Breadcrumbs

Desperate to gain an edge before next year's bread-eating contest, you have sent your spy, Amelia, to infiltrate a rival city. After months of careful observation, Amelia has finally uncovered their secret techniques. Any slip up will immediately reveal her as a spy, and so she must be very careful when sending information back to you.

Direct communication is impossible. Instead, she will transmit a message by walking through the city and leaving behind a trail of breadcrumbs. You must reconstruct the hidden message by observing her path.

The city has N locations, numbered from 1 to N, and M streets that each connect two different locations. You can travel along the streets in either direction, and it is possible to travel between any two locations using the streets.

Amelia's walk is a sequence of locations $V_1, V_2, \ldots, V_L$ such that for every $1 \leq i < L$, the locations V_i and V_{i+1} are connected by a street. The walk must start at **location 1** (so $V_1 = 1$) and may revisit the same location or street multiple times. Using this, you must recover a binary message consisting of **exactly 1 000 bits**.

Your task is to implement both sides of this scheme:

- **Encoding:** Given the map of the city and a sequence of 1 000 bits, produce a walk of any length that starts at location 1.
- **Decoding:** Given the same map and the walk that you produced, recover the 1 000 bits.

You are scored based on the length of the encoding walk, where shorter walks receive higher scores. See the scoring section for details.

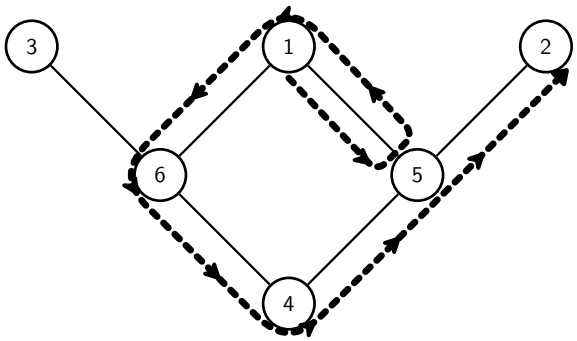

Figure 1: An example city and a walk

Figure 1 shows a city with $N = 6$ locations and streets between $(1, 5)$, $(1, 6)$, $(2, 5)$, $(3, 6)$, $(4, 5)$, and $(4, 6)$.

The walk shown goes through $1 \to 5 \to 1 \to 6 \to 4 \to 5 \to 2$. This is a valid walk of length 7, starting at location 1.

This walk might encode a binary message of length 1 000 (e.g. $[1, 0, 1, 1, 0, 0, \ldots]$). The decoder, given this walk and the city, must output the same 1 000 bits.

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 `crumbs.h` (`#include "crumbs.h"`) and interact with it as described below.

Functions

You must implement the functions `init`, `encode` and `decode`:

```
void init(int N, int M, std::vector<int> A, std::vector<int> B);
```

- N is the number of locations and M is the number of streets.
- Vectors A and B each have size M. For $0 \leq j < M$, street j connects locations `A[j]` and `B[j]`. Locations are numbered from 1 to N. It is possible to travel between any pair of locations using the streets and no two streets connect the same pair of locations.
- This function is called once for each instance of your program. See the Judging section and the Experimentation section for more details.

```
std::vector<int> encode(std::vector<int> bits);
```

- `bits` has size 1 000 and each element is 0 or 1.
- You must return a walk starting at location 1: a `std::vector<int>` of location indices $V_1, \ldots, V_L$ with $V_1 = 1$, where each consecutive pair (V_i, V_{i+1}) is connected by a street. The length of the walk is L, the length of the vector. Note that the vector should start from index 0, so that index 0 has V_1 and index $L - 1$ has V_L.
- This function is called 50 times per test case.
- **Your score for the test case depends on the maximum length of the walk you return. See the scoring section for details.**

```
std::vector<int> decode(std::vector<int> walk);
```

- `walk` is a sequence of locations forming a valid walk starting at location 1 (as produced by your encoder).
- You must return a `std::vector<int>` of 1 000 integers (each 0 or 1) that is the same as the bits that were passed into `encode` to produce this walk.
- This function is called 50 times per test case.

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.

Judging

The judge will start two separate instances of your program: an *encoder* and a *decoder*.

- Firstly, the judge will call `init` once on the encoder.
- Secondly, the judge chooses 50 binary messages and calls `encode` 50 times on the encoder.
- Thirdly, the judge will call `init` once on the decoder. **The exact same parameters used for the first step will be provided to this function.**
- Lastly, the judge will call `decode` 50 times on the decoder, once for each walk produced by the encoder, in an arbitrary order.

The encoder and decoder cannot communicate in any way beyond sending a walk. In particular, any information saved to static or global variables in the encoder is not available in the decoder.

The time spent by your solution will be measured as the sum of the time spent by the two instances.

Subtasks and Constraints

For all subtasks:

- $3 \leq N \leq 1\,000$.
- $2 \leq M \leq 4\,000$.
- $1 \leq A[i] < B[i] \leq N$ for all i. No two streets connect the same pair of locations.
- It is possible to travel between any two locations using a sequence of streets.

There is one test case per subtask. Additional constraints for each subtask are given below.

Subtask	Points	Additional constraints	J (judge)	T (threshold)
1	8	$N = 3$ and the city forms a complete graph.[1]	1,001	1,001
2	8	$N = 5$ and the city forms a complete graph.	501	501
3	8	$N = 4$ and the city forms a complete graph.	632	1,001
4	8	$N = 10$, locations 1 to 5 form a complete graph, and there is a street $(i, i-5)$ for all $6 \leq i \leq 10$. There are no other streets.	481	501
5	8	$N = 3$ and the city forms a line, with streets between locations i and $i+1$ for all i.	1,999	2,001
6	8	$N = 5$ and the city forms a line, with streets between locations i and $i+1$ for all i.	1,263	1,391
7	8	$N = 9$ and the city forms a 3×3 grid.[2]	668	831
8	8	$N = 25$ and the city forms a 5×5 grid.[3]	560	711
9	12	$N = 10$, $M = 15$, and the city layout is randomly generated.[4]	591	821
10	12	$N = 100$, $M = 800$, and the city layout is randomly generated.	247	291
11	12	$N = 1\,000$, $M = 4\,000$, and the city layout is randomly generated.	315	391

For all subtasks, the city layout is available for download from the contest site. They will be in the same format as specified by the sample grader.

Scoring

If your solution produces an incorrect decoding (wrong bits), or violates any of the conditions in the Implementation Details section, your score will be 0.

Otherwise, let W be the maximum length of the walk that your encoder produces over all calls for that test case, and let J be the longest length of the walk the judges' solution can produce for the same test case over all possible encoding messages. Additionally, each subtask has a threshold T (see the table above). Your percentage score for that test case is:

- **100%** if $W \leq J$,
- $\mathbf{25 + \frac{T-W}{T-J} \times 75\%}$ if $J < W \leq T$,
- **10%** if $T < W \leq 10000$,
- **0%** otherwise.

[1] In a complete graph, there is a street between every pair of locations

[2] If we rearrange the city into 3 rows and 3 columns where the locations are numbered 1 to N when going left to right and top to bottom, there is a street between two locations if and only if they are adjacent horizontally or vertically.

[3] Similarly defined as the previous subtask.

[4] All cities that satisfy the given constraints are equally likely to be generated.

There is one test case per subtask. Your score for that subtask is the score of that test case multiplied by the number of points for that subtask.

Experimentation

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

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

Compile your solution with:

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

This will create an executable `crumbs` which you can run with `./crumbs`. 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 line of input contains N and M.
- The next M lines of input describe the streets. The ith such line contains two integers A and B, representing a street between locations A and B.

The grader will first call `init` and will then decide 50 different messages. For each message, it will run `encode` to decide a walk, then immediately call `decode`. It will return an error if the returned message is not the same as the original. If all the messages pass, it will return the length of the longest walk returned by `encode`.

The sample grader chooses the bits randomly. **The judge grader may not do this**. Also note that the sample grader runs `encode` and `decode` from the same instance of your program.

Sample Grader Input & Sample Session

The sample grader is supplied the following input:

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

This corresponds to the diagram on the first page. One possible interaction is as follows:

The grader first calls `init(6, 6, [1, 1, 2, 3, 4, 4], [5, 6, 5, 6, 5, 6])`. This corresponds to the supplied input.

The grader then decides on some message composing of 1 000 bits.

- The grader calls `encode([1, 0, 1, 1, 0, 0, ...])` with these 1 000 bits.
- The student returns `[1, 5, 1, 6, 4, 5, 2]`, a walk of length 7 starting at location 1.
- The grader calls `decode([1, 5, 1, 6, 4, 5, 2])` with that walk.
- The student returns `[1, 0, 1, 1, 0, 0, ...]`, the same 1 000 bits. This is correct.

The sample grader makes 49 more calls to `encode` and `decode` with different messages.

The sample grader prints the length of the longest walk generated by `encode`.

Hedge Maze III

Jacques has trapped you in a maze — a hedge maze!

You know that the maze has N rooms, numbered from 1 to N, and that there are $N-1$ passageways that each connect two different rooms. If you follow the passageways in either direction, it is possible to travel between any pair of rooms. However, Jacques has not told you where these passageways are — your goal is to discover all of them.

To help you escape, Jacques has agreed to answer *queries*. In each query, you provide two distinct rooms A and B. Jacques will tell you the room with the **lowest index** that lies on the unique path from room A to room B, **excluding** rooms A and B themselves. If rooms A and B are directly connected by a passageway (so that no other rooms lie between them), Jacques responds with -1.

You must recover the entire structure of the maze using a limited number of queries. See the scoring section for details.

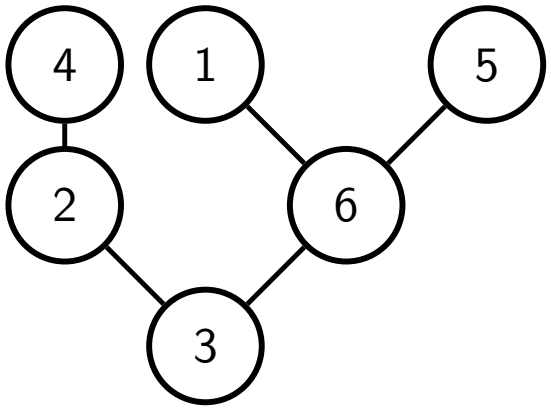

Figure 1: An example maze

Figure 1 shows a maze with $N = 6$ rooms and passageways between $(1, 6)$, $(5, 6)$, $(2, 3)$, $(3, 6)$, and $(2, 4)$. For example:

- If you ask a query with $A = 1$ and $B = 5$, the path from room 1 to room 5 passes through room 6. The only room between them is room 6, so Jacques responds with 6.
- If you ask a query with $A = 1$ and $B = 2$, the path passes through rooms 6 and 3. The lowest index between them is 3, so Jacques responds with 3.
- If you ask a query with $A = 2$ and $B = 4$, rooms 2 and 4 are directly connected. Jacques responds with -1.

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 `maze.h` (`#include "maze.h"`) and interact with it as described below.

Functions

You must implement the function `map_maze`, which is called once in every test case:

```
void map_maze(int N);
```

- `N` is the number of rooms in the maze.
- From this function you can call `query`.
- You must call `report_passageway` exactly $N - 1$ times: once for each passageway in the maze.

From `map_maze`, you can make calls to the following functions:

```
int query(int A, int B);
```

- `A` and `B` must be two distinct rooms, each between 1 and N inclusive.
- This function returns the room with the lowest index on the path from A to B, excluding A and B.
- If A and B are directly connected by a passageway, this function instead returns -1.
- **Your score depends on the number of times this function is called. See the scoring section for details**.

```
void report_passageway(int A, int B);
```

- You should call this function for each passageway between rooms A and B.
- You must call this function exactly $N - 1$ times: once for each passageway in the maze.
- Each passageway is bidirectional; it does not matter whether you report (A, B) or (B, A).

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.

The grader is not adaptive. This means that the answers to all queries are based on a fixed input.

Subtasks and Constraints

For all subtasks:

- $2 \leq N \leq 1\,000$.
- There are $N - 1$ passageways in the maze.
- It is possible to travel between any pair of rooms using the passageways.

Additional constraints for each subtask are given below.

Subtask	Points	Additional constraints
1	25	For every room i, there are less than 10 rooms between room i and room N.
2	20	The maze forms a line: that is, every room is connected to at most 2 other rooms.
3	15	For all $i < N$, room i is connected to at most 2 other rooms. Room N may be connected to any number of other rooms.
4	40	No additional constraints.

Scoring

If your solution fails to find all the passageways, or violates any of the conditions in the Implementation Details section, your score will be 0.

Otherwise, let Q be the number of calls that your solution makes to `query`:

- If $Q \leq 15\,000$, you will receive 100% for the test case.
- If $Q \leq 100\,000$, you will receive 20% for the test case.
- If $Q \leq 1\,000\,000$, you will receive 10% for the test case.
- Otherwise, you will receive 0% for the test case.

Your score for a subtask will be the **minimum** score of all test cases in the subtask, multiplied by the number of points you can score in the subtask.

Experimentation

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

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

Compile your solution with:

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

This will create an executable `maze` which you can run with `./maze`. 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 line of input contains N.
- The next $N - 1$ lines of input describe the passageways. The ith such line contains two integers a and b, representing a passageway between rooms a and b.

Note that the sample grader checks whether the provided passageways form a valid input.

Sample Grader Input & Sample Session

The sample grader is supplied with the following input:

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

This corresponds to the diagram on the first page.

One possible interaction is described below:

Grader	Student	Description
`map_maze(6)`		The grader calls your program.
	`query(1, 5)`	You ask a query for the path from room 1 to room 5.
returns `6`		The path passes through room 6, so Jacques responds with 6.
	`query(1, 2)`	You ask a query for the path from room 1 to room 2.
returns `3`		The path passes through rooms 6 and 3. The lowest index is 3.
	`query(2, 4)`	You ask a query for the path from room 2 to room 4.
returns `-1`		Rooms 2 and 4 are directly connected, so Jacques responds with -1.
	`report_passageway(1, 6)`	You report a passageway between rooms 1 and 6. This is correct.
	...	(You continue until all 5 passageways are reported.)

Perfection

An array of positive integers is called *perfect* if, for every positive integer K, the value K appears either **0 times** or **exactly K times**. An empty array is also considered perfect. For example, the array $[2, 4, 2, 4, 4, 4]$ is perfect because 2 appears exactly 2 times, 4 appears exactly 4 times, and every other value appears 0 times.

You are given an array of N positive integers $V_1, \ldots, V_N$ and may perform the following operations any number of times, in any order:

- **Add** an element (any positive integer) for A coins.
- **Delete** an element for D coins.
- **Modify** an element (change its value to any positive integer) for M coins.

What is the minimum number of coins required to make the array perfect?

Subtasks and Constraints

For all subtasks:

- $1 \leq N \leq 5\,000$.
- $1 \leq V_i \leq 5\,000$ for all i.
- $1 \leq A, D, M \leq 1\,000\,000\,000$.

Additional constraints for each subtask are given below.

Subtask	Points	Additional constraints
1	10	$N = 1$.
2	10	$V_i = V_j$ for all i, j.
3	15	$M > A + D$.
4	25	$M = 1$, $A = 1\,000\,000\,000$, $D = 1\,000\,000\,000$.
5	20	$N \leq 300$, $V_i \leq 300$ for all i.
6	20	No additional constraints.

Input

- The first line contains four integers N, A, D, and M.
- The second line contains N integers $V_1, \ldots, V_N$.

Output

Output a single integer, the minimum number of coins required to make the array perfect.

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

Sample Input 1

```
6 1 2 3
2 4 2 4 4 4
```

Sample Output 1

```
0
```

Sample Input 2

```
1 5 3 4
2
```

Sample Output 2

```
3
```

Sample Input 3

```
3 1 5 2
3 1 1
```

Sample Output 3

```
3
```

Sample Input 4

```
14 10 6 5
3 3 3 3 3 4 4 5 5 5 5 5 5 14
```

Sample Output 4

```
20
```

Sample Input 5

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

Sample Output 5

```
7
```

Explanation

In the first sample case, the array $[2, 4, 2, 4, 4, 4]$ is already perfect, so no operations are required. The answer is 0.

In the second sample case, we can delete the element 2 for 3 coins. This results in the empty array, which is perfect. There is no cheaper option, so the answer is 3.

In the third sample case, we can modify one 1 into a 3 for 2 coins and add another 3 for 1 coin. This results in $[3, 3, 1, 3]$ which is a perfect array. There is no cheaper option, so the answer is 3.

In the fourth sample case, we can modify two 3s into 2s for 10 coins, modify a 5 into a 4 for 5 coins and modify the 14 into a 4 for 5 coins. This results in $[2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 4]$ which is a perfect array. There is no cheaper option, so the answer is 20.

In the fifth sample case, we can modify a 2 into a 1 for 3 coins and delete a 2 for 4 coins. This results in $[2, 2, 1]$ which is a perfect array. There is no cheaper option, so the answer is 7.