Arrayser

You and your friend are playing your favourite game, called *Arrayser*. Like all great games, Arrayser involves an array A with N elements, initially set as $A[i] = 1$ for all $0 \leq i \leq N-1$. You and your friend alternate turns, with you going first. On each turn:

1. The player selects an index i with $A[i] = 1$.
2. $A[i]$ is set to 0. Additionally,
   - If $i \neq 0$, then $A[i-1]$ is set to 0.
   - If $i \neq N-1$, then $A[i+1]$ is set to 0.

   Note that $A[i-1]$ and $A[i+1]$ may have already been set to 0 in a previous turn.

The game ends when $A[i] = 0$ for all i.

After playing several games, you realise that your friend is playing completely randomly! Specifically, they always select an index i uniformly at random from all indices where $A[i] = 1$.

Unlike your friend, you absolutely love playing Arrayser and want each game to last as long as possible. In this problem, you will play $T = 5000$ games and your objective is to maximise the mean[1] number of turns that each game lasts for.

Implementation Details

In this task, **do not read from standard input nor write to standard output.** Do not interact with any files. Do not implement a `main` function. Instead, begin your program by including the header file `arrayser.h` (`#include "arrayser.h"`) and interact with it as described below.

Functions

Your solution will play T games in a single test case. You must implement `play_arrayser`, which is called once for each game:

```
void play_arrayser(int N);
```

- `N` is the size of the array.
- This function must call `do_turn` once for each of your turns.
- This function must return when the game ends.
- This function will be called T times in a single test case. We recommend resetting any global variables each time it is called.

From `play_arrayser`, you can make calls to `do_turn`:

```
int do_turn(int i);
```

- i must be an integer with $0 \leq i \leq N-1$ where $A[i] = 1$.
- After you perform your turn, your friend will perform their turn. This function will return the index i that your friend randomly chose. If your turn was the final turn of the game, then this function will instead return -1.

If you violate any of the conditions listed above, your program will be judged as incorrect and you will receive 0% of the points for the test case.

[1]The *mean* is the sum of the values divided by the number of values. For example, the mean of $[3, 5, 4, 8]$ is $\frac{3+5+4+8}{4} = \frac{20}{4} = 5$.

Subtasks and Constraints

This problem has one subtask with one test case (note the unusually-high time limit). The case has $T = 5000$ games all with $N = 300$. Let S be the mean number of turns that were played in each game (this includes your turns and your friend's turns).

- If $S > 144.2$, your score will be 100.
- If $137 \leq S \leq 144.2$, your score will be on a linear scale from 40 to 100. That is, your score will be $40 + 60 \times \frac{S-137}{144.2-137}$.
- If $S < 137$, your score will be $40 \times \left(\frac{S}{137}\right)^{10}$.

A table with some values of S is shown below.

S	100	110	120	130	136	138	140	142	144
Points	1.72	4.45	10.63	23.67	37.17	48.33	65	81.67	98.33

Experimentation

In order to experiment with your code on your own machine, first downloaded the provided files `arrayser.cpp`, `arrayser.h` and `grader.cpp`.

You should modify `arrayser.cpp`, which contains a basic example implementation.

Compile your solution with:

```
g++ -std=c++20 -O2 -Wall arrayser.cpp grader.cpp -o arrayser
```

This will create an executable `arrayser` which you can run with `./arrayser`. If you have trouble compiling, please send a message in the Communication section of the contest website.

The provided grader reads from standard input in the following format:

- The first and only line of input contains N and T.

Sample Grader Input

The following input will run $T = 10$ games, each with $N = 300$:

```
300 10
```

Random Seed

The sample grader has an integer value `RANDOM_SEED` defined on line 9. If this variable is changed, then the grader will make different random choices. The grader used for judging will use a different random seed to the sample grader.

Sample Grader Debug Output

The sample grader has an integer value `DEBUG_LEVEL` defined on line 10, which is set to `1` by default:

- If `DEBUG_LEVEL = 0`, then the sample grader will print the mean number of turns used after all T games conclude.
- If `DEBUG_LEVEL = 1`, then after each game the grader will print the number of turns used in that game. It will also print the mean after all games conclude.
- If `DEBUG_LEVEL = 2`, then after each turn the grader will print information about that turn. In particular, the grader will print the index that was chosen by your code and the grader, as well as the current array A. It will also print the mean after all games conclude.

Flooding

The Bitwise Ranges consist of N mountains, numbered 1 to N from left to right. The ith mountain is 1 kilometre wide and H_i kilometres tall. Atop each mountain is a village.

The government is planning for Q possible scenarios. In the ith scenario, W_i massive downpours of rain will land on the first mountain. Each downpour delivers enough water to submerge a single village in 1 kilometre of water. After the water lands on the first mountain, it will flow using the following rules:

- If the water can flow down, then it will.
- Otherwise, if the water can flow to the right, then it will.
- Finally, if the water cannot flow down or to the right, then it will not move.

If water flows rightwards off mountain N, then its future movements can be ignored.

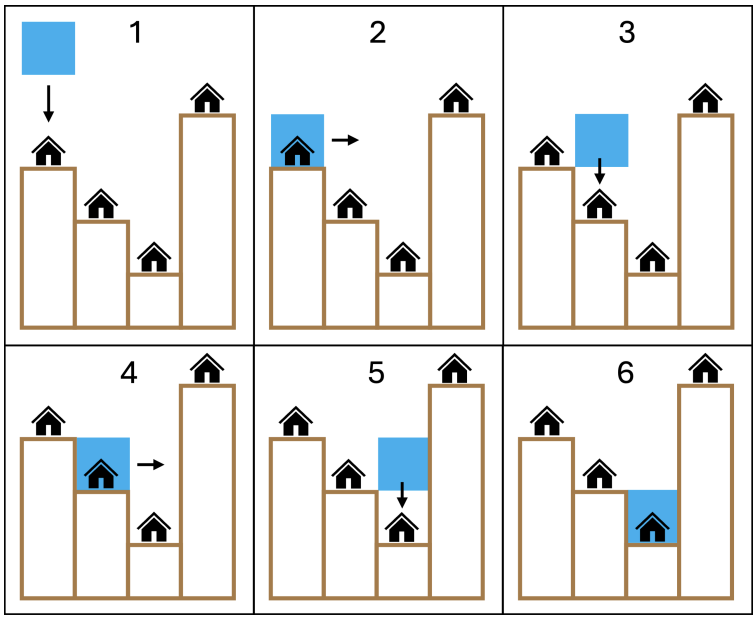

Figure 1: An example downpour. The rain initially lands on mountain 1 and eventually finishes atop mountain 3. Each pane shows one step of the process.

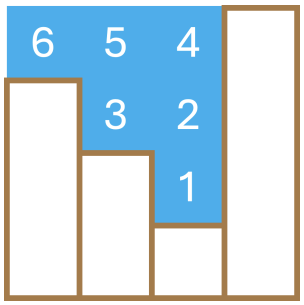

Figure 2: An example scenario with $W_i = 6$ downpours of rain. The final locations of the 6 downpours are labelled.

A village is said to be *flooded* if at least one downpour passes over it, **even if the downpour doesn't finish at this village**.

In each scenario, the government has a budget of K dollars. For one dollar, the government can raise the height of a single mountain by 1 kilometre. By wisely spending their K dollars, the government hopes to minimise the number of villages that are flooded. You have been asked to calculate the minimum number of villages that will be flooded if the government optimally spends its K dollars before the downpours begin.

Note that the Q scenarios are independent. That is, all heights return to their original values before future scenarios begin. However, the budget K is the same in all scenarios.

Subtasks and Constraints

For all subtasks:

- $2 \leq N \leq 200\,000$.
- $1 \leq Q \leq 200\,000$.
- $0 \leq K \leq 1\,000\,000\,000$.
- $1 \leq H_i, W_i \leq 1\,000\,000\,000$ for all i.

Additional constraints for each subtask are given below.

Subtask	Points	Additional constraints
1	17	$N, Q \leq 50$ and $K = 0$.
2	20	$N, Q \leq 50$.
3	18	$N \leq 2000$.
4	21	$K = 0$.
5	24	No additional constraints.

Input

- The first line of input contains the integer N.
- The second line contains N integers $H_1, H_2, \ldots, H_N$.
- The third line contains the integers Q and K.
- The fourth line contains the Q integers $W_1, W_2, \ldots, W_Q$.

Output

Output Q integers on a single line: the answers to the Q scenarios.

Sample Input 1

```
4
3 2 1 4
4 0
1 6 7 1000
```

Sample Output 1

```
3 3 4 4
```

Sample Input 2

```
4
3 2 1 4
3 2
1 7 1000
```

Sample Output 2

```
1 3 4
```

Sample Input 3

```
7
2 1 1 3 1 2 1
4 1
10 9 8 1
```

Sample Output 3

```
7 5 3 2
```

Explanation

The first sample case is shown in Figure 1 and Figure 2 earlier in the statement. The government has a budget of $K = 0$ and so cannot raise any mountains:

- When $W_1 = 1$ and $W_2 = 6$, the first three villages are flooded.
- When $W_3 = 7$ and $W_4 = 1000$, all four villages are flooded.

The second sample case has the same mountains as the first case, but the budget is $K = 2$:

- When $W_1 = 1$, the government can raise the second mountain by 2 kilometres, so that it now has a height of 4 kilometres. Now, only the first village is flooded.
- When $W_2 = 7$, the goverment can raise the first and last mountains by 1 kilometre each. Then, only the first three villages are flooded.
- When $W_3 = 1000$, all four villages will be flooded regardless of how the budget is spent.

The third sample case is shown in Figure 3 below. The government has a budget of $K = 1$:

- When $W_1 = 10$, all seven villages will be flooded regardless of how the budget is spent.
- When $W_2 = 9$, the government can raise the fourth mountain by 1 kilometre, so that only five villages are flooded.
- When $W_3 = 8$, the government can raise the fourth mountain by 1 kilometre, so that only three villages are flooded.
- When $W_4 = 1$, the government can raise the third mountain by 1 kilometre, so that only two villages are flooded.

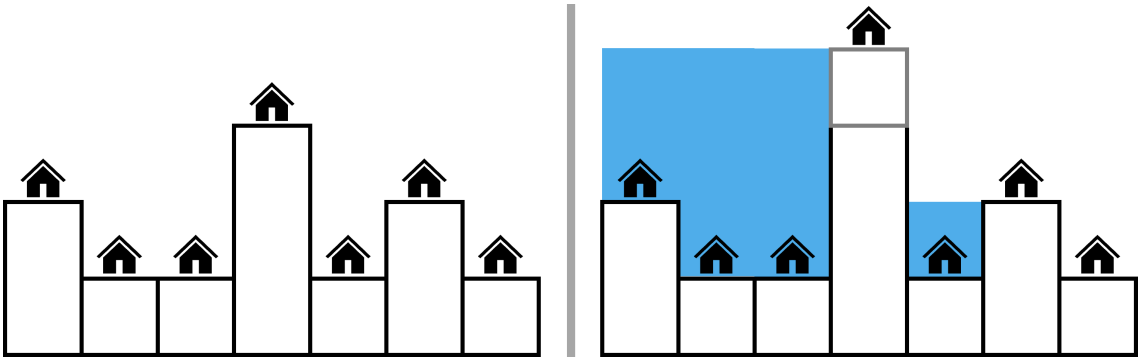

Figure 3: Sample input 3. On the left are the original mountains. On the right is the second scenario with $W_2 = 9$, where the fourth mountain has been raised by 1 kilometre. The first five villages are flooded.

Tree Dash

You have come across a *rooted tree*. This tree consists of N vertices, numbered from 1 to N. One of these vertices, labelled R, serves as the *root*.

Each non-root vertex i has a parent vertex P_i, where $P_i \neq i$. If you start at any vertex and repeatedly move to its parent, then its parent's parent, and so on, you will eventually reach the root vertex. The vertices encountered on this path are called the *ancestors* of i.

Vertex i is a *child* of vertex P_i. The children of a vertex, along with their children, and so on, are called the *descendants* of i.

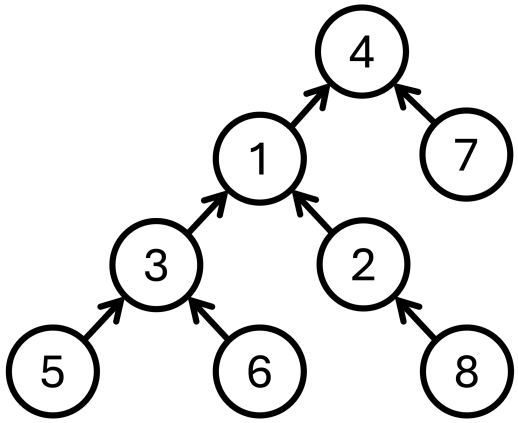

Figure 1: An example tree with $N = 8$. The root is $R = 4$ and each non-root has an arrow to its parent. Vertex 1 has one ancestor (4) and five descendants (2, 3, 5, 6, 8), while vertex 8 has three ancestors (1, 2, 4) and no descendants. The root has no ancestors and $N - 1$ descendants.

If you are at vertex i in this tree, you can travel to any of the following vertices:

- The ancestor of i with the minimum index (if i has at least one ancestor).
- The ancestor of i with the maximum index (if i has at least one ancestor).
- The descendant of i with the minimum index (if i has at least one descendant).
- The descendant of i with the maximum index (if i has at least one descendant).

A vertex u can reach another vertex v if it is possible to travel from u to v, potentially using some intermediate vertices.

Each vertex has a *weight* W_i. Compute the sum of $W_u \times W_v$ across all ordered pairs of vertices (u, v) such that $u \neq v$ and vertex u can reach v.

Subtasks and Constraints

For all subtasks:

- $2 \leq N \leq 500\,000$.
- $1 \leq R \leq N$.
- $1 \leq P_i \leq N$ for all $i \neq R$.
- $1 \leq W_i \leq 1000$ for all i.
- If you start at any vertex and repeatedly move to its parent, then its parent's parent, and so on, you will eventually reach the root vertex.

Additional constraints for each subtask are given below.

Subtask	Points	Additional constraints
1	20	$N \leq 100$ and the tree is a *line* (see $*$ below).
2	15	$N \leq 3000$.
3	25	The tree is a line (see $*$ below).
4	20	The root is 1 and it has only one child, which is N. In other words, $R = 1$, $P_N = 1$, and $P_i \neq 1$ for all $2 \leq i \leq N - 1$.
5	20	No additional constraints.

$*$: a *line* is a tree where every vertex has at most 1 child. Sample input 2 is a line.

Input

- The first line of input contains the two integers N and R.
- The second line contains N integers $P_1, P_2, \ldots, P_N$. Since R does not have a parent, P_R is replaced with a 0.
- The third line contains N integers $W_1, W_2, \ldots, W_N$.

Output

Output a single integer, the sum of $W_u \times W_v$ across all pairs $u \neq v$ where vertex u can reach v.

Note: Your solution may involve integers which are large. Consider using 64-bit integers ('long long' in C++) in your solution.

Sample Input 1

```
8 4
4 1 1 0 3 3 4 2
1 1 1 1 1 1 1 1
```

Sample Output 1

```
30
```

Sample Input 2

```
6 5
6 5 4 2 0 3
1 2 3 4 5 6
```

Sample Output 2

```
228
```

Sample Input 3

```
12 1
0 3 12 3 6 12 8 2 8 12 5 1
70 92 86 72 22 79 20 98 42 56 76 34
```

Sample Output 3

```
228776
```

Explanation

Sample input 1 shown in Figure 1 on the first page:

1. Vertex 1 can reach vertices $2, 4, 8$.
2. Vertex 2 can reach vertices $1, 4, 8$.
3. Vertex 3 can reach vertices $1, 2, 4, 5, 6, 8$.

4. Vertex 4 can reach vertices $1, 2, 8$.
5. Vertex 5 can reach vertices $1, 2, 4, 8$.
6. Vertex 6 can reach vertices $1, 2, 4, 8$.
7. Vertex 7 can reach vertices $1, 2, 4, 8$.
8. Vertex 8 can reach vertices $1, 2, 4$.

Sample input 2 is a line shown in Figure 2:

1. Vertex 1 can reach vertices $2, 5, 6$.
2. Vertex 2 can reach vertices $1, 5, 6$.
3. Vertex 3 can reach vertices $1, 2, 5, 6$.
4. Vertex 4 can reach vertices $1, 2, 5, 6$.
5. Vertex 5 can reach vertices $1, 2, 6$.
6. Vertex 6 can reach vertices $1, 2, 5$.

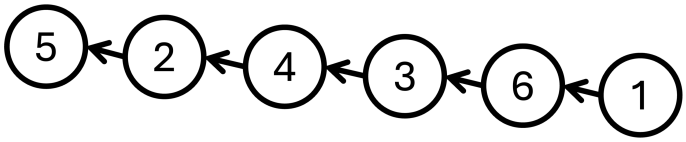

Figure 2: Sample input 2. Each vertex has $W_i = i$.

Sample input 3 is a tree satisfying the constraints of subtask 4, shown in Figure 3:

1. Vertex 1 can reach vertices $2, 7, 9, 11, 12$.
2. Vertex 2 can reach vertices $1, 7, 9, 11, 12$.
3. Vertex 3 can reach vertices $1, 2, 7, 9, 11, 12$.
4. Vertex 4 can reach vertices $1, 2, 7, 9, 11, 12$.
5. Vertex 5 can reach vertices $1, 2, 7, 9, 11, 12$.
6. Vertex 6 can reach vertices $1, 2, 5, 7, 9, 11, 12$.
7. Vertex 7 can reach vertices $1, 2, 9, 11, 12$.
8. Vertex 8 can reach vertices $1, 2, 7, 9, 11, 12$.
9. Vertex 9 can reach vertices $1, 2, 7, 11, 12$.
10. Vertex 10 can reach vertices $1, 2, 7, 9, 11, 12$.
11. Vertex 11 can reach vertices $1, 2, 7, 9, 12$.
12. Vertex 12 can reach vertices $1, 2, 7, 9, 11$.

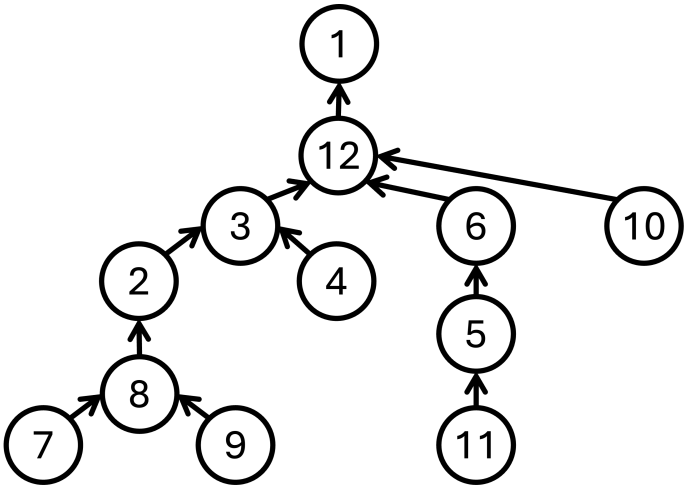

Figure 3: Sample input 3.