Assignment 2
Data Structures and Algorithms

Notes.

- There are 3 programming problems in this assignment. Solutions to all the problems have to be submitted via the automated marker system: https://www.automarker.cs.auckland.ac.nz/student.php
- You can use Java or C++.
  - **Java**: Please use provided `FastScanner` class for reading the input (see template files on canvas).
  - **C++**: If you are using std::cin and std::cout (which you should), make sure your main function starts with the following:

```
ios_base::sync_with_stdio(false);
std::cin.tie(NULL);
std::cout.tie(NULL);
```

- There is a limit of 15 submission attempts for these problems! This is to prevent spamming the Automarker. You should test your program carefully before submitting it to the Automarker.
- Small sample input and its corresponding output are provided on Canvas. Nonetheless, you should test your program with as many different inputs as possible. Feel free to share your developed test cases on Ed Discussion.
- The last submission before the assignment deadline will be the one marked.
- Please do not share your code with other students!

1 Maze [20 marks]

Problem. You are given a table of size $n \times n$, with cells labelled as follows: Cells in the first row are labelled 0 to $n-1$ (left to right), cells in the second row n to $2n-1$, in the third row $2n$ to $3n-1$, and so on. Cells in the last row are labeled $n(n-1)$ to n^2-1. The figure below shows the labelling for $n = 4$.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

You are also given a sequence of m pairs of adjacent cells. Pairs in this sequence are processed one after the other. If the i-th pair is (a_i, b_i), then you should destroy the wall between a_i and b_i **if**, currently, there is no way to get from a_i to b_i. Your goal is to output the table after processing all the pairs.

Input. The first line contains two integers, n and m. Each of the following m lines contains two integers, a_i and b_i. It is guaranteed that $0 \leq a_i, b_i < n^2$, and a_i and b_i are neighbouring cells in the table.

Output. Output the resulting table. For the format of the output, see the example.

Constraints. There are two data sets, `Maze_small` (worth 5 marks) and `Maze_large` (worth 15 marks). In `Maze_small` we guarantee $n \leq 5$, and in `Maze_large` we guarantee $n \leq 300$. In both cases we have $m \leq 2 \cdot n(n-1)$,

Example.

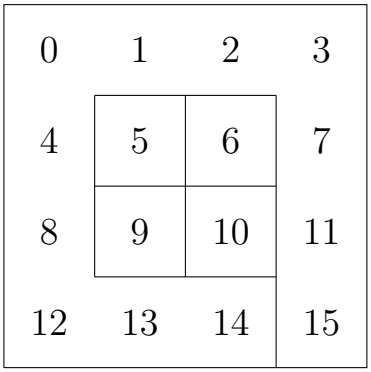
Input:

```
4 17
0 1
1 2
2 3
3 7
7 11
11 15
0 4
4 8
8 12
12 13
13 14
14 15
1 5
5 6
6 10
10 9
9 8
```

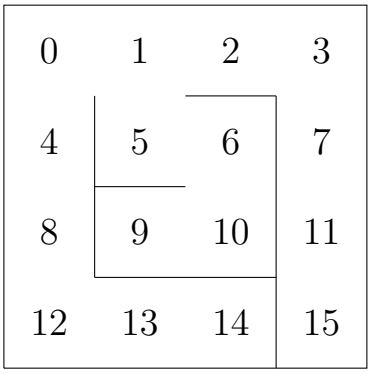
Output:

```
+-+-+-+-+
|       |
+ + +-+ +
| |   | |
+ +-+ + +
| |   | |
+ +-+-+ +
|     | |
+-+-+-+-+
```

Explanation of the example. After processing the first 11 pairs, the table looks as follows:



The next pair is $(14, 15)$, however as there is already a way to get from 14 to 15, we leave the wall between them intact. The next few pairs are $(1, 5)$, $(5, 6)$, $(6, 10)$, and $(10, 9)$. We delete all these walls, which results in the following table:



The last pair is $(9, 8)$. As there is already a way to get from 9 to 8 (namely: 9 – 10 – 6 – 5 – 1 – 0 – 4 – 8), we leave the wall between them intact.

Hint. In the second part of the course you will learn about graphs and algorithms on graphs. This problem can be phrased as a graph problem, however the solution we have in mind does not use **any** knowledge about graphs! Instead, think about **which data structure is suitable for this problem**.

2 Middle [15 marks]

Problem. You start with an empty collection of numbers. You are then given a sequence of m operations, where each operation is either `add(x)`, which asks you to add the number x to the collection, or `middle()`, which asks you to output the $(n+1)/2$-th largest number in your collection (the **middle** element), where n is the current number of elements in the collection. It is guaranteed that `middle()` will only be invoked when n is odd.

Input. The first line contains integer m. Each of the following m lines has one of the two following formats:

- `1 x` (invoke `add(x)`)
- `0` (invoke `middle()`)

Output. For each input line which invokes `middle()`, output the middle element in the collection at that particular point in time.

Constraints. There are two data sets, `Middle_small` (worth 5 marks) and `Middle_large` (worth 10 marks). In `Middle_small` we guarantee $n \leq 1000$, and in `Middle_large` we guarantee $n \leq 100000$. In both data sets, all the numbers in a collection are between 0 and 2^{28}.

Example.

Input:

```
14
1 1
1 1
1 3
0
1 5
1 7
0
1 5
1 1
1 1
1 1
1 10
1 1
0
```

Output:

```
1
3
1
```

Explanation of the example. We are given the following sequence of operations:

1. `add(1)`
2. `add(1)`
3. `add(3)`
4. `middle()` – At this point the collection contains 3 elements $\{1, 1, 3\}$, and the 2-nd largest is 1.
5. `add(5)`
6. `add(7)`
7. `middle()` – At this point the collection contains 5 elements $\{1, 1, 3, 5, 7\}$, and the 3-rd largest is 3.
8. `add(5)`
9. `add(1)`
10. `add(1)`
11. `add(1)`
12. `add(10)`
13. `add(1)`
14. `middle()` – At this point the collection contains 11 elements $\{1, 1, 3, 5, 7, 5, 1, 1, 1, 10, 1\}$, and the 6-th largest is 1.

3 Pattern [15 Marks]

Problem. Given two strings, `pattern` and `text`, count the number of occurrences of `pattern` in `text`.

Input. The first line contains string `pattern`, the second string `text`.

Output. The number of occurrences of `pattern` in `text`.

Constraints. It is guaranteed that `pattern` is of length at most 5000, and `text` of length at most 5000000. All characters in these string are small letters 'a' to 'z'. It is guaranteed that there are less than 200 occurrences of `pattern` in `text`.

Example.

Input:	**Output:**
`abababab` `abababababcababababababab`	`6`

Hint. This is a classic problem with many different solutions. The solution relevant to what we covered in the course so far is called *Rabin-Karp's algorithm*. Here is the basic idea: Instead of checking whether a substring of `text` starting at the position, say, i, is equal to `pattern`, we first calculate its *hash*. If the computed hash of that substring is different than `hash(pattern)`, then we know for sure that they are not equal. Otherwise, we proceed with comparing them character by character.

Clearly, we can calculate `hash(pattern)` once at the beginning and store the value. The key for good performance of this algorithm is the ability to quickly calculate `hash(text[i+1..i+P])` if we already know `hash(text[i..i+P-1])`, where P is the length of `pattern`. If we use the hash function as implemented in Java (discussed in the last lecture), how can we do it without going through all the P characters in `text[i+1..i+P]`?

Note. If you solve this problem by any algorithm other than Rabin-Karp, you will still get your points but you will miss an opportunity to learn a valuable lesson about hash functions!