`[`	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.	`while (*ptr) {`
`]`	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.	`}`

The language consists of eight commands, listed below. A BF program is a sequence of these commands. The commands are executed sequentially, with some exceptions: an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command. The program terminates when the instruction pointer moves past the last command. The following table presents all the valid commands in BF.

You are requested to design an interpreter that accepts a sequence of commands as input and executes the program. For input and output of the executed program, you need to read input from the standard input stream and write output to the standard output stream. Besides, for the convenience of dynamically analyzing the input BF program, you need to design an interface for this functionality.

Hints

Dynamic program analysis: Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. Therefore, you need to let the dynamic checkers access the virtual memory and the command being executed / to be executed.