COMP2119A Introduction to Data Structures and Algorithms
Programming Assignment 1 **Due Date:** 7pm, Oct 17, 2023

Rules: discussion of the problems is permitted, but writing the assignment together is not (i.e. you are not allowed to see the actual solution of another student). We are unable to assist with debugging their source code due to resource limitations. However, we can provide high-level hints and guidance. If you have any questions, you can reach me by email: csxuequan@connect.hku.hk. The PDF may be updated to provide clarifications whenever students have questions.

Course Outcomes

- **[O4]. Implementation**

[O4] In this assignment, you are requested write a program to solve four tasks, **A**, **B**, **C** and **D**, using ADT we learned during class.

Your program will be judged by a computer automatically, please follow the exact format. You should read from standard input and write to standard output. e.g, scanf/print, cin/cout.

Languages.

You must use a C++ compiler. The sample code is written in C++.

Judging.

Please note that your solution is automatically judged by a computer.

Solutions that fail to compile or execute get 0 points.

We have designed 40 test cases, 10 for each task. Each test case is of 2.5 points.

The time limit for each test case is 1 second.

For each test case, you will get 2.5 points if your program passes it otherwise you will get 0.

Self Testing.

You should test your program by yourself using the provided sample input/output file.

The sample input/output is **different** from the ones used to judge your program, and it is designed for you to test your program by your own.

Note that your program should always use standard input/output.

To test your program in Linux or Mac OS X (this is the preferred option):
1. Put your source code "main.cpp" and sample input file "input.txt" in the same directory.
2. Open a terminal and go to that directory.
3. Compile your program by "g++ main.cpp -o main"
4. Run your program using the given input file, "./main < input.txt > myoutput.txt"
5. Compare myoutput.txt with sample output file.
6. Your output needs to be **exactly** the same as the sample output file.

7. You may find the **diff** command useful to help you compare two files. Like "diff -w myOutput.txt sampleOutput.txt". The $-w$ option ignores all blanks (SPACE and TAB characters)

To test your program in Windows (it's fine but maybe it's complicated to install the compiler):
1. Compile your program, and get the executable file, "main.exe"
2. Put sample input file, "input.txt", in the same directory as "main.exe"
3. Open command line and go to the directory that contains "main.exe" and "input.txt"
4. Run "main.exe < input.txt > myoutput.txt"
5. Compare myoutput.txt with sample output file. You may find "fc.exe" tool useful.
6. Your output needs to be **exactly** the same as the sample output file.

If those two options do not work, you can use an online compiler as a last resort.

`https://www.onlinegdb.com/online_c++_compiler`

Note that myoutput.txt file should be **exactly** the same as sample output. Otherwise it will be judged as wrong.

Sketch file. We provide a source file as a sketch for task **B**. You should write your code based on that and you should not modify them.

Submission.

Please submit your source files through moodle.

You should submit four source files(**without** compression), one for each task.

Please name your files in format UniversityNumber-X.cpp for X in {A, B, C, D}, e.g. 1234554321-A.cpp.

Task A

There is a speical kind of binary string. The $(i+1)$th string consists of the reversed string of inverted ith string, a bit "0" and the original ith string. Let S_i denote the ith binary string. S_{i+1} = reverse(invert(S_i))+"0"+S_i. The base case is S_1 = "1".

"reverse" means reverse the string, the first bit becomes the last bit, the second bit becomes the second last bit, and so on. "inverse" inverts the string, 0 changes to 1, and 1 changes to 0.

The following are the first 4 binary strings:

- S_1 = "1"
- S_2 = "001"
- S_3 = "0110001"
- S_4 = "011100100110001"

Now, given two integers $n > 0$ and $k > 0$, $k \leq 2^n - 1$, output the kth bit in S_n.

Input

The input contains two lines.
The first line contains an integer n, $1 \leq n \leq 53$.
The second line contains an integer k, $1 \leq k \leq 1.3 \times 10^9$.

Output

Print the kth bit in S_n.

Example

Sample Input	Sample Output
2 2	0
4 11	1

Task B.

You are given a square matrix A of size $n \times n$. A matrix A is considered nice if it satisfies the following two properties:

- For every $1 \leq i \leq n$ and every $1 \leq j_1 < j_2 \leq n$, we have $A[i][j_1] < A[i][j_2]$.
- For every $1 \leq i_1 < i_2 \leq n$ and every $1 \leq j \leq n$, we have $A[i_1][j] < A[i_2][j]$.

You are given an integer value b. Your task is to find the indices i and j such that $A[i][j] = b$. If b is not present in A, return i = -1, j = -1. You need to optimize the running time to find b.

Input

The input consists of the following parameters separated by line breaks:

- An integer n ($1 \leq n \leq 1000$), representing the size of the square matrix A.
- An integer b, representing the value to be searched in the nice matrix.
- An $n \times n$ matrix A, consisting of space-separated integers, representing the nice matrix.

The matrix A should be presented as n lines, each containing n space-separated integers.

Output

Output the indices i and j ($1 \leq i, j \leq n$) such that $A[i][j] = b$.

Examples.

1. **Input:**

   ```
   3
   7
   1 2 3
   4 6 8
   5 7 9
   ```

   Output:

   ```
   3
   2
   ```

   Explanation: In the given example, the nice matrix A is of size 3×3. The value 7 is present at $A[3][2]$, so the indices i and j are 3 and 2, respectively.

2. **Input:**

```
4
12
1 3 5 7
2 4 6 8
9 11 13 15
10 12 14 16
```

Output:

```
4
2
```

Explanation: In this example, the nice matrix A is of size 4×4. The value 12 is present at $A[4][2]$, so the indices i and j are 4 and 2, respectively.

Task C

Write a program that takes an input string representing a mathematical expression and outputs the result of the expression.

This question consists of three parts, each with increasing difficulty:

Part 1: The program should handle various arithmetic operations, including addition (+) and subtraction (-). It should also consider the order of operations using parentheses and follow standard mathematical rules. There are seven test cases corresponding to this part.

Part 2: In addition to addition (+), subtraction (-), and parentheses, the program should also support multiplication (*) and division (/). There are two test cases corresponding to this part.

For division operations specifically: If both operands of the division operation are integers, the result should be an integer obtained by flooring the division result.

Part 3: In addition to the operations mentioned previously, the program should also support power (^) and factorial (!). There's no ambiguity between the orders of power (^) and factorial (!). There is one test case corresponding to this part. However, please note that this test case is optional and carries a weight of only 2.5 points due to its higher difficulty.

You have two options for submitting your code:

Option 1: You can submit a single file that supports operations for all or some of the parts. Please name the file `UniversityNumber-C.cpp` to indicate it as the main code file.

Option 2: Alternatively, you can submit separate code files, each corresponding to a specific part. Please name the files as follows:

- For Part 1: `UniversityNumber-C-1.cpp`

- For Part 2: `UniversityNumber-C-2.cpp`

- For Part 3: `UniversityNumber-C-3.cpp`

Both options are acceptable for submission. Choose the one that suits your coding structure and preferences best.

Input

The input is a string representing a mathematical expression. The expression is valid and consists of at most 200 characters, excluding spaces. The expression can include the following:

Integers: Positive integers consisting of only one digit.

Arithmetic Operators: addition (+), subtraction (-), multiplication (*), division (/), power (^), factorial (!). Parentheses: Used to specify the order of operations. The input string may contain whitespace characters (spaces) between the elements of the expression, but it should not contain any other non-numeric or non-operator characters.

Output

A single integer value, which is the result of evaluating the mathematical expression provided as input.

Examples.

1. **Input**:
   (4 + 2) - (3 - 1)
   Output:

   4

   Explanation: The input represents the expression "(4 + 2) - (3 - 1)". The addition inside the first parentheses is evaluated first, resulting in (4 + 2) = 6. Similarly, the subtraction inside the second parentheses is evaluated, resulting in (3 - 1) = 2. Finally, the subtraction operation outside the parentheses is performed, resulting in the final output of 6 - 2 = 4.

2. **Input**:
   (5 + 3) * (6 - 2) / 2

   Output:

   16

   Explanation: The input represents the expression "(5 + 3) * (6 - 2) / 2". The addition inside the first parentheses is evaluated first, resulting in (5 + 3) = 8. Similarly, the subtraction inside the second parentheses is evaluated, resulting in (6 - 2) = 4. Then, the multiplication operation is performed, resulting in 8 * 4 = 32. Finally, the division operation is performed, resulting in the final output of 32 / 2 = 16.

3. **Input**:
   2^3 * (4 - 1^2)!

   Output:

   48

   Explanation: The input represents the expression "2^3 * (4 - 1^2)!". The exponentiation is evaluated first, resulting in 2 raised to the power of 3, which is 8. Then, the subtraction inside the parentheses is performed, resulting in (4 - 1^2) = (4 - 1) = 3. Next, the factorial operation is applied to 3, resulting in 3! = 3 * 2 * 1 = 6. Finally, the multiplication is performed, giving the result 8 * 6 = 48.

Task D

A parentheses string s contains just the characters '(' and ')'. The string matches an operation sequence starting from an empty stack: '(' stands for 'push' operation, and ')' stands for 'pop' operation. A parentheses string is valid if open brackets are closed in the correct order, which has equal numbers of '(' and ')', and there is no underflow when executing the matched operation sequence on the stack.

Now, we delete the last character, which is ')', and then insert the deleted ')' into some position. Observe that if there are n symbols in the original string, then the last character has n possible insertion positions (including its original position).

After the insertion, if the new parentheses string satisfies the following properties, we call it a valid insertion. Given a stack of maximum capacity k,

1. there is no underflow when executing the operation sequence. (Underflow happens when we try to pop an item from an empty stack.)
2. there is no overflow when executing the operation sequence. (Overflow happens when we try to push an item to a stack of size equal to its maximum capacity.)

Assume the given stack is empty at first. Output the number of positions where insertion is valid. Your algorithm must run in $O(n)$ time, where n is the length of the parentheses string.

Input

The input contains three lines:
The first line is the the number of characters in the string, which is at most 1024.
The second line is a parentheses string.
The third line is an integer, which stands for the maximum capacity of the given stack.

Output

Output an integer – the number of positions where insertion is valid.

Examples.

1. **Input**:
   6
   ()()()
   2

   Output:

   1

   Explanation: There are 6 possible insertions, and the generated strings are:
   string 0:)()()(, string 1: ())()(, string 2: ()))()(, string 3: ()())(, string 4: ()()))(, string 5: ()()().

Underflow happens: string 0, 1, 2, 3 ,4
Overflow happens:

2. **Input**:
   6
   (()())
   3

   Output:

   5

   Explanation: There are 6 possible insertions, and the generated strings are:
   string 0:)(()(), string 1: ()()(), string 2: (())(), string 3: (())(), string 4: (()()), string 5: (()()).
   Underflow happens: string 0
   Overflow happens:

3. **Input**:
   6
   ((()))
   2

   Output:

   2

   Explanation: There are 6 possible insertions, and the generated strings are:
   string 0:)((()), string 1: ()(()), string 2: (()()), string 3: ((())), string 4: ((())), string 5: ((())).
   Underflow happens: string 0
   Overflow happens: string 3, 4, 5

4. **Input**:
   6
   (())()
   1

   Output:

   0

   Explanation: There are 6 possible insertions, and the generated strings are:
   string 0:)(())(, string 1: ()())(, string 2: (()))(, string 3: (()))(, string 4: (()))(, string 5: (())().

Underflow happens: string 0, 1, 2, 3, 4
Overflow happens: string 5

Note

To compile the files by g++, you can use the command "`g++ main.cpp function.cpp -o main`", which creates an executable file `main` or `main.exe`. For other compilers, you can search online how to compile multiple files.