CITY PLANNING

A space station is being built on a planet. The station will employ N persons. They have to live somewhere, and for that, a city will be built around the station. The land surrounding the station is divided into equal-sized square lots, and an apartment building with up to K floors may be built on each lot. Each building shall have exactly one apartment on each floor, and each person shall live in a separate apartment. The lots are assigned coordinates in the form (x, y), where the space station has coordinates (0, 0) and the rest of the lots are numbered as shown below:

…	…	…	…	…
…	(-1, 1)	(0, 1)	(1, 1)	…
…	(-1, 0)	(0, 0)	(1, 0)	…
…	(-1, -1)	(0, -1)	(1, -1)	…
…	…	…	…	…

As the traffic can only flow on streets between the lots, the distance of the lot (x, y) from the space station is $|x| + |y| - 1$ units.

The cost of building a house is equal to the sum of the costs of each floor. It is known that the cost to build a floor depends on the height of the floor, but not on the location of the building.

The houses to be built will last for 30 years. People living in those houses will go to work in the space station, and to transport them to and from the station during these 30 years will cost $T \cdot d$ per person, where d is the distance of the person's building from the station.

We can assume that the planet is big enough and the city to be built occupies a small enough portion of the surface that we do not need to consider the curvature of the planet's surface.

Your task is to write a program that determines the minimal total cost of building the houses and operating the transportation system for the 30 years.

INPUT

The input is read from a text file named **`city.in`**. The first line of the input file contains the integers N ($1 \le N \le 1\,000\,000\,000\,000$), T ($1 \le T \le 500\,000$), and K ($1 \le K \le 20\,000$). The following K lines specify the costs of building the apartments on each floor. The $(i+1)$-th line contains the integer c_i ($1 \le c_i \le 2\,000\,000\,000$), the cost of building an apartment on the i-th floor (assuming all the $i-1$ floors below have already been built). It is known that building a higher floor is always more expensive than building a lower floor: $c_1 < c_2 < \ldots < c_K$.

OUTPUT

The output is written into a text file named **`city.out`**. The first and only line of the output file should contain a single integer — the total cost to build the city and to operate the transportation system for 30 years. It may be assumed that the answer will not exceed $8 \cdot 10^{18}$ (that is, it will fit into a 64-bit signed integer).

EXAMPLE

`city.in`	`city.out`

`17 5 4`
`100`
`107`
`114`
`121`
	`1778`

RLE COMPRESSION

RLE is a simple compression algorithm used to compress sequences containing subsequent repetitions of the same character. By compressing a particular sequence, we obtain its *code*. The idea is to replace repetitions of a given character (like **`aaaaa`**) with a counter saying how many repetitions there are. Namely, we represent it by a triple containing a repetition mark, the repeating character and an integer representing the number of repetitions. For example, **`aaaaa`** can be encoded as **`#a5`** (where `#` represents the repetition mark).

We need to somehow represent the alphabet, the repetition mark, and the counter. Let the alphabet consist of n characters represented by integers from the set $\sum = \{0, 1, \ldots, n-1\}$. The code of a sequence of characters from $\sum$ is also a sequence of characters from $\sum$. At any moment, the repetition mark is represented by a character from $\sum$, denoted by e. Initially e is 0, but it may change during the coding.

The code is interpreted as follows:

- any character a in the code, except the repetition mark, represents itself,
- if the repetition mark e occurs in the code, then the two following characters have special meaning:
  - if e is followed by ek, then it represents $k+1$ repetitions of e,
  - otherwise, if e is followed by $b0$ (where $b \neq e$), then b will be the repetition mark from that point on,
  - otherwise, if e is followed by bk (where $b \neq e$ and $k > 0$), then it represents $k+3$ repetitions of b.

Using the above scheme, we can encode any sequence of characters from $\sum$. For instance, for $n = 4$, the sequence 10022222233333030200000 can be encoded as 10010230320100302101. First character of the code 1 means simply 1. Next 001 encodes 00. Then, 023 represents 222222, 032 represents 33333, and 010 switches the repetition mark to 1. Then 0302 represents itself and finally 101 encodes 0000.

A sequence may be encoded in many ways and code length may vary. Given an already encoded sequence, your task is to find a code with the least number of characters.

Write a program that:

- Reads the size of the alphabet and the code of a sequence.
- Finds the shortest code for that sequence.
- Writes the result.

INPUT

The input is read from a text file named **`rle.in`**. The first line contains one integer n ($2 \le n \le 100\,000$): the size of the alphabet. The second line contains one integer m ($1 \le m \le 2\,000\,000$): the length of the code. The last line contains m integers from the set $\{0, 1, \ldots, n-1\}$ separated by single spaces, representing the code of a sequence.

OUTPUT

The output is written into a text file named **`rle.out`**. The first line should contain one integer m': the least number of characters in a code representing the given sequence. The last line of the output should contain m' integers from the set $\{0, 1, \ldots, n-1\}$ separated by single spaces: the code of the sequence. If there exist several shortest sequences, your program should output any one of them.

EXAMPLES

For the input file **`rle.in`**:

```
4
20
1 0 0 1 0 2 3 0 3 2 0 1 0 0 3 0 2 1 0 1
```

the correct output file **`rle.out`** is as follows:

```
19
1 0 1 0 0 0 1 2 3 1 3 2 0 3 0 2 1 0 1
```

And for the input file **`rle.in`**:

```
14
15
10 10 10 0 10 0 10 10 13 10 10 13 10 10 13
```

the correct output file **`rle.out`** is as follows:

```
9
0 10 13 0 10 13 0 10 10
```

JUMP THE BOARD!

An $n \times n$ game board is populated with integers, one nonnegative integer per square. The goal is to jump along any legitimate path from the upper left corner to the lower right corner of the board. The integer in any one square dictates how large a step away from that location must be. If the step size would advance travel off the game board, then a step in that particular direction is forbidden. All steps must be either to the right or toward the bottom. Note that a 0 is a dead end which prevents any further progress.

Consider the 4×4 board shown in Figure 1, where the solid circle identifies the start position and the dashed circle identifies the target. Figure 2 shows the three legitimate paths from the start to the target, with the irrelevant numbers in each removed.

(2)	3	3	1
1	2	1	3
1	2	3	1
3	1	1	(0)

Figure 1

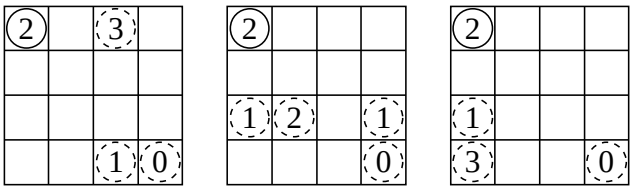

Figure 2

Your task is to write a program that determines the number of legitimate paths from the upper left corner to the lower right corner.

INPUT

The input file **`jump.in`** contains a first line with a single positive integer n, $4 \le n \le 100$, which is the number of rows in this board. This is followed by n rows of data. Each row contains n integers, each one from the range 0…9.

OUTPUT

The output file **`jump.out`** should consist of a single line containing a single integer, which is the number of legitimate paths from the upper left corner to the lower right corner.

EXAMPLE

`jump.in`

```
4
2 3 3 1
1 2 1 3
1 2 3 1
3 1 1 0
```

`jump.out`

```
3
```

GRADING

The number of legitimate paths can be quite big. Only 70% of the score can be achieved using a 64-bit integer variable (`long long int` in C, `Int64` in Pascal). It is guaranteed that all inputs will lead to a number of legitimate paths that can be written with no more than 100 digits.