Chen Jun Hong
chen.junhong@u.nus.edu

CS2040 Lab 4!

AY22/23 Sem 1, Week 6

Administrative matters

- Lab 4 is due on 18 September 2359
- Lab 4 will be manually graded. For Lab Grading details, check Luminus: Files > Lab Materials > Lab Grading Criteria
- Midterm test on Saturday 1st October (Week 7)

Lab 4 - Additional administrative stuff

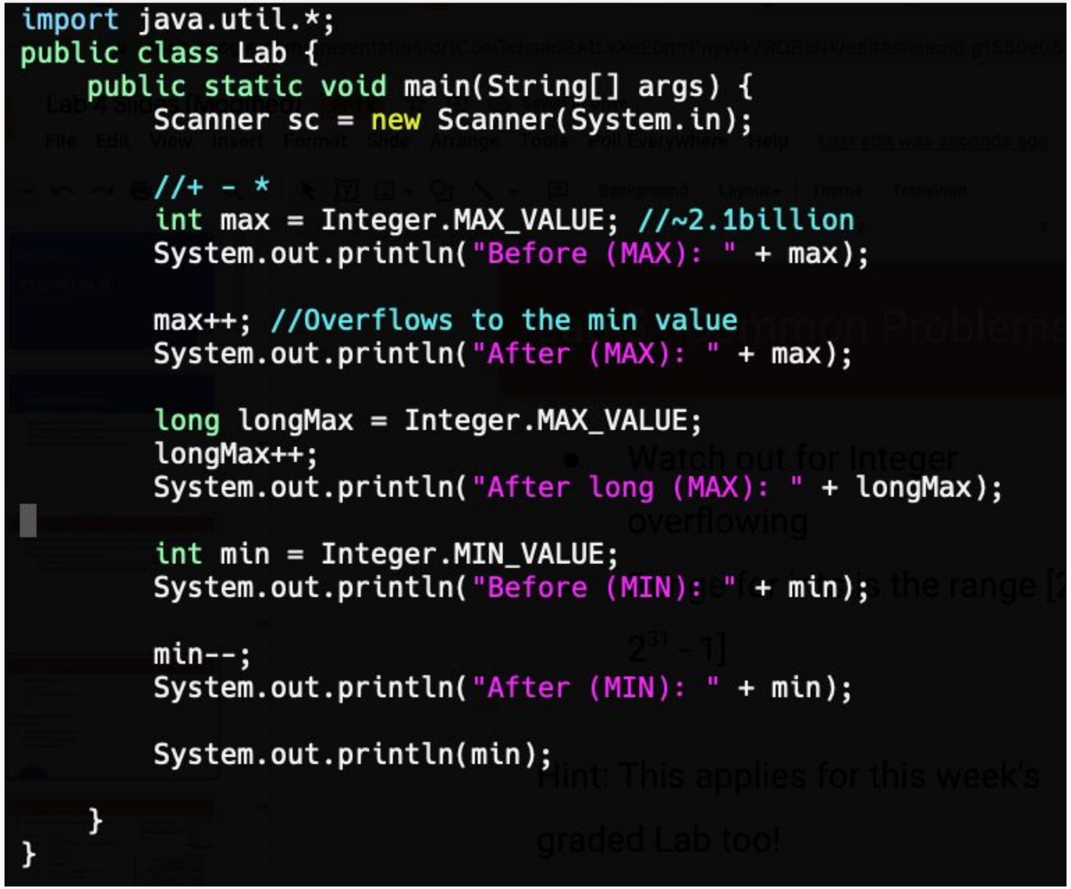
- Some of you have mentioned that the feedback given for your graded lab was not explicit enough. I double checked the grading and have added some additional comments (I also realised that i privated some of the comments). Do take a look again.
    - There are also inline comments written for the code that you all have submitted
    - If the comments are still not clear, please feel free to clarify with me :)

Lab 4 - Common Problems faced

- Watch out for Integer overflowing
- Range for ints is the range [2^{31}, 2^{31} - 1]

Hint: This applies for this week's graded Lab too!

- Think of how exponentially big the calculation can get. Is even using *long* sufficient?



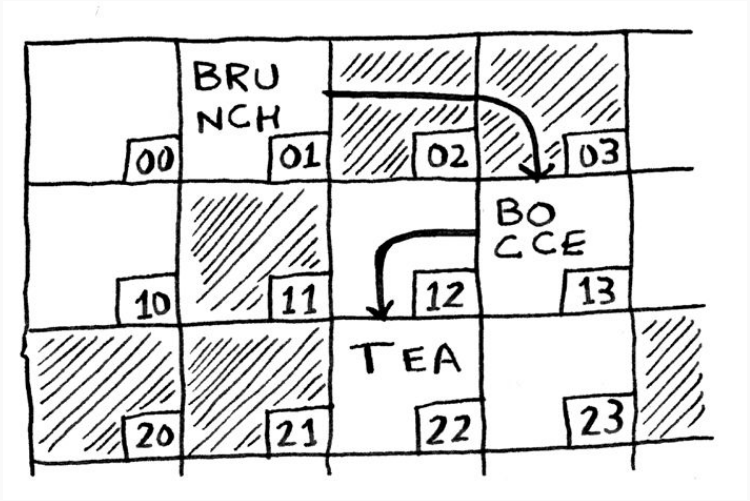
```
import java.util.*;
public class Lab {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        //+ - *
        int max = Integer.MAX_VALUE; //~2.1billion
        System.out.println("Before (MAX): " + max);

        max++; //Overflows to the min value
        System.out.println("After (MAX): " + max);

        long longMax = Integer.MAX_VALUE;
        longMax++;
        System.out.println("After long (MAX): " + longMax);

        int min = Integer.MIN_VALUE;
        System.out.println("Before (MIN): " + min);

        min--;
        System.out.println("After (MIN): " + min);

        System.out.println(min);

    }
}
```

Lab 4 - Common Problems faced cont.

- If you are using LinkedList, be wary of using .get()
- Ex:

```
import java.util.*;
LinkedList<String> list = new LinkedList<>();
list.addLast(“brunch”);
list.addLast(“bocce”);
list.addLast(“tea”);
for (int i = 1; i <= 1000000; i++) {
    list.addLast(“random”);
}
for (int i = 0; i < list.size(); i++) {
    String curr = list.get(i); //O(N), do not
use this!
}
```

Think of memory as a chest of many drawers, each drawer holding one element.



Source: Grokking Algorithms - Arrays and LinkedList

Lab 4 - Recursion Explanation

- Recursion is a form of (algorithm) design. It is used mainly as a problem-solving technique in a "divide-and-conquer" style
  - Ex: Fibonacci recursive function
  - Getting good at recursion is more about tweaking the way you think and approach the question in a more recursive manner
- It is a method where the solution to the problem <u>*depends on* solutions to smaller instances of the SAME problem (sub-problems)</u>
- NO new syntax is needed for recursion (a conceptual understanding of Stacks will be good enough)
- Knowing/Appreciating recursion will be crucial as it is used

Imagine you are entering a cinema. You sit down in a row (which you do not know the row number of) but you realise you had to sit in row 16. How do you figure out where is row 16? Assume that the row numbers are NOT labelled.

Lab 4 - Recursion Cinema Analogy...

Scenario:

Imagine you are entering a cinema. You sit down in a row (which you do not know the row number of) but you realise you had to sit in row 16. How do you figure out where is row 16?
Assume that the row numbers are NOT labelled.

Normal human (iterative) approach:

Get out of your seat.
Walk to the start/back of the cinema (whichever is closer) and start counting to row 16.

Lazy (recursive) approach:

You know that by asking the person in front of you, can determine your own row number. You ask the person in front of you (and the person in front of you asks the person in front of them...) and get your row number. Use the difference between your own row number and 16 and move

Lab 4 - Recursion (Divide and Conquer) - Group Project

Scenario:
You are the team leader in a group of 5 for a report writing group project for one of your modules in NUS.

Your report has four sections (Executive Summary, Challenges, Proposed Solution, Conclusion) and every section can have sub-sections

Assume that the portions have already been written and now you want to perfect it. Each section will be revised by first reading the section again, getting feedback from others and applying the changes needed. How can you achieve this?

Iterative Approach:
1. Start from the smallest sub-section and revise it until the last section

Recursive Approach:
1. Split the sections to be revised to your group members, and let them *keep making revisions* until it is considered complete.

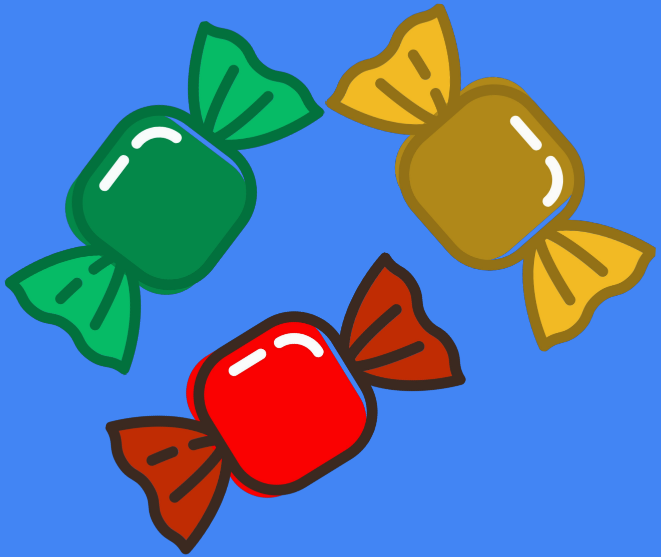
Lab 4 - Recursion (Divide and Conquer) - Group Project (Code Snippet)

```
public Section perfect(Section section) {
     //Base Case
     if (section does not have any subSections)
          read(section);
          getFeedbackFor(section);
          applyChangesFor(section);
          return Section;
     else //Recursive step
          for (every subSection in section) {
               perfect(subSection);
          }
}

perfect(report);
```

```
//The entire report is a
Section (with sections inside
it)
```

Lab 4A

Candyland



Problem Statement

Tom has a sweet tooth, and he frequently visits his local candy shop to buy **N pieces** of his favourite candy.

Tom notices that the scoop used by the shopkeeper is rather small, and he can only **take up a maximum of 4 candies per scoop**.

Tom is very interested in **combinatorics**; hence, Tom wants to count out the **number of distinct ways** to scoop up all N candy pieces, assuming that the storekeeper does not make empty scoops.

Note that a distinct sequence of scoops must include at least one non-empty scoop.

Would you be able to help him?

Problem Statement

Situation:

We have **N** candy pieces

We can take out **1/2/3/4** pieces of candy per scoop

We must repeatedly do this till there is no more candy

How many **possible combinations** are there to clear the candy?

Input:

- Number of candy pieces N
  - 0 <= N <= 12

Output:

- Number of distinct ways

Examples

For **N = 3**

#1 → { 1, 1, 1 }
#2 → { 1, 2 }
#3 → { 2, 1 }
#4 → { 3 }

Thus there are **4** ways

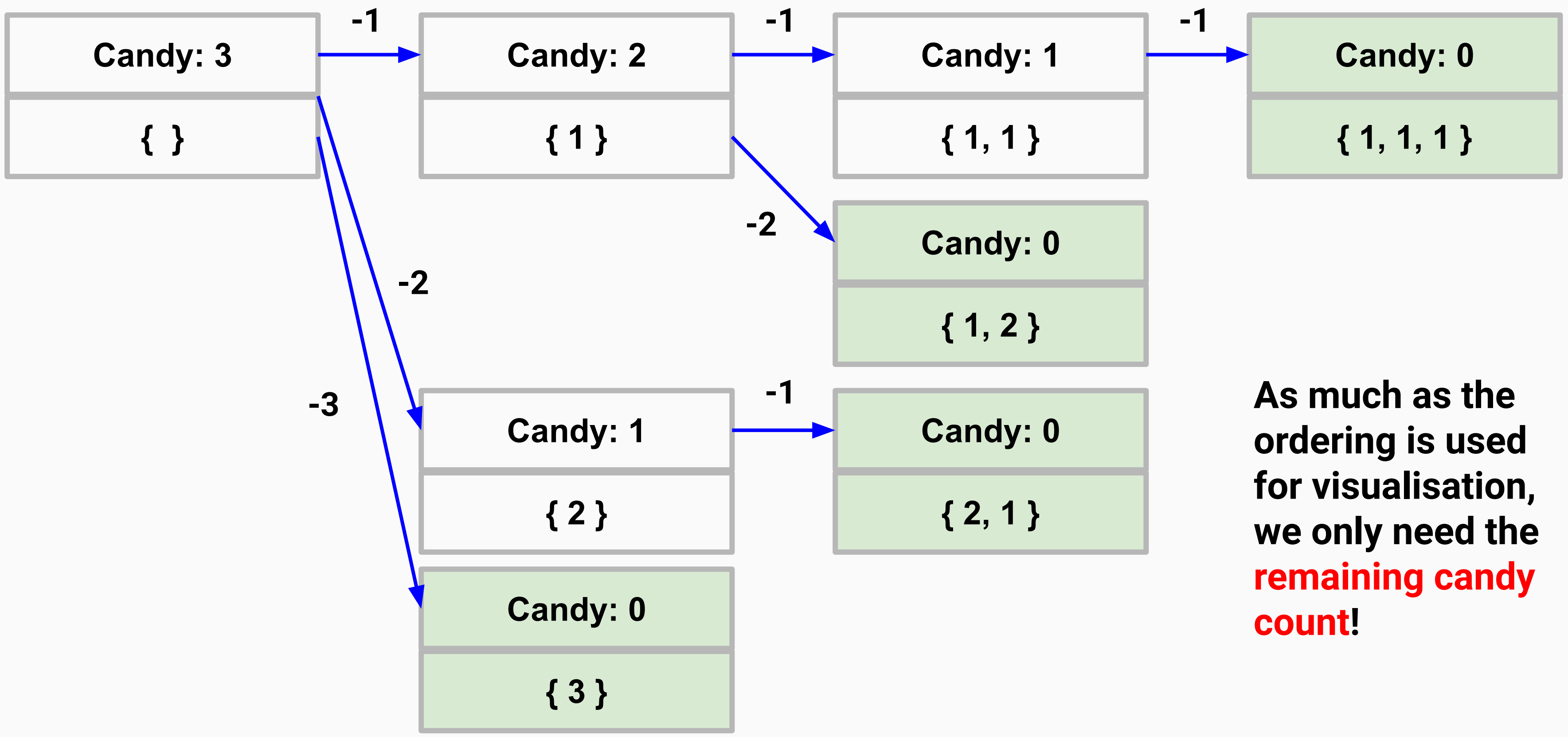
For **N = 4**

#1 → { 1, 1, 1, 1 }
#2 → { 1, 2, 1 }
#3 → { 1, 1, 2 }
#4 → { 1, 3 }
#5 → { 2, 1, 1 }
#6 → { 2, 2 }
#7 → { 3, 1 }
#8 → { 4 }

Thus there are **8** ways

Observations

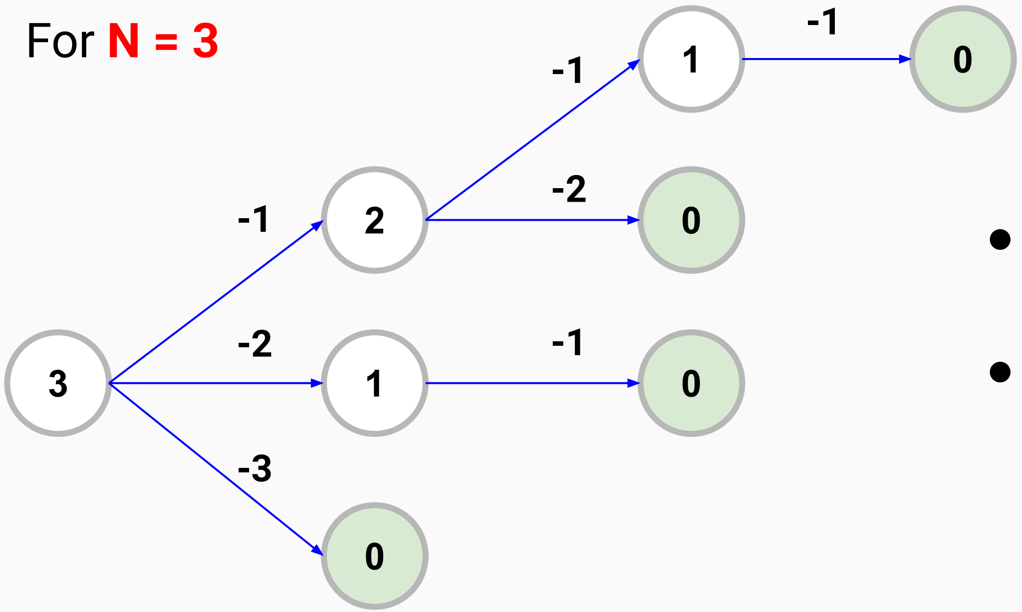
For **N = 3**



As much as the ordering is used for visualisation, we only need the remaining candy count!

Observations

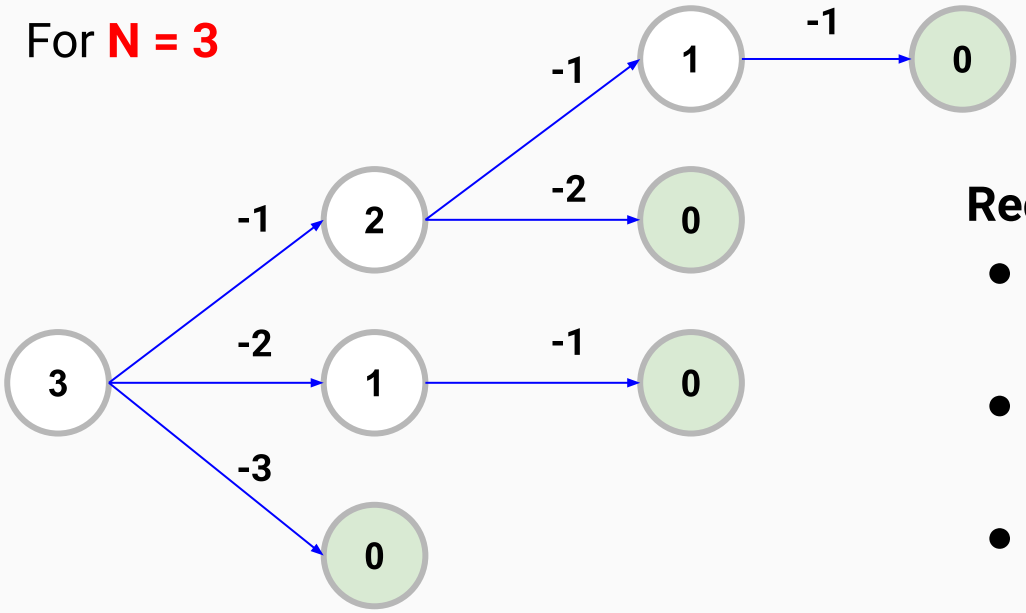
For **N = 3**



- We see there is some **tree structure**
- A big problem is broken down into smaller problems
  - And these smaller problems also face the same situation
- **Recursion?**

Recap

For **N = 3**



Recursion

- A big problem can be broken down to smaller ones
- Smaller problems exhibit same situation
- Generally has:
  - Base case(s)
  - Recursive step

Relook

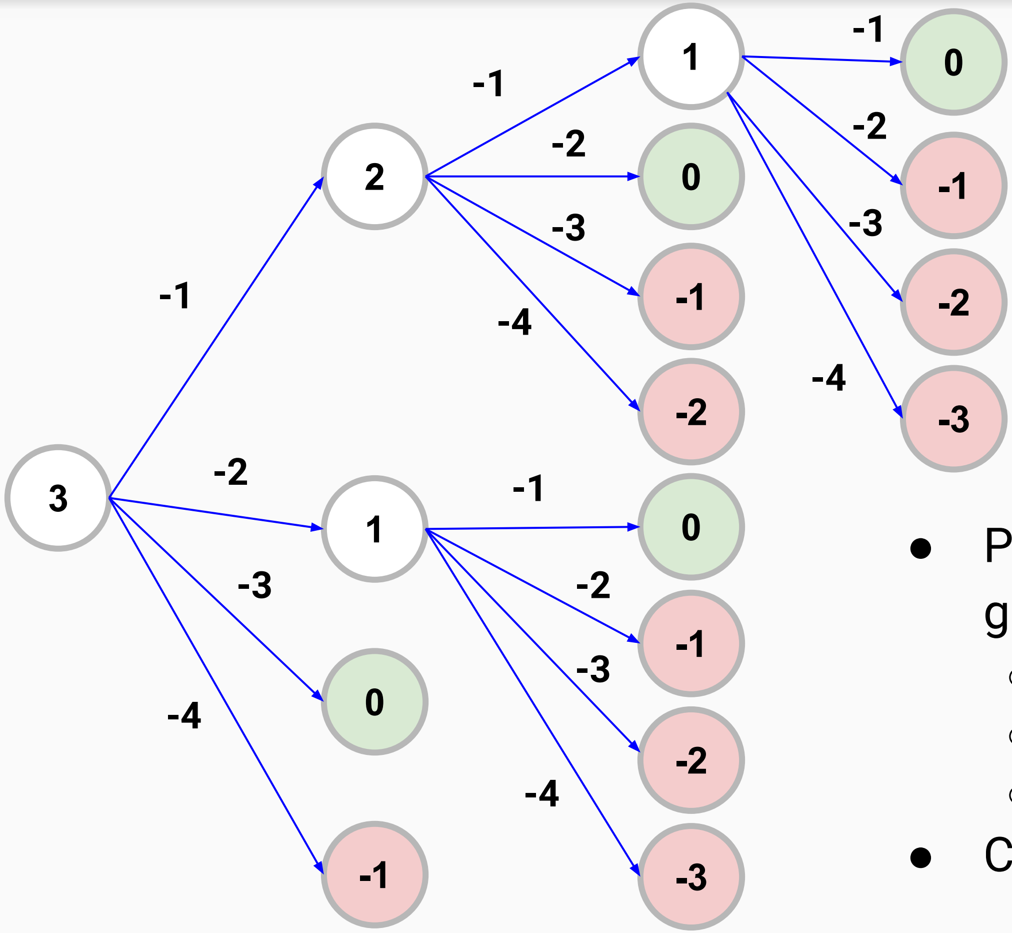
For **N = 3**

$$\begin{tikzcd}
 & & 1 \arrow[r, "-1"] & 0 \\
 & 2 \arrow[ru, "-1"] \arrow[r, "-2"] & 0 & \\
3 \arrow[ru, "-1"] \arrow[r, "-2"] \arrow[rd, "-3"] & 1 \arrow[r, "-1"] & 0 & \\
 & 0 & &
\end{tikzcd}$$

- Notice how we only look at the “good” cases
  - I.e. those that terminate
- Are there any bad ones?
- How are we actually considering if we need to branch out?

Relook again

For **N = 3**



- Problem seems to look more general here
  - End case (good)
  - Bad case
  - ‘Needs more work’ case
- Can we make use of this?

Generalising

What are they?	What do they have in common?	What do we need to do?
3, 1, 2	• Positive • Needs more work	•
0	• Zero • Final case	•
-1, -2, -3	• Negative • Bad case	•

Generalising

What are they?	What do they have in common?	What do we need to do?
3 1 2	● Positive ● Needs more breaking down	● Break it down by subtracting 1/2/3/4 to it ● Group the outcomes of your smaller cases ● How? Summation
0	● Zero ● Final case	● Return 1 ● This means that the combination we have is valid
-1 -2 -3	● Negative ● Bad case	● Return 0 ● This is an invalid combination ● This should not add up to the total number of combinations

Recursive Relation

Let function **combi(n)** be the number of possible combinations with **n** items

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n - 1) + \text{combi}(n - 2) + \text{combi}(n - 3) + \text{combi}(n - 4) & \text{otherwise} \end{cases}$$

Idea in action!

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n-1) + \text{combi}(n-2) + \text{combi}(n-3) + \text{combi}(n-4) & \text{otherwise} \end{cases}$$

For **N = 3**

$$\begin{aligned} \text{combi}(3) &= \text{combi}(2) \\ &\quad + \text{combi}(1) \\ &\quad + \text{combi}(0) \\ &\quad + \text{combi}(-1) \\ &= [\ \text{combi}(1) + \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2)\ ] \\ &\quad + [\ \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2) + \text{combi}(-3)\ ] \\ &\quad + [\ 1\ ] \\ &\quad + [\ 0\ ] \end{aligned}$$

Idea in action!

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n-1) + \text{combi}(n-2) + \text{combi}(n-3) + \text{combi}(n-4) & \text{otherwise} \end{cases}$$

For **N = 3**

$$\begin{aligned} \text{combi}(3) &= [\ \text{combi}(1) + \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2)\ ] \\ &\quad + [\ \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2) + \text{combi}(-3)\ ] \\ &\quad + [\ 1\ ] \\ &\quad + [\ 0\ ] \\ &= \{\ [\ \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2) + \text{combi}(-3)\ ] + 1 + 0 + 0\ \} \\ &\quad + [\ 1 + 0 + 0 + 0\ ] \\ &\quad + 1 \\ &\quad + 0 \end{aligned}$$

Idea in action!

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n - 1) + \text{combi}(n - 2) + \text{combi}(n - 3) + \text{combi}(n - 4) & \text{otherwise} \end{cases}$$

For **N = 3**

$$\begin{aligned} \text{combi}(3) = \{ & [\ \text{combi}(0) + \text{combi}(-1) + \text{combi}(-2) + \text{combi}(-3)\ ] + 1 + 0 + 0 \} \\ & + [\ 1 + 0 + 0 + 0\ ] \\ & + 1 \\ & + 0 \end{aligned}$$

$$\left. \begin{aligned} = \{ & [\ 1 + 0 + 0 + 0\ ] + 1 + 0 + 0 \} \\ & + [\ 1 + 0 + 0 + 0\ ] \\ & + 1 \\ & + 0 \end{aligned} \right\} = 4$$

We have our answer!

Comments

Let function **combi(n)** be the number of possible combinations with **n** items

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n-1) + \text{combi}(n-2) + \text{combi}(n-3) + \text{combi}(n-4) & \text{otherwise} \end{cases}$$

- Notice when we expand out, it can get **very messy**
- This is where we use the notion of **wishful thinking**
  - Observe relations
  - Prove (to some extent) that it works at specific examples
  - Code and hope it works :)
  - If not, rinse and repeat

Analysis

Let function **combi(n)** be the number of possible combinations with **n** items

$$\mathbf{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \mathbf{combi}(n-1) + \mathbf{combi}(n-2) + \mathbf{combi}(n-3) + \mathbf{combi}(n-4) & \text{otherwise} \end{cases}$$

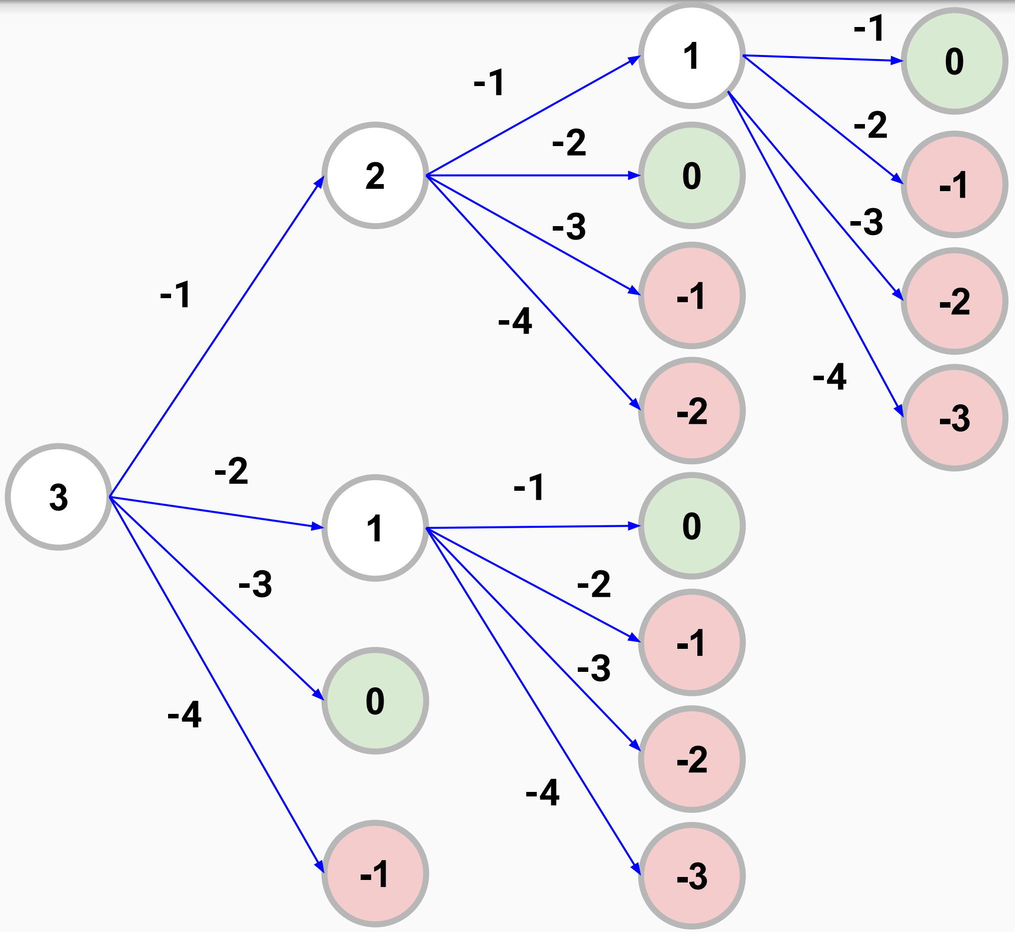
With our recursive relation we have done 2 things:

- Created our <u>code template</u>
- Determined a way to analyse our <u>runtime</u>

Analysis

For **recursive functions** (revisit Tutorial 1):

- Simplify function
  - Determine work done locally
  - Determine subcalls
- Obtain recurrence relation
- Draw tree
  - Optional if can see trend
- Determine height and work done per level
- Observe pattern
- Perform summation



Analysis

$$\text{combi}(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ \text{combi}(n-1) + \text{combi}(n-2) + \text{combi}(n-3) + \text{combi}(n-4) & \text{otherwise} \end{cases}$$

$$< \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ 4 \text{ x } \text{combi}(n-1) & \text{otherwise} \end{cases}$$

Can get very messy when determining work done per level

- Let us just attempt to obtain a simplified upper bound

Analysis

$$\text{combi}(n) < \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ 4 \text{ x combi}(n - 1) & \text{otherwise} \end{cases}$$

We shall do some rough estimation from here

- Each node will branch out to **4 children**
    - This means every time we go down one level we will have **x 4 nodes**
- As each level we decrease our n by 1, our **height h = n**
- Locally in a single method call, O(1) time complexity
    - If-else and summation steps are done
- Thus we have $1 + 4 + 16 + 64 + \ldots + 4^n \approx$ **$O(4^n)$**

Analysis

$$\text{combi}(n) < \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n == 0 \\ 4 \times \text{combi}(n-1) & \text{otherwise} \end{cases}$$

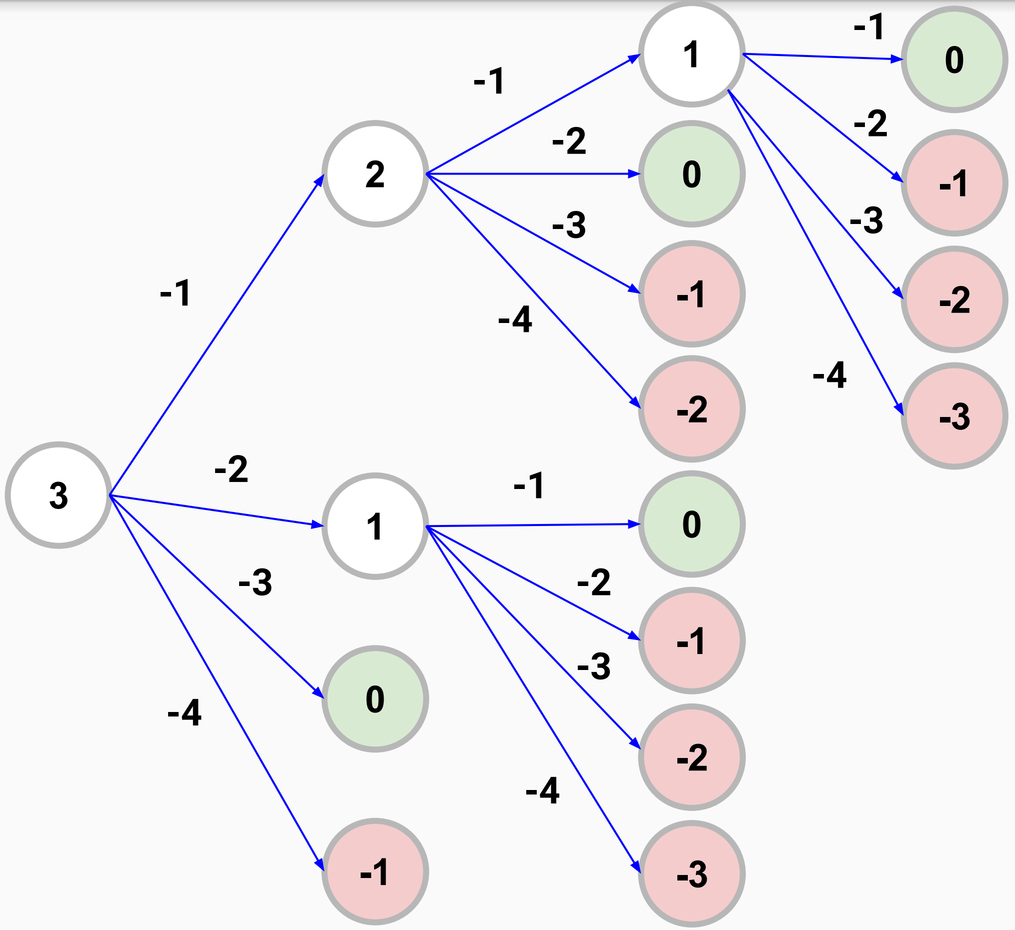
… we have $1 + 4 + 16 + 64 + \ldots + 4^n \approx$ **$O(4^n)$**

- Moral of the story?
  - Recursion can be “easy” to implement
  - But might not be the most efficient method
- Trivia
  - Recursion is the code version of Mathematical Induction !!!

Remarks

Are there any optimisations possible?

- Notice how there are **repeated computations**
  - E.g. there are 2 points where n = 1 exists, and it still breaks down
  - What if n is super large? Alot or repeated values and branches
- Maybe we can sort of **tabulate** our results so we can refer back?



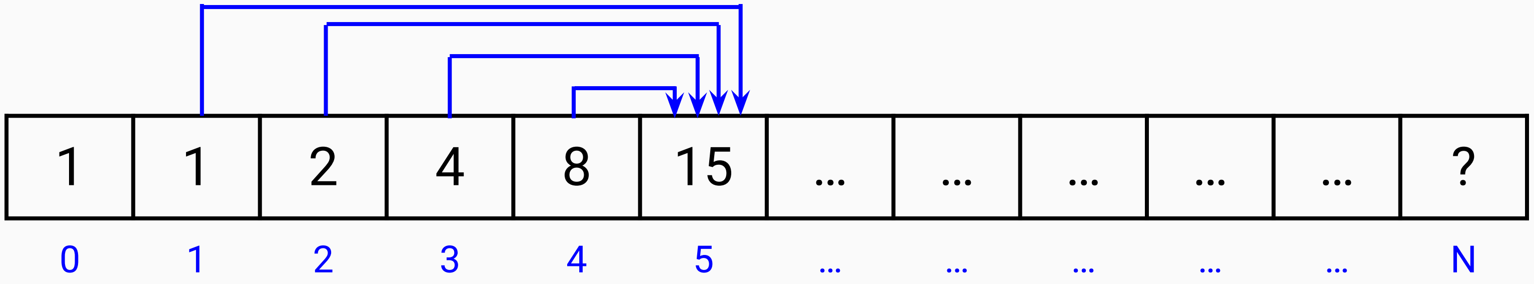
Remarks

Are there any optimisations possible?

- Notice how there are **repeated computations**
- Maybe we can sort of **tabulate** our results so we can refer back?
    - Run time = n x 4*O(1) = **O(n)**
    - Space = O(n)
        - Can we make it O(1)?

<u>Observation</u>

- Given a value n
    - We sum up cases for **n-1, n-2, n-3, n-4**
- Thus how about we fill up an array of **N + 1** items
    - Then for an idx i > 3, add the previous **4 items**



When poll is active, respond at **pollev.com/chenjunhong427**
Text **CHENJUNHONG427** to **+65 8241 0042** once to join

What should be in every recursive function?

For Loop
Base Case(s)
ArrayList
Recursive Case(s)
Divide and Conquer

Powered by Poll Everywhere

Lab 4B

Constellations

When poll is active, respond at **pollev.com/chenjunhong427**

Text **CHENJUNHONG427** to **+65 8241 0042** once to join

There are 8 people in the running for positions in a Student Club. 3 will be selected to become Treasurers, 2 will be selected to become Secretaries and 1 will be appointed President. How many possible ways are there to pick the appointed members?

20160

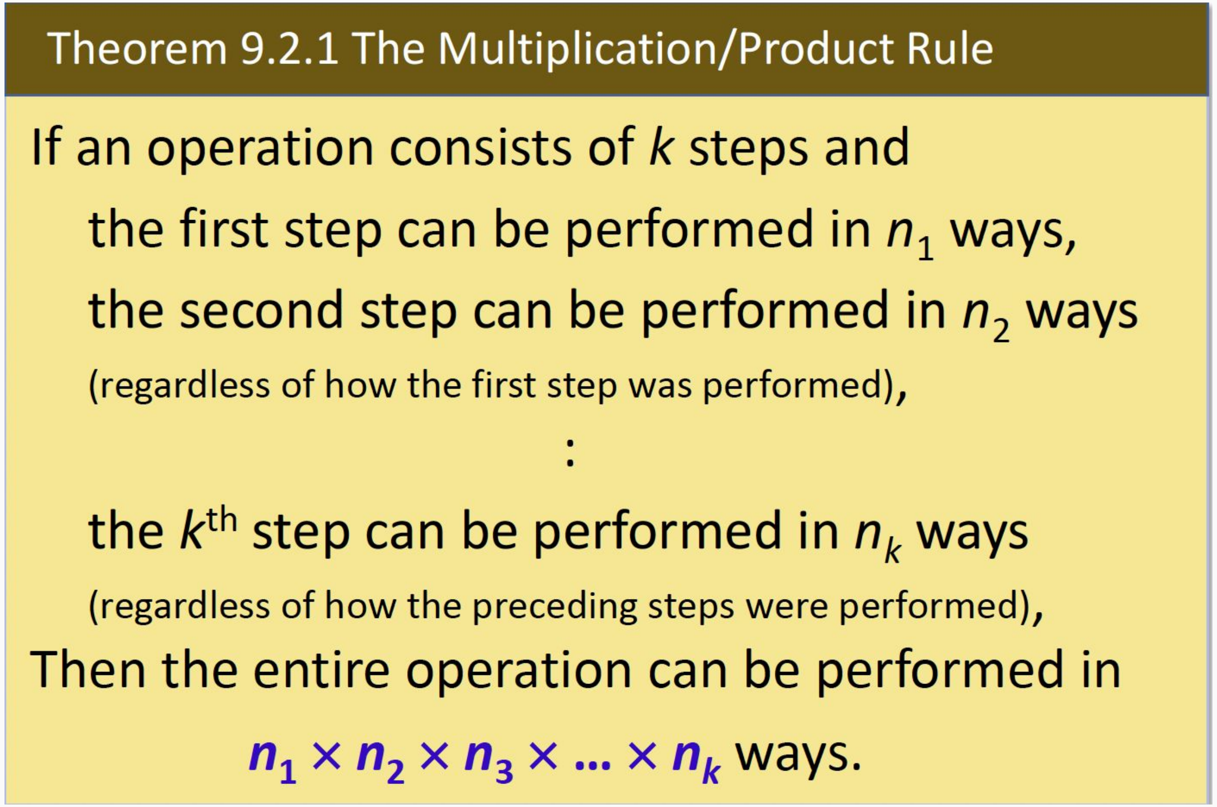
1680

12544

2048

69

Lab 4 Extra - Multiplication Principle



Theorem 9.2.1 The Multiplication/Product Rule

If an operation consists of k steps and

the first step can be performed in n_1 ways,

the second step can be performed in n_2 ways

(regardless of how the first step was performed),

:

the k^{th} step can be performed in n_k ways

(regardless of how the preceding steps were performed),

Then the entire operation can be performed in

$n_1 \times n_2 \times n_3 \times \ldots \times n_k$ ways.

Source: CS1231 Counting Lecture Notes

Problem Statement

Abby the Astronaut wants to create **new constellations**. Given a star map with **n different stars**, he wants to **find the number of different constellations** he can make to the star map.

He can make a **constellation** from the star map by **grouping up x different stars together**, where **a <= x <= b**.

He needs to **fill the star map** with different constellations **until no more constellations are possible** to be made for it to count as a valid configuration.

Every star can only be part of at most 1 constellation.

Problem Statement

Situation:

- Given n unique stars
- Arrange the stars in constellations of min size a and max size b
- How many **possible configurations** are there?

Input:

- Integers `n, a, b`
  - `max(1, n/10) <= a <= b <= n <= 50`
  - `b - a <= 5`

Output:

- Total number of constellations **modulo `10^9 + 7`**

Examples

Example 1

n = 3, a = 1, b = 3

```
Config 1: [1, 2, 3]
Config 2: [1, 2] [3]
Config 3: [1, 3] [2]
Config 4: [2, 3] [1]
Config 5: [1] [2, 3]
Config 6: [2] [1, 3]
Config 7: [3] [1, 2]
Config 8: [1] [2] [3]
Config 9: [1] [3] [2]
Config 10: [2] [1] [3]
Config 11: [2] [3] [1]
Config 12: [3] [1] [2]
Config 13: [3] [2] [1]
```

Total configurations: 13

Examples

Example 2

n = 7, a = 6, b = 7

```
Config 1: [1, 2, 3, 4, 5, 6]
Config 2: [1, 2, 3, 4, 5, 7]
Config 3: [1, 2, 3, 4, 6, 7]
Config 4: [1, 2, 3, 5, 6, 7]
Config 5: [1, 2, 4, 5, 6, 7]
Config 6: [1, 3, 4, 5, 6, 7]
Config 7: [2, 3, 4, 5, 6, 7]
Config 8: [1, 2, 3, 4, 5, 6, 7]
```

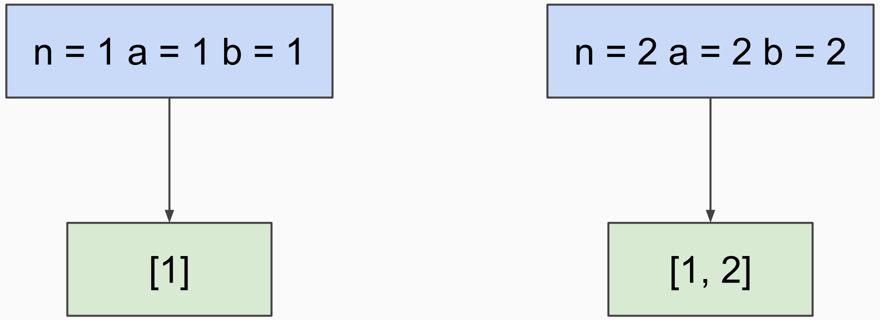
Total configurations: 8

Things to Consider

- Objective: calculate number of all possible constellation combination modulo by `10^9 + 7` (`1000000007`)
- Time complexity
  - Inputs are relatively small and further restricted (`b – a <= 5`)
  - `2^N` algorithm should still pass
- Recursion depth
  - Java max recursion depth ~1000-10000 (depending on memory)
  - Should not hit StackOverflowError
- Variable types
  - Hint: `long` is not big enough to store the total result
  - Either use `BigInteger` or perform modulo operation to prevent overflow

Approach

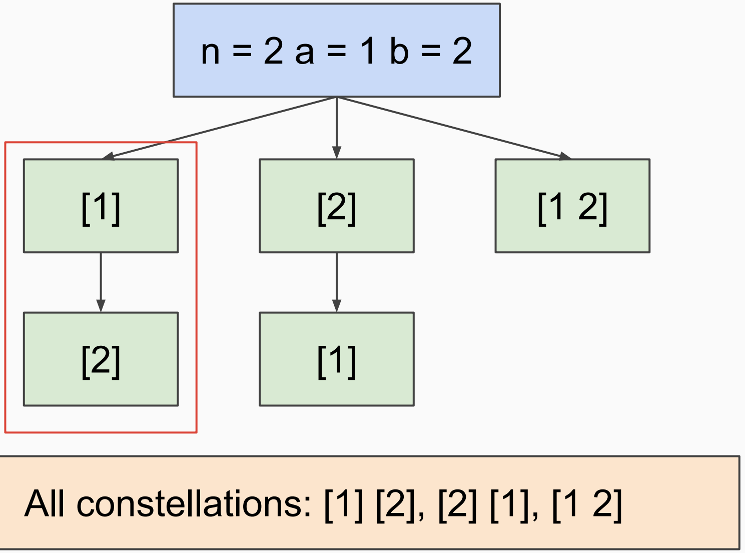
- The question can be difficult and overwhelming at start
- Sometimes it's helpful to think "how would you do it if you were asked to do it manually?" and visualise it using a decision tree
- Try it out with a small case first!
  - What is the simplest case?
  - When n == a (and by extension n == b since b cannot be > n)



Approach

- How about more complex cases?
  - Try drawing it out and see if there’s any similarity with a simpler case

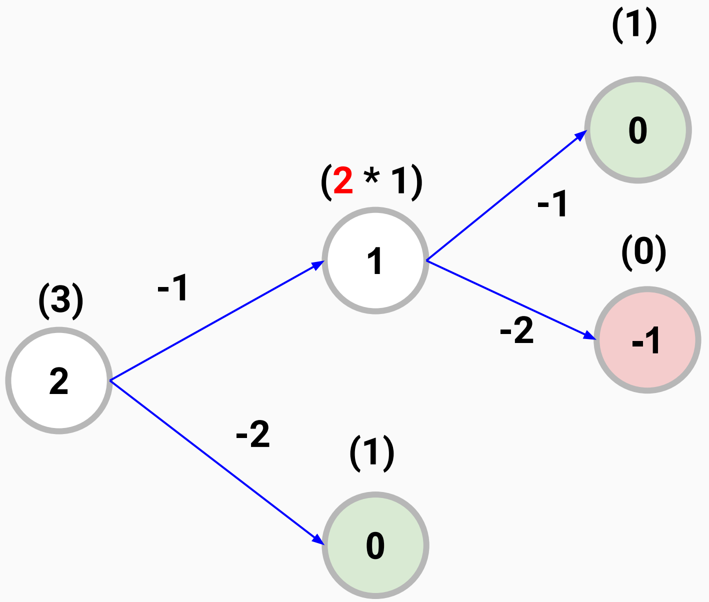
Looks familiar?



All constellations: [1] [2], [2] [1], [1 2]

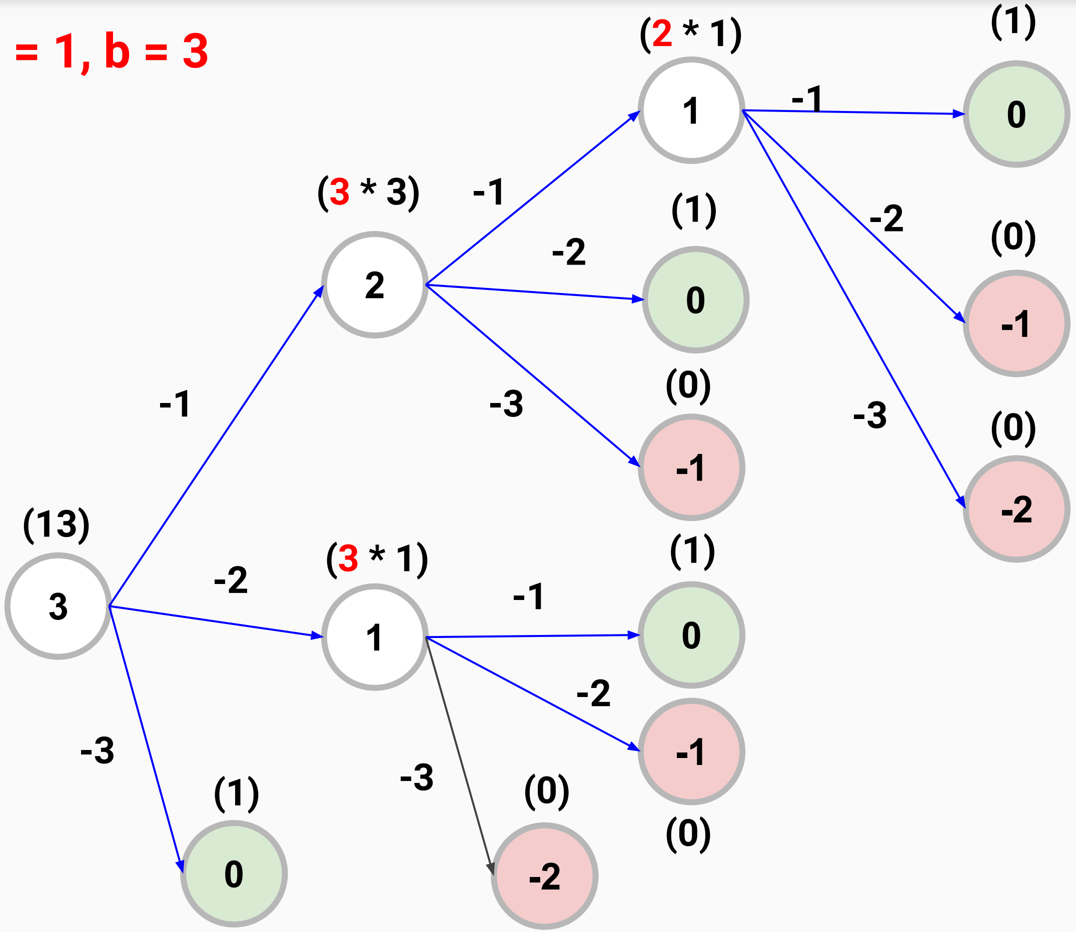
A more familiar approach to visualising the question

For **n = 2, a = 1, b = 2**



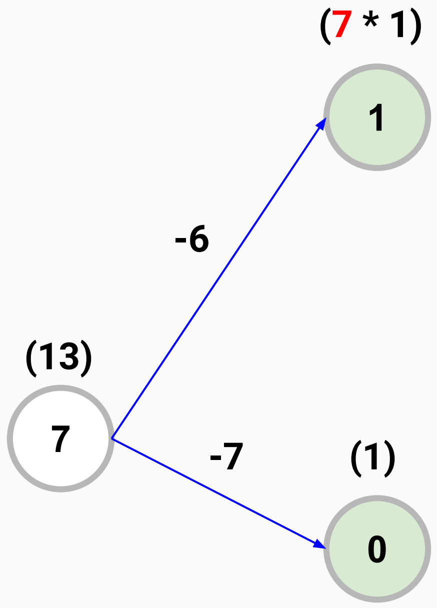
A more familiar approach to visualising the question cont.

For **n = 3, a = 1, b = 3**

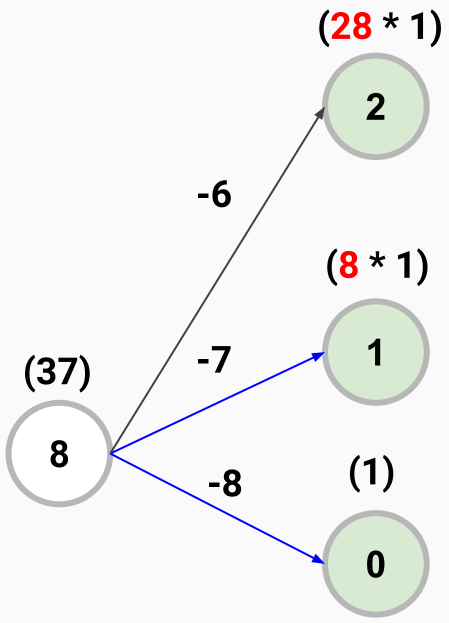


A more familiar approach to visualising the question cont.

For **n = 7, a = 6, b = 7**

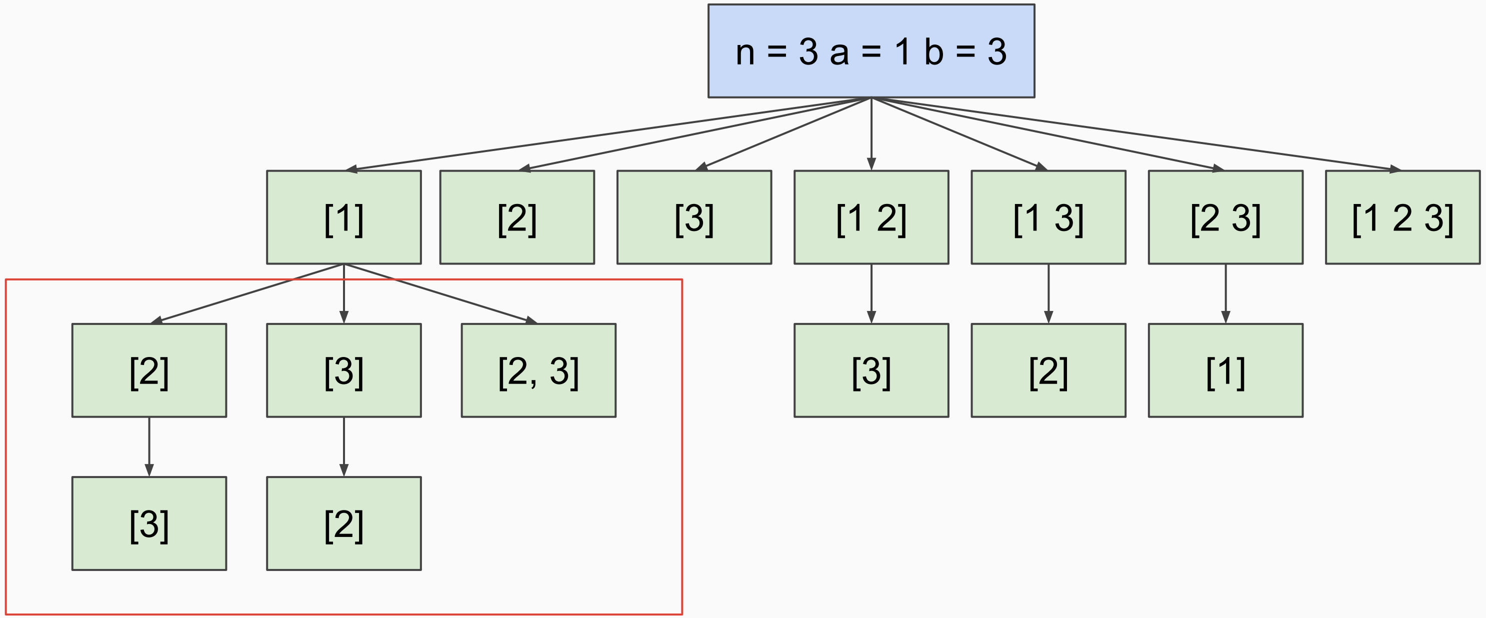


For **n = 8, a = 6, b = 8**



What is the additional base case here?

Approach



Looks familiar?

Further Considerations

- Do you need to store all the possible combinations?
  - No, we only need the number of possibilities
- Where should you perform the modulo operations?
  - Wherever there’s a possibility of the result exceeding the max limit
- How do I check if I’m not exceeding the max limit of my variables?
  - Try to come up with a test case that results in the highest number of possibilities
  - Print out all your variables to make sure there’s no overflow

Based on what we discussed so far, what are the base case(s) for this question?

n = numberOfTotalStars

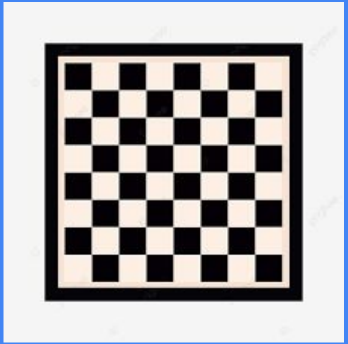
a = minNumOfStarsToChoose

b = maxNumOfStarsToChoose

if (numberOfTotalStars < 0)	**A**
if (numberOfTotalStars <= minNumOfStarsToChoose)	**B**
if (numberOfTotalStars <= maxNumOfStarsToChoose)	**C**
if (minNumOfStarsToChoose < 0)	**D**
if (numberOfTotalStars >= minNumOfStarsToChoose)	**E**

Lab 4C

Queens



Problem Statement

Timmy is playing chess with his brother, Jimmy. Unfortunately, both of them do not know the rules of chess, except the fact that the queen piece can move and attack horizontally, vertically or diagonally from its own position. They decide to play on n x n chessboard.

Jimmy has initially placed m queens on the board, with none of them attacking each other.

Timmy needs to place the remaining n-m queens on the board such that no queens attack each other. Help him find out whether it is possible to do so.

Problem Statement

<u>Situation:</u>
We have a **n x n** chess board.
Queen piece can move and attack horizontally, vertically or diagonally from its own position.
There are **m** pieces of queen on the board.
We must find if it is possible to add the remaining n - m pieces on the board such that they do not attack each other.

<u>Input:</u>
- Size of board 1 <= n <= 12
- Initial number of queens on board m 0 <= m <= n

<u>Output:</u>
- true or false

Examples

On a 4 X 4 board the following combinations are possible.
They do not attack each other in this position

	Q		
			Q
Q			
		Q	

		Q	
Q			
			Q
	Q		

Approach 1(Brute force)

- Generate all possible combinations on the chessboard.
- For a 4 X 4 chessboard there are 16C4 combinations, which is equal to 1820.
- We filter those combinations with the queen in the fixed position and then loop through all the different combinations, if a combination can be found we return true.
- If there are no possible combination available that satisfies the criteria return false.

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

Approach 1 analysis

- Since we are generating all the possible combinations there are a total n^2Cn combinations. Time complexity is $O(n^2Cn)$.
- Worst case we get 12^2C12 combinations which is equal to 1 X 10^{17}.
- Since it is significantly greater than the allowed 1 X 10^8 this solution will TLE.

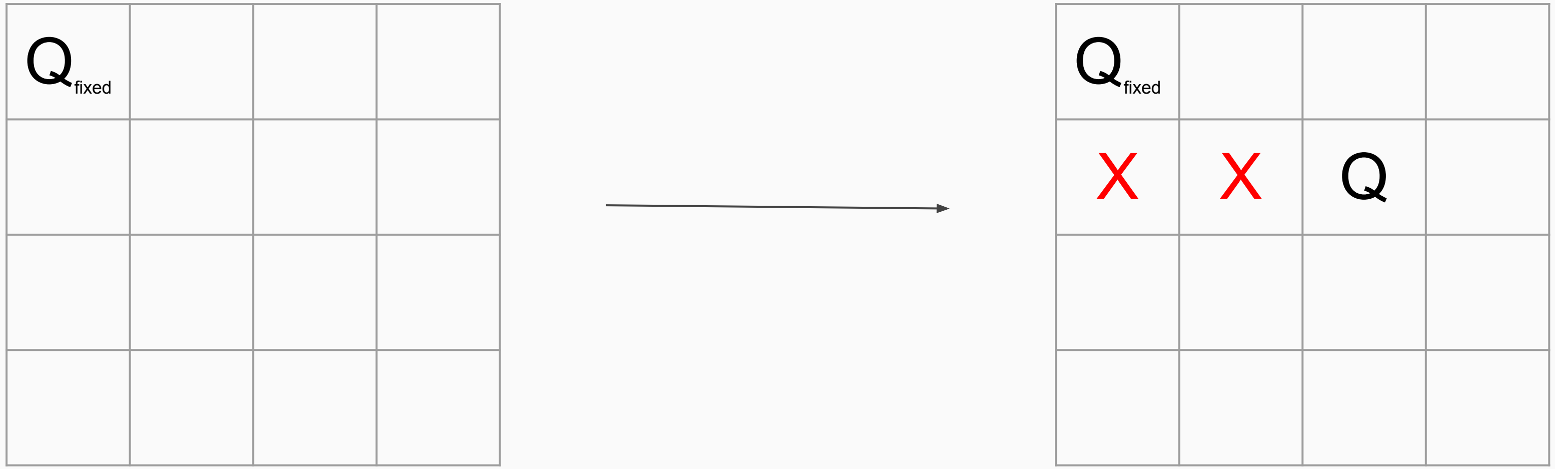
Approach 2 (backtracking)

- Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time.
- We know that each row and column can only have at most 1 queen.
- The idea is to place queens one by one in different columns, starting from the leftmost column.
- When we place a queen in a column, we check for clashes with already placed queens.
- In the current row, if we find a column for which there is no clash, we mark this row and column as part of the solution

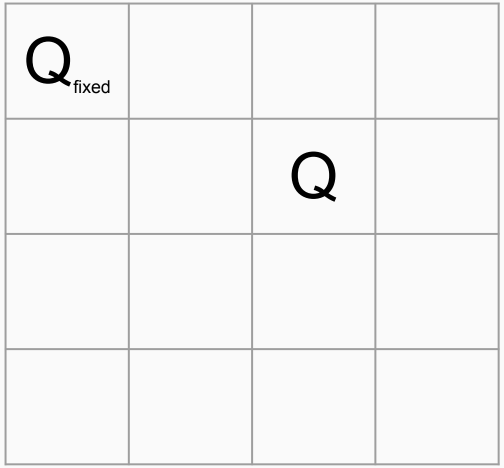
Approach 2

Example on a 4 X 4 matrix

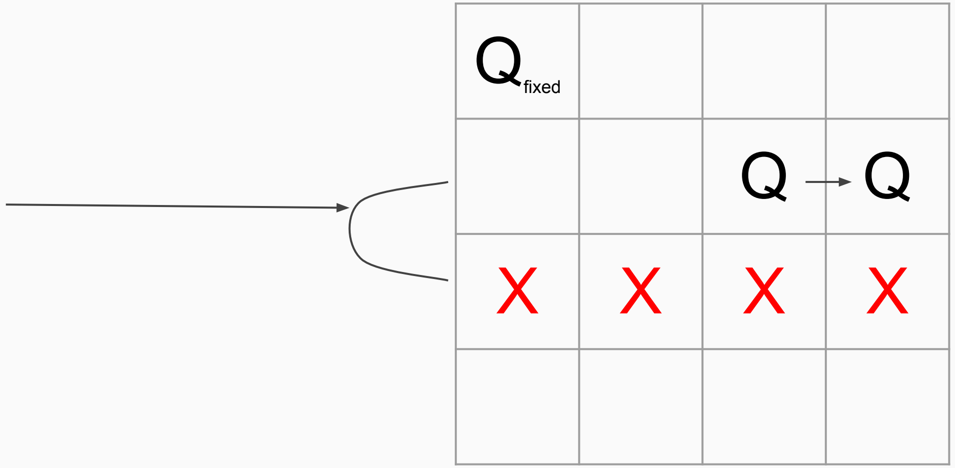
Second row, first available column is index 2



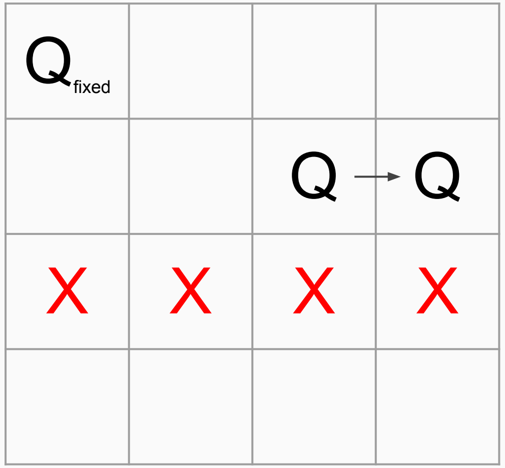
Approach 2



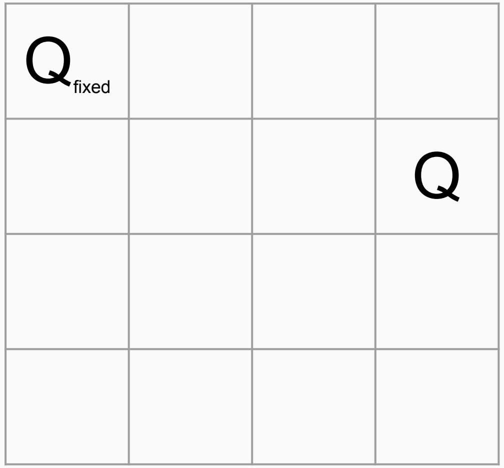
Third row cannot insert queen, recurse to second row and move one column right



Approach 2

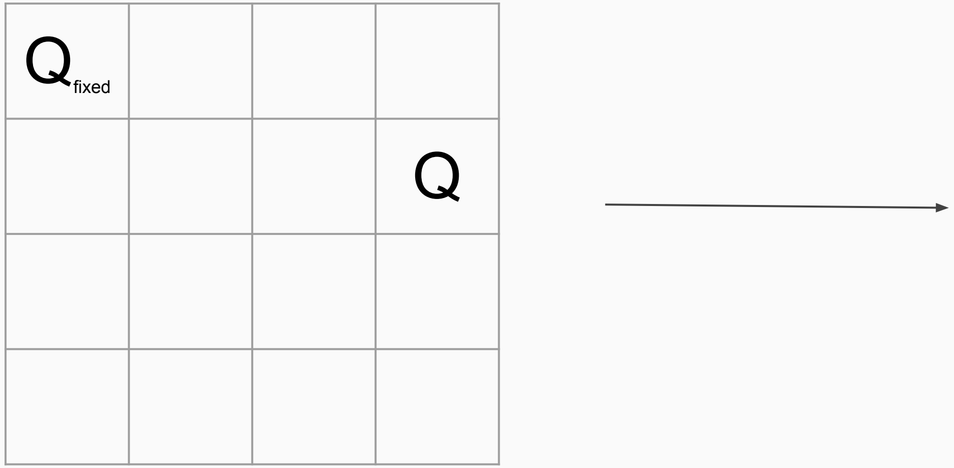


Second row queen move one column right

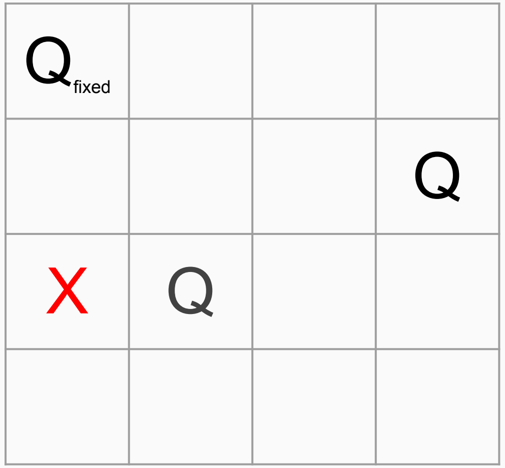


Approach 2

Example on a 4 X 4 matrix

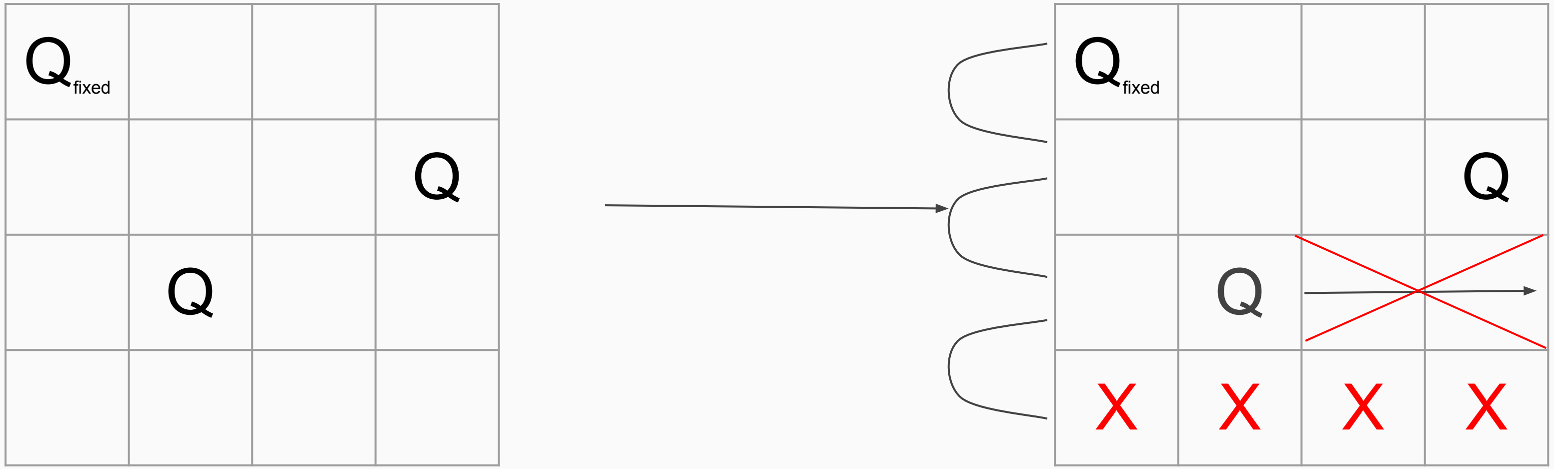


Third row first available column is index 1



Approach 2

Example on a 4 X 4 matrix

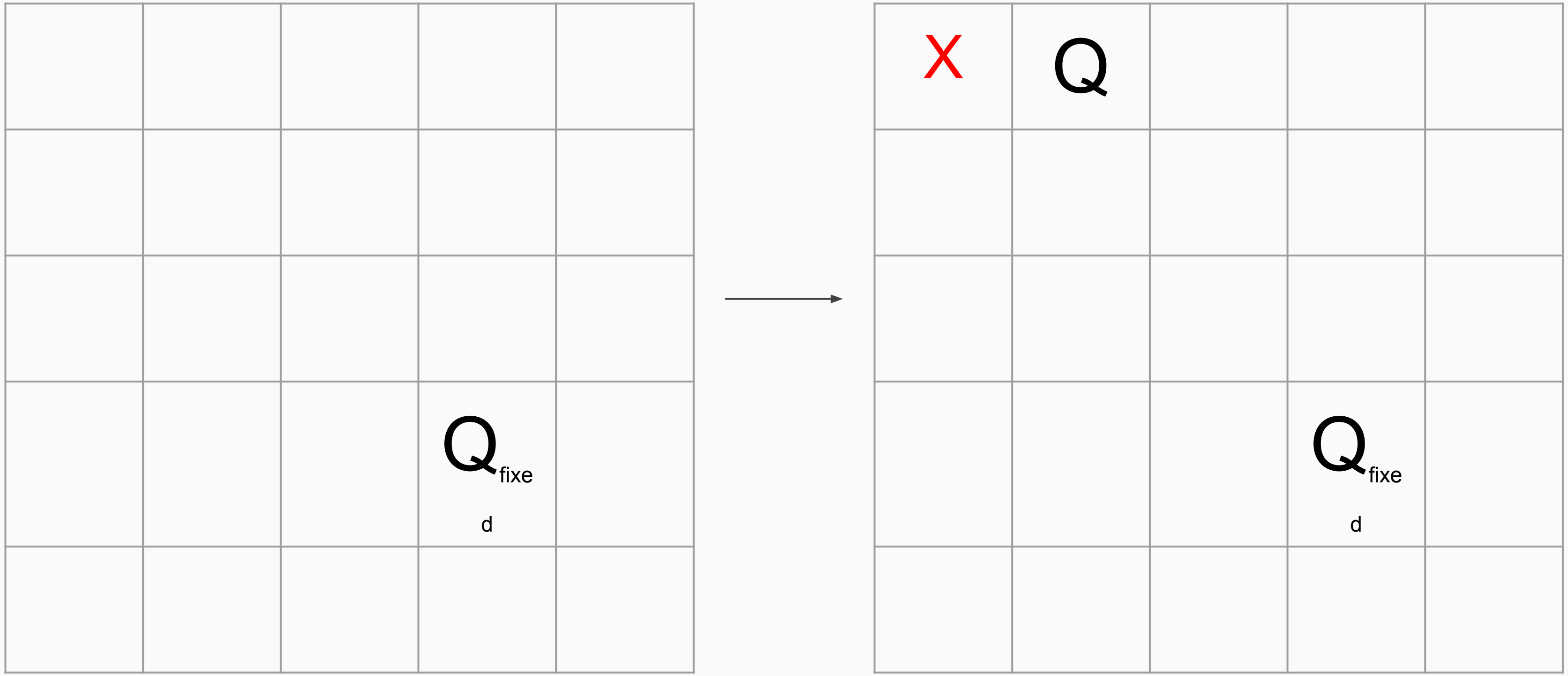


Fourth row cannot insert queen, third row queen cannot move right and second row queen already at most right (No solution!!)

Approach 2

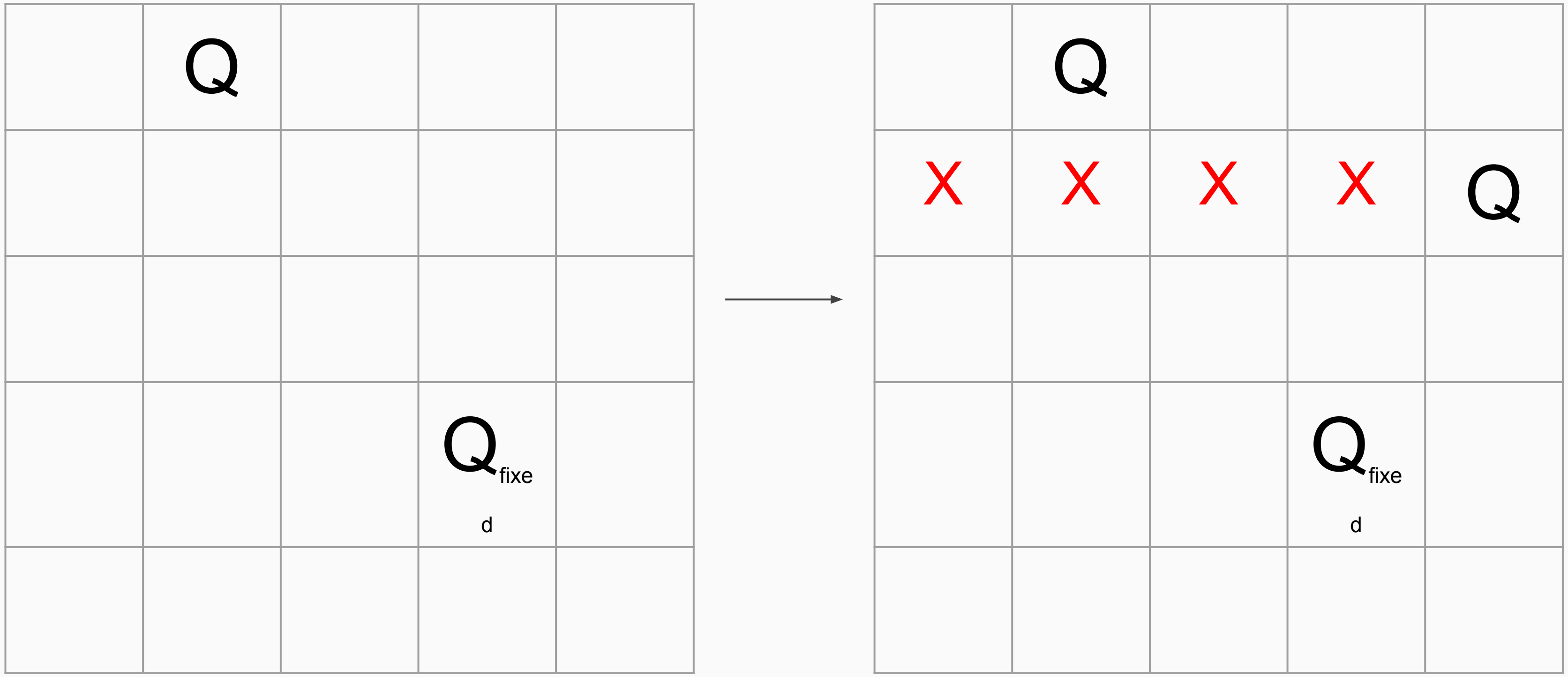
Example on a 5 X 5 matrix

First row first available column is index 1



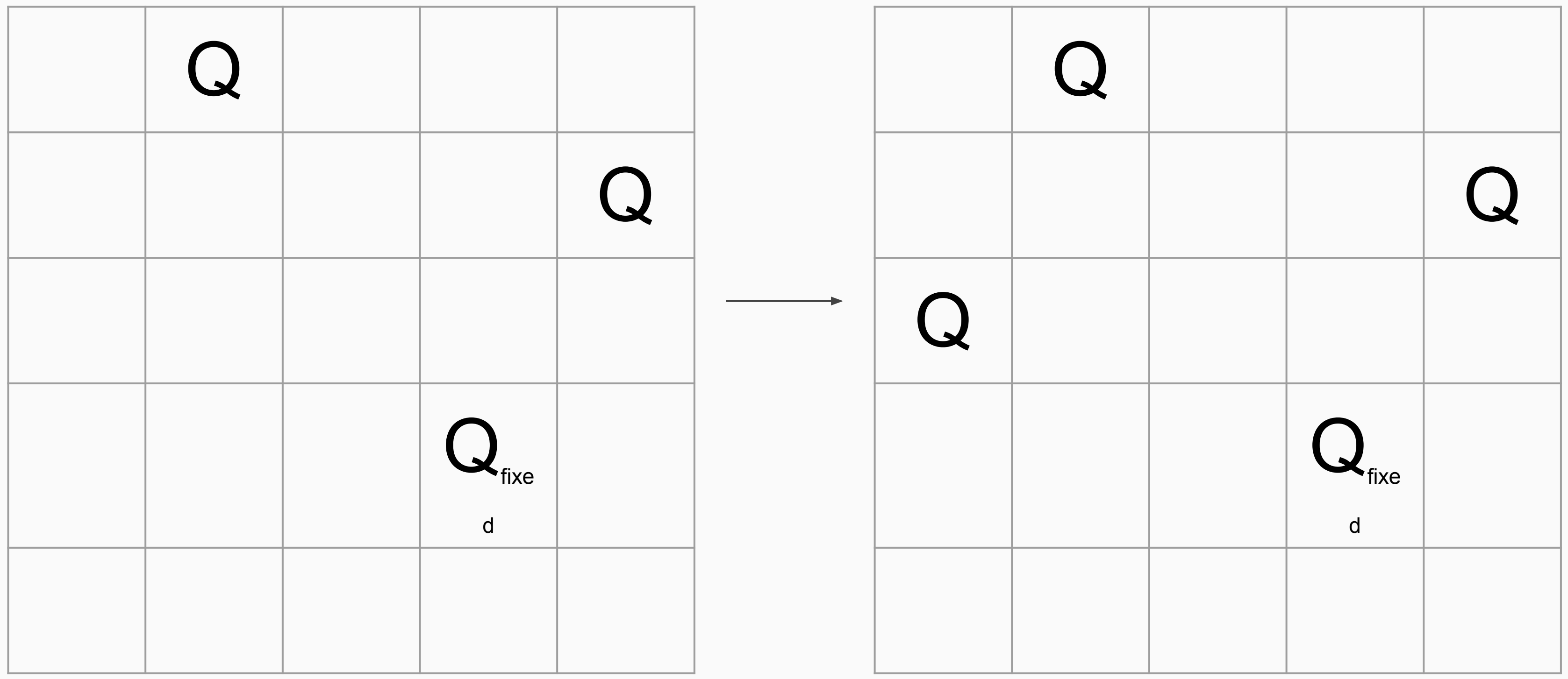
Approach 2

Second row first available column is index 4



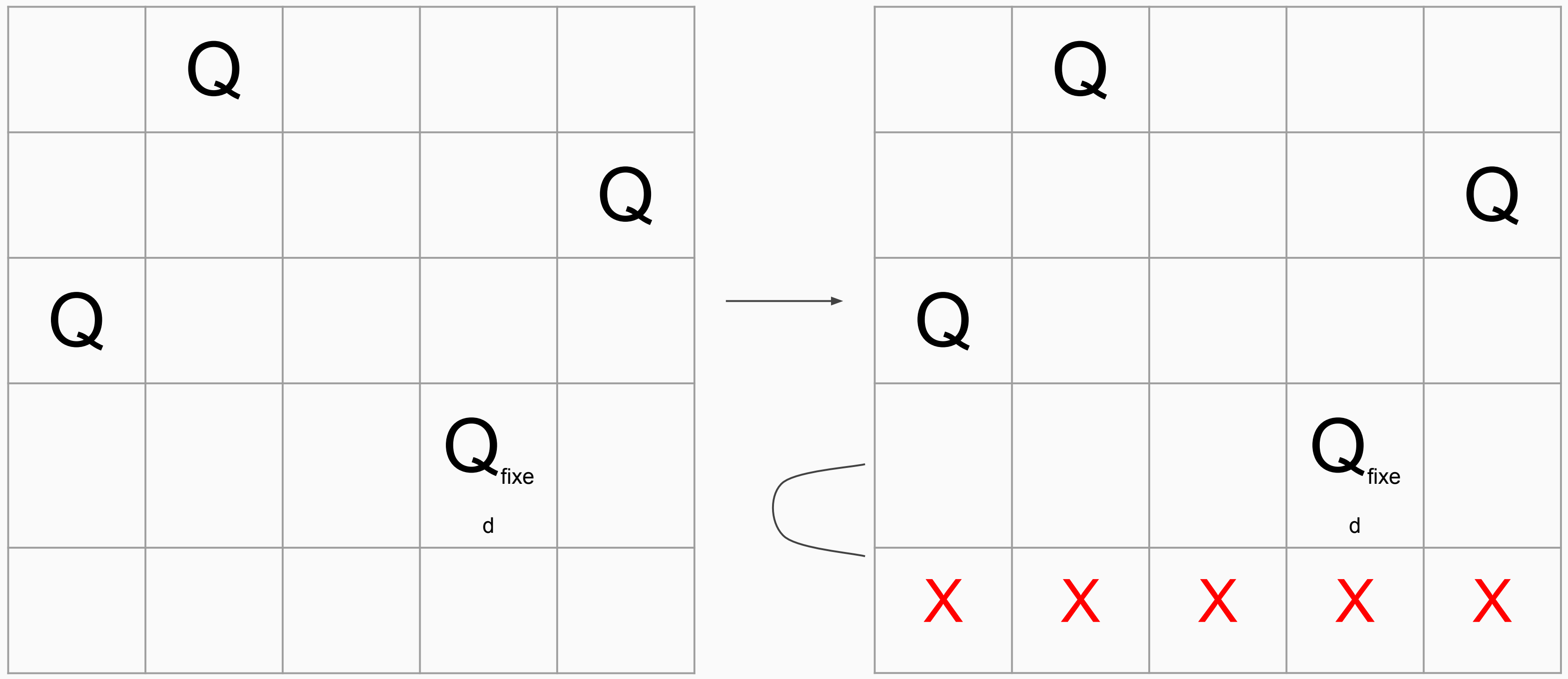
Approach 2

Third row first available column is index 0

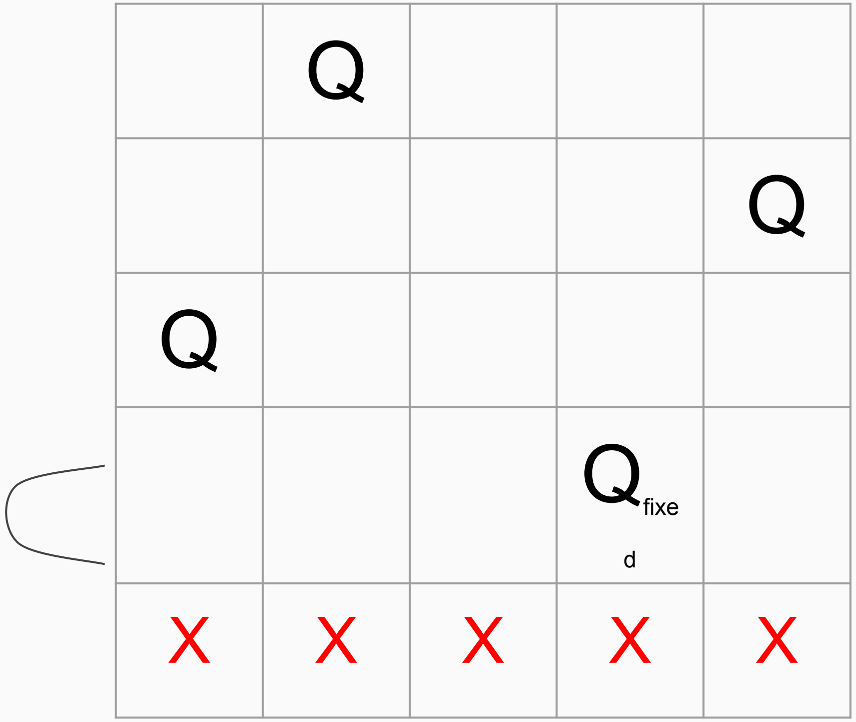


Approach 2

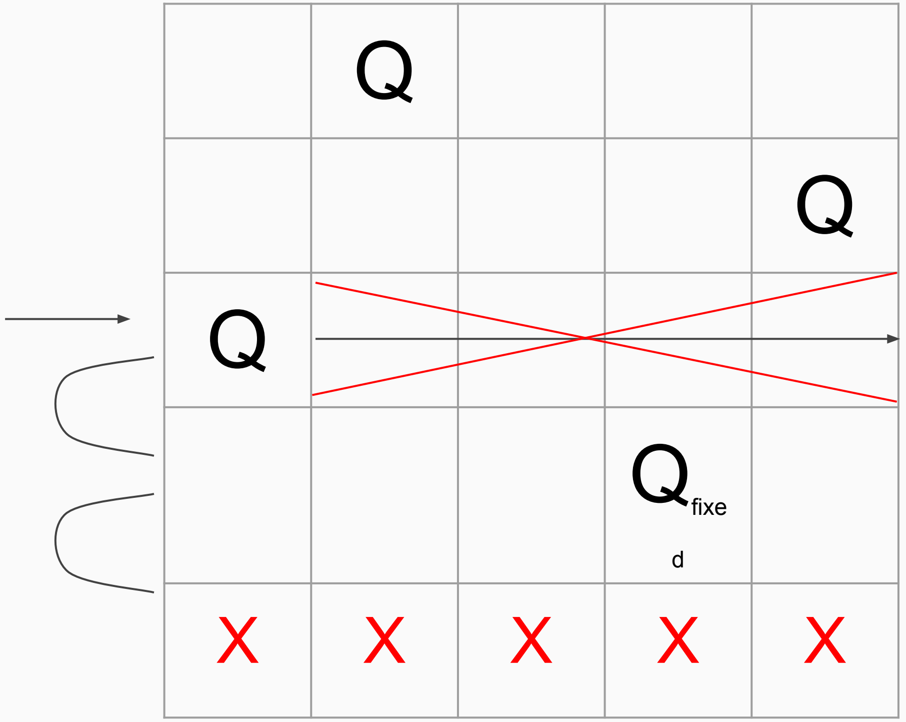
Fifth row cannot insert queen, recurse to fourth row but fourth row is fixed queen



Approach 2

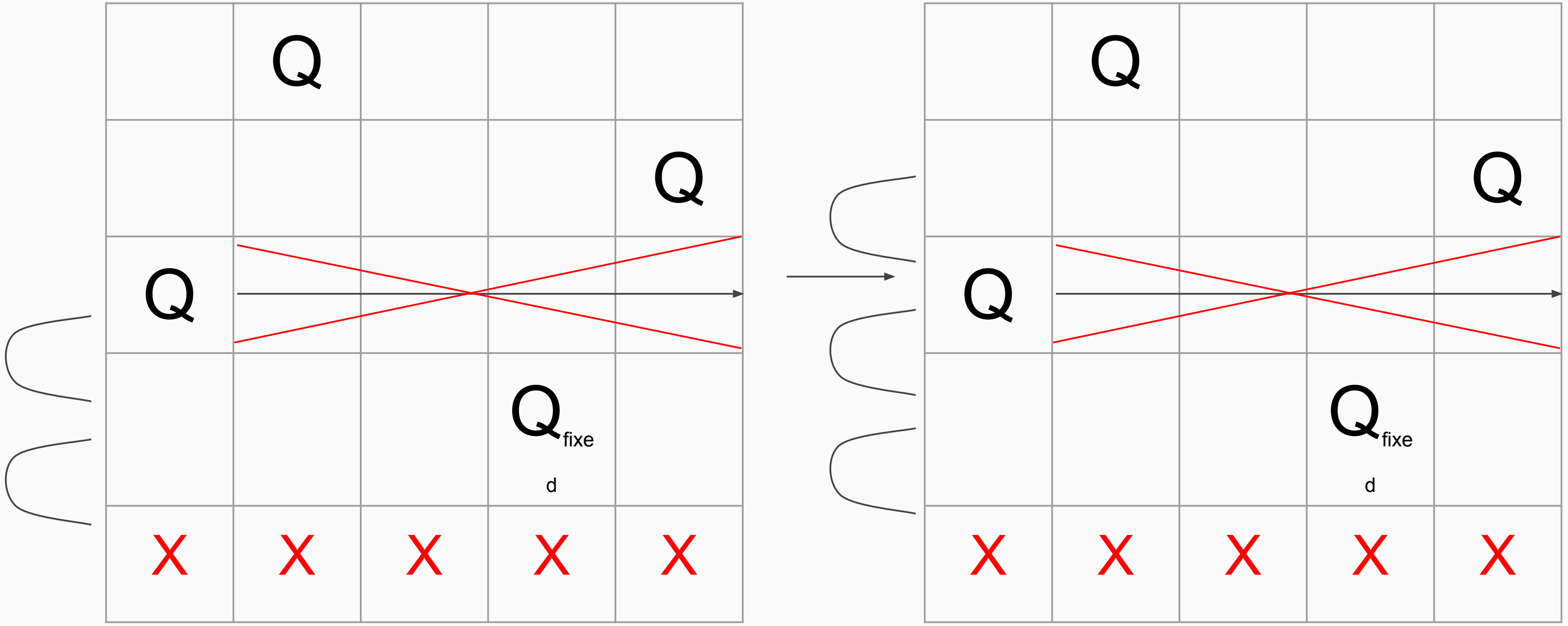


Fourth row is fixed queen, so we recurse to third row, but third row cannot move to any column to the right



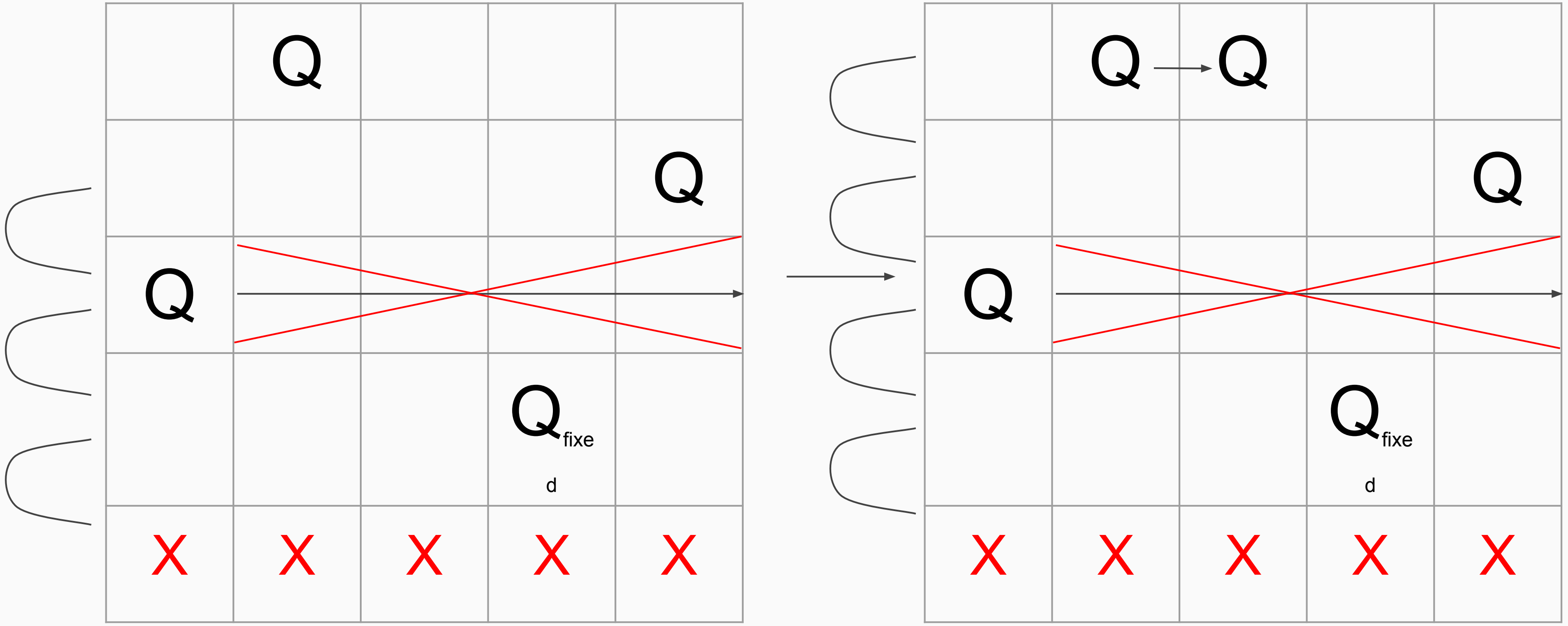
Approach 2

Third row cannot move queen, recurse to second row but second row queen is at right most column

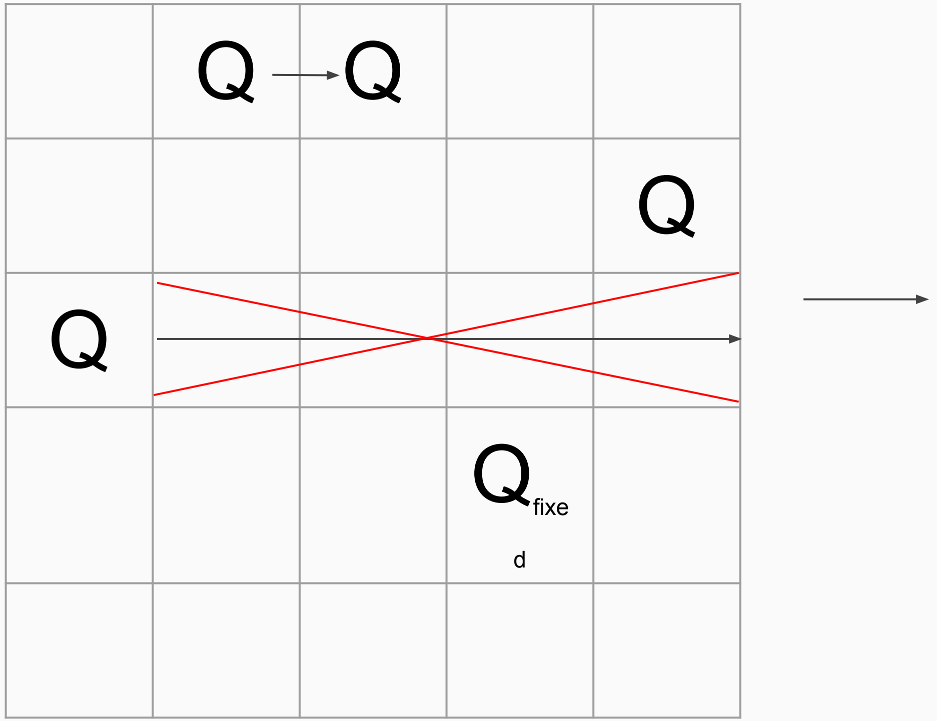


Approach 2

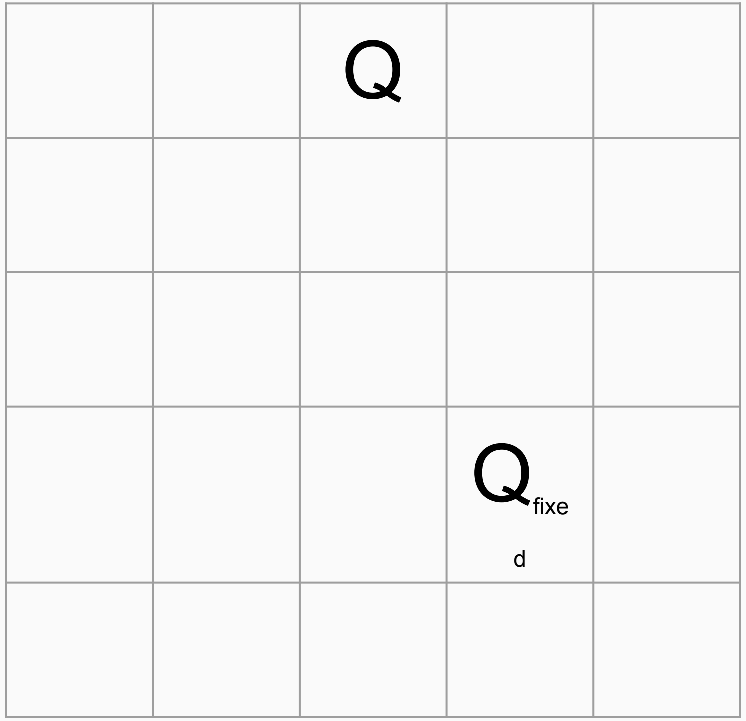
Second row cannot move queen, recurse to first row, first row queen move one column right



Approach 2

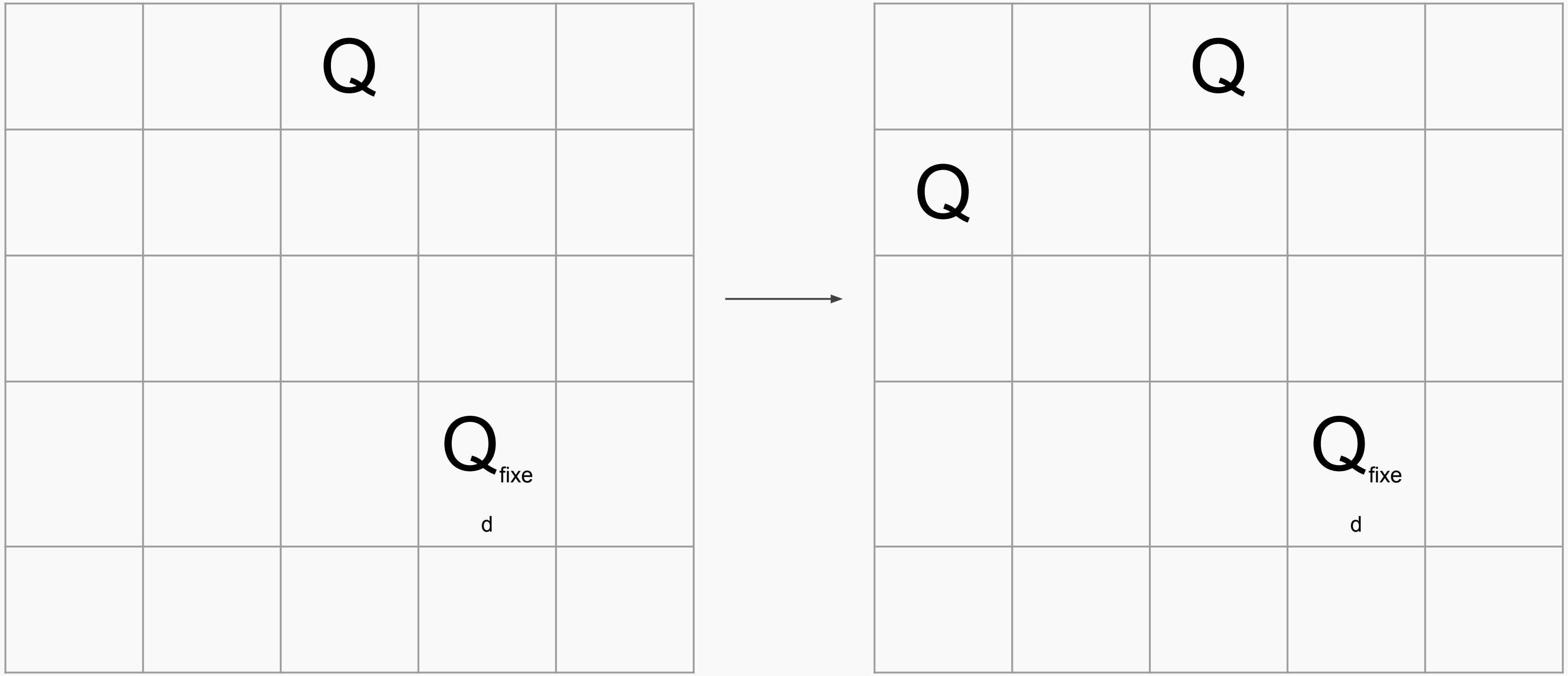


First row queen move right

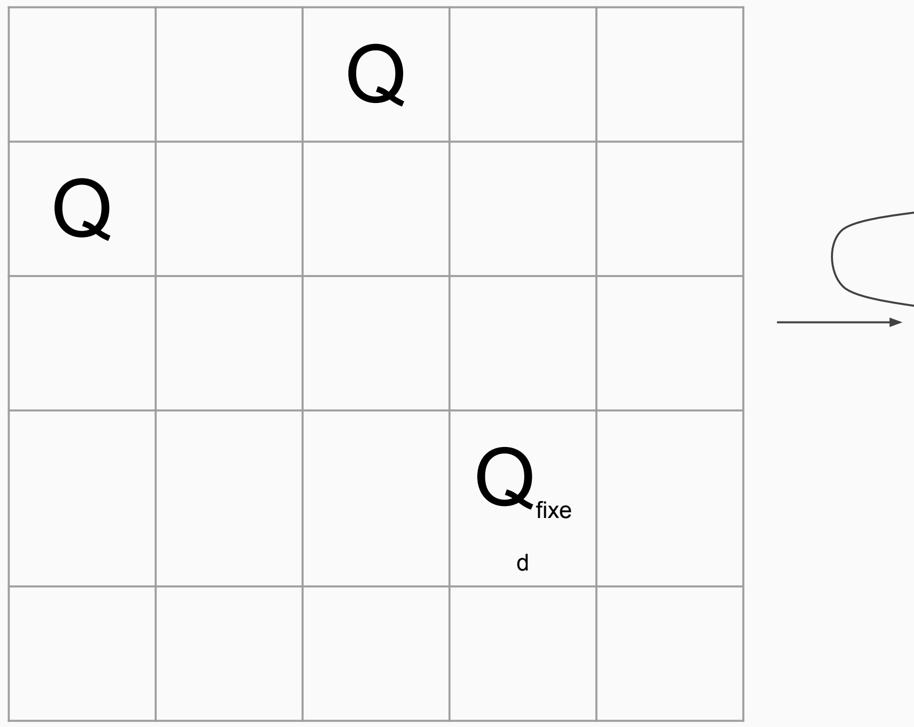


Approach 2

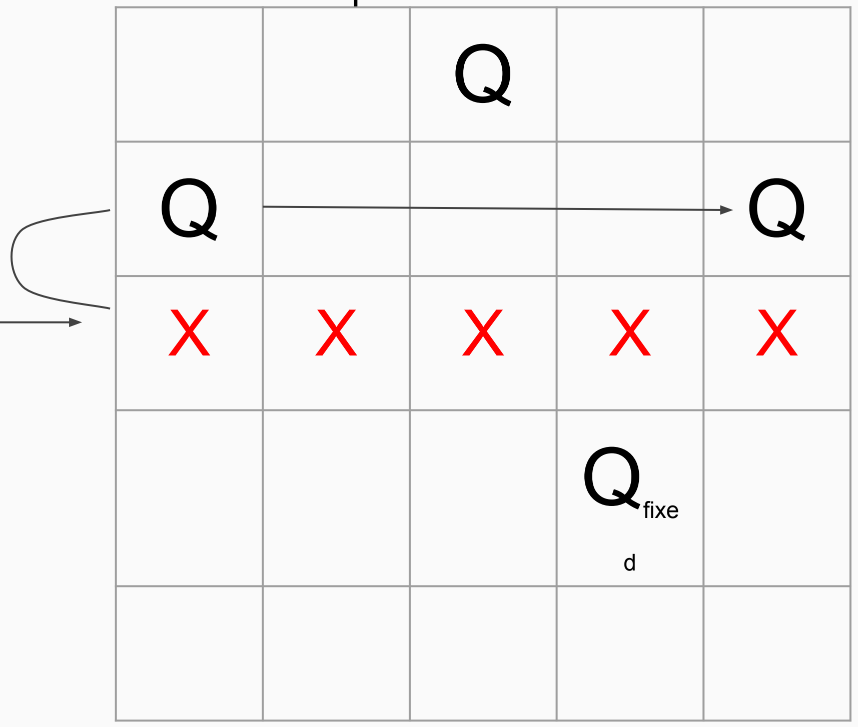
Second row queen first available position is index 0



Approach 2

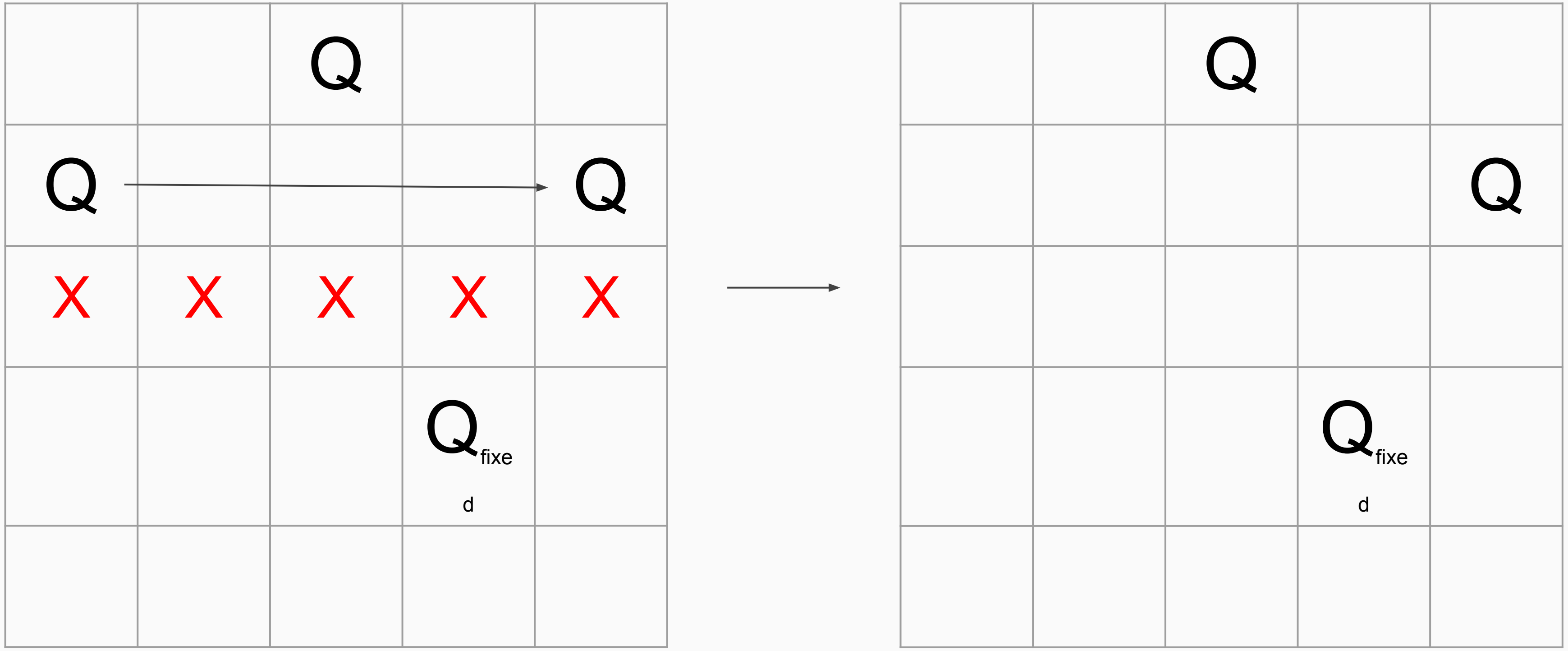


Third row cannot insert queen, recurse to second row and shift queen to next available column



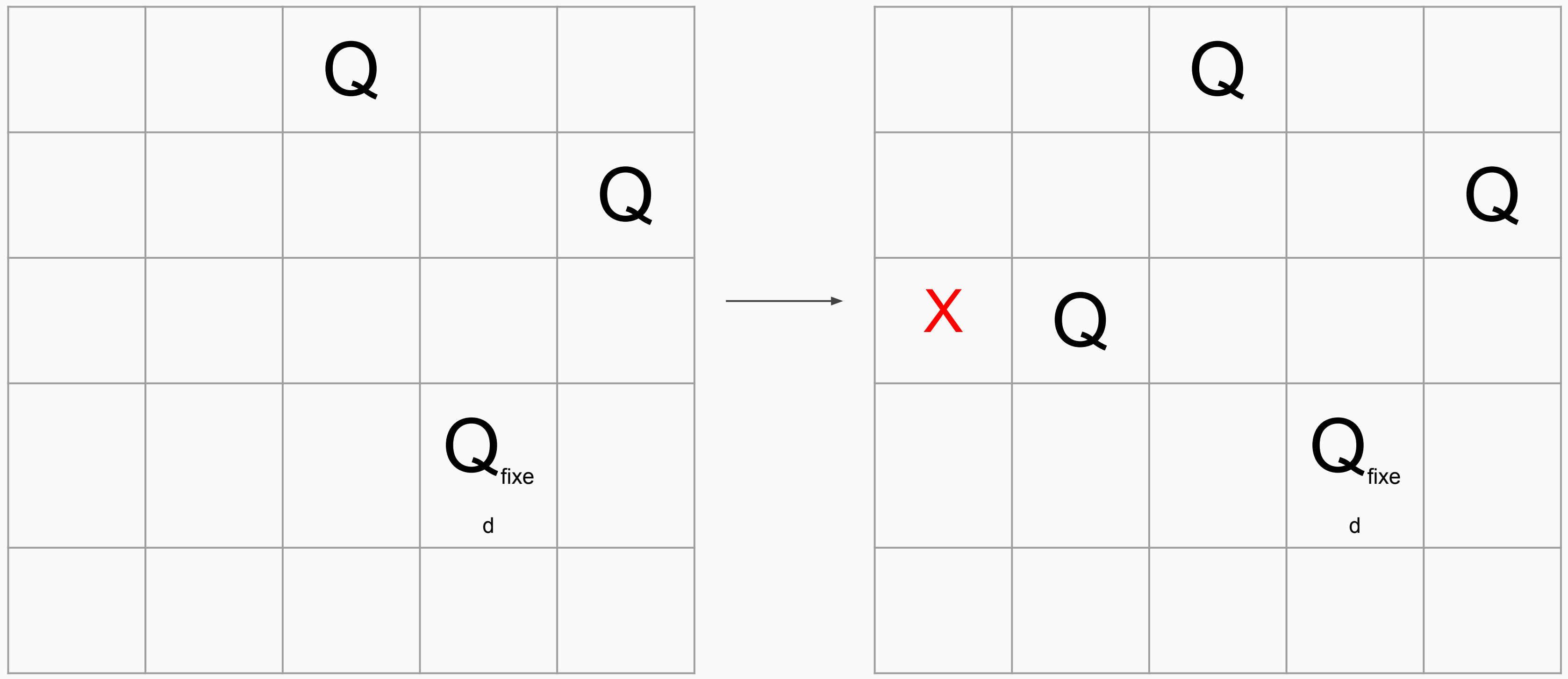
Approach 2

Second row queen move right.



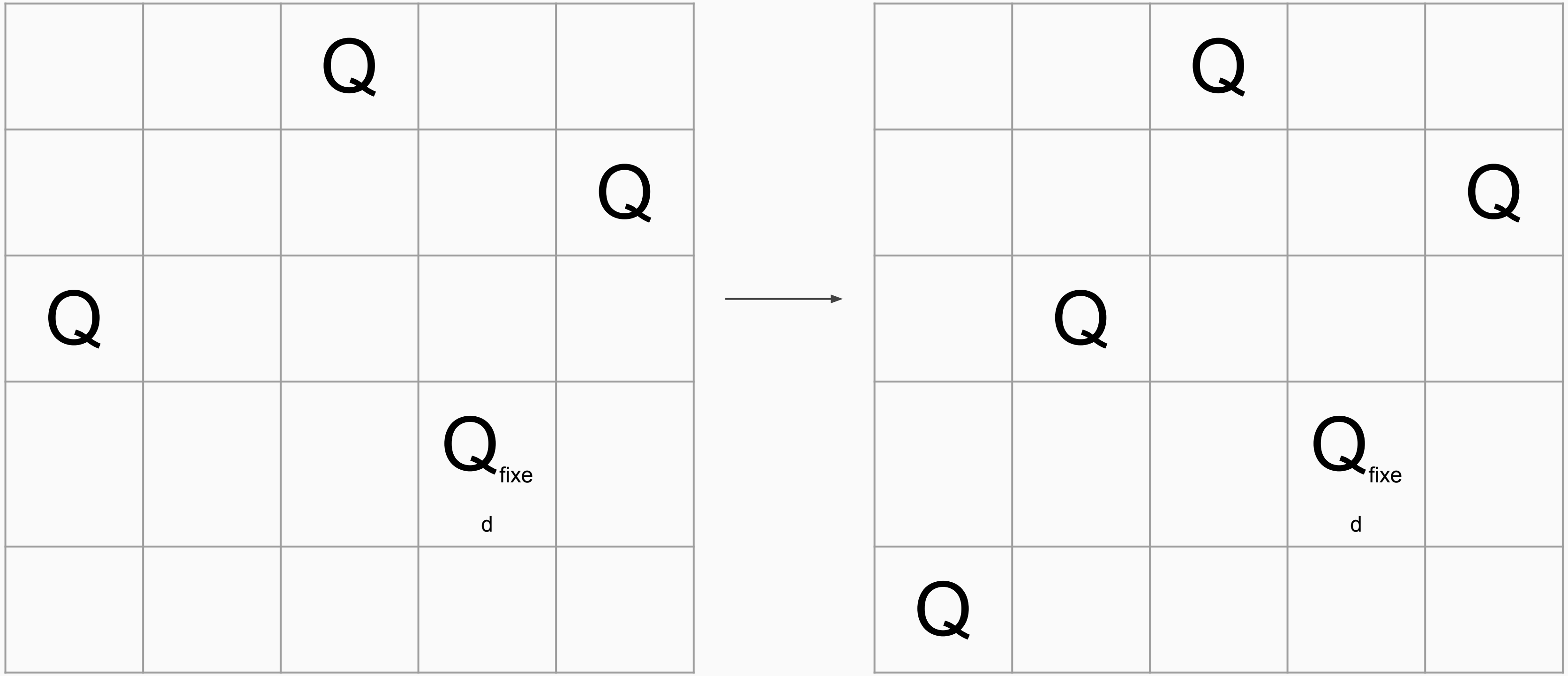
Approach 2

Third row queen first available position is index 1



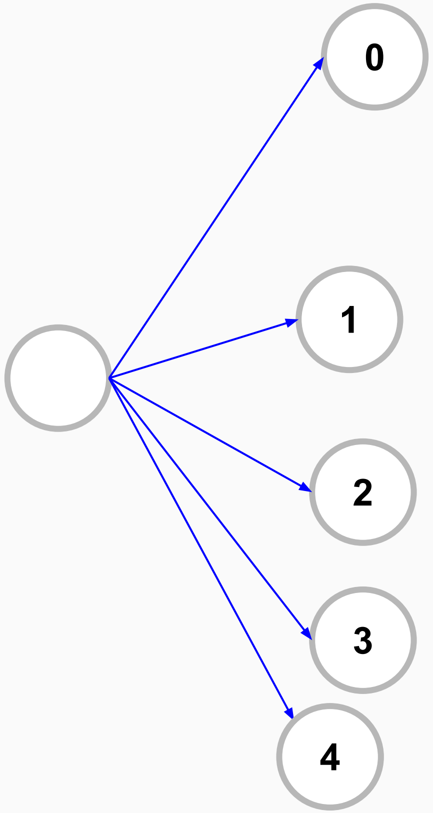
Approach 2

Fifth row (Solution)



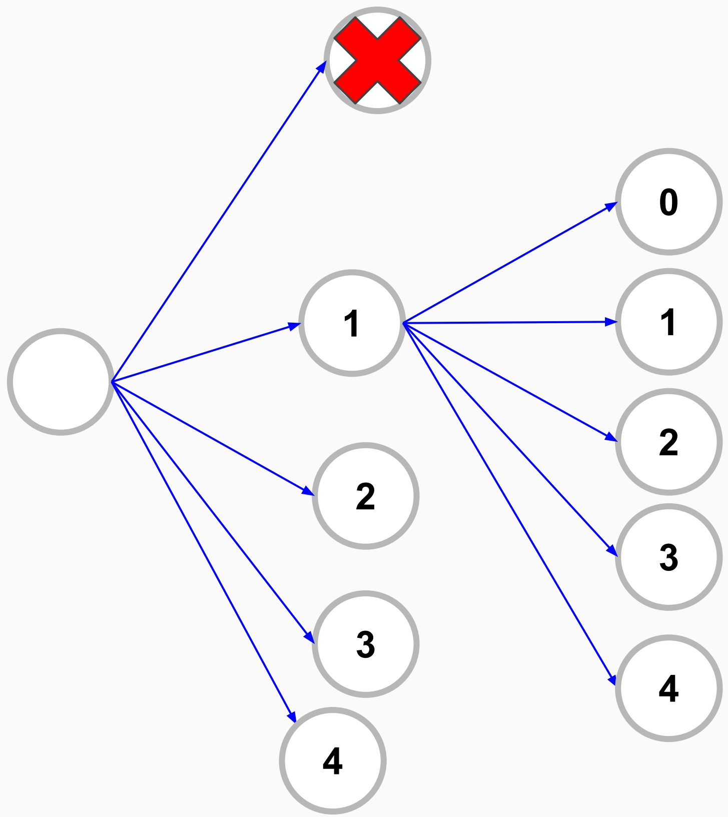
Approach 2 (Time complexity)

For **N = 5**



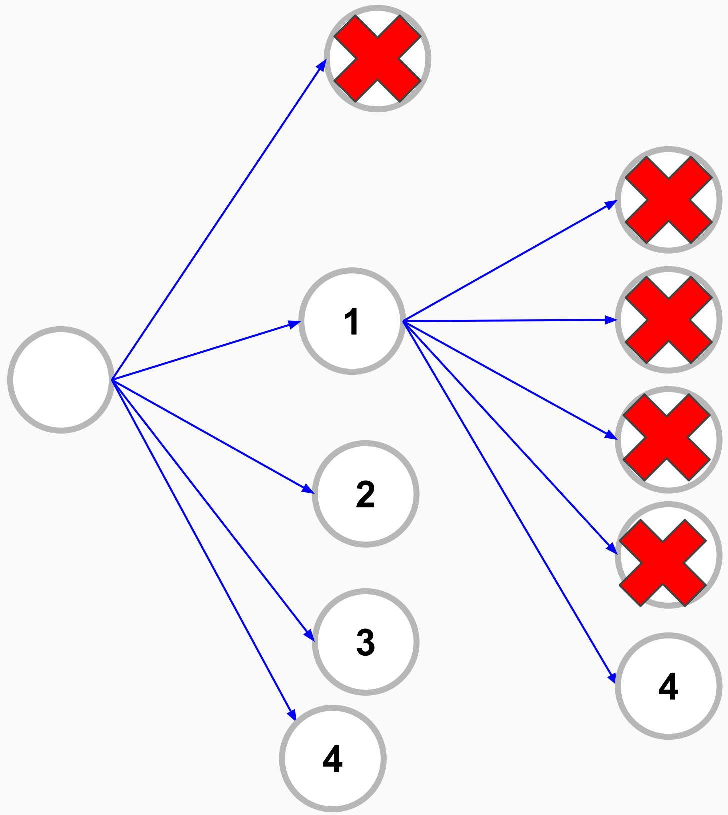
Approach 2 (Time complexity simulation)

For **N = 5**



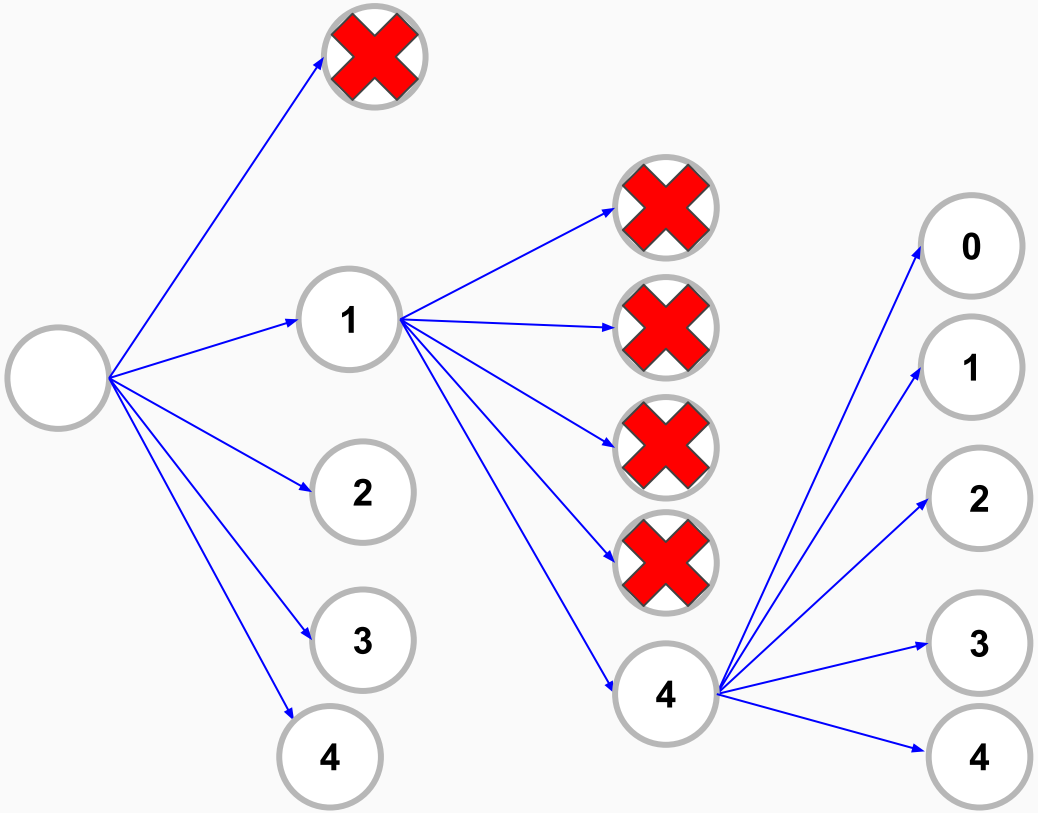
Approach 2 (Time complexity simulation)

For **N = 5**



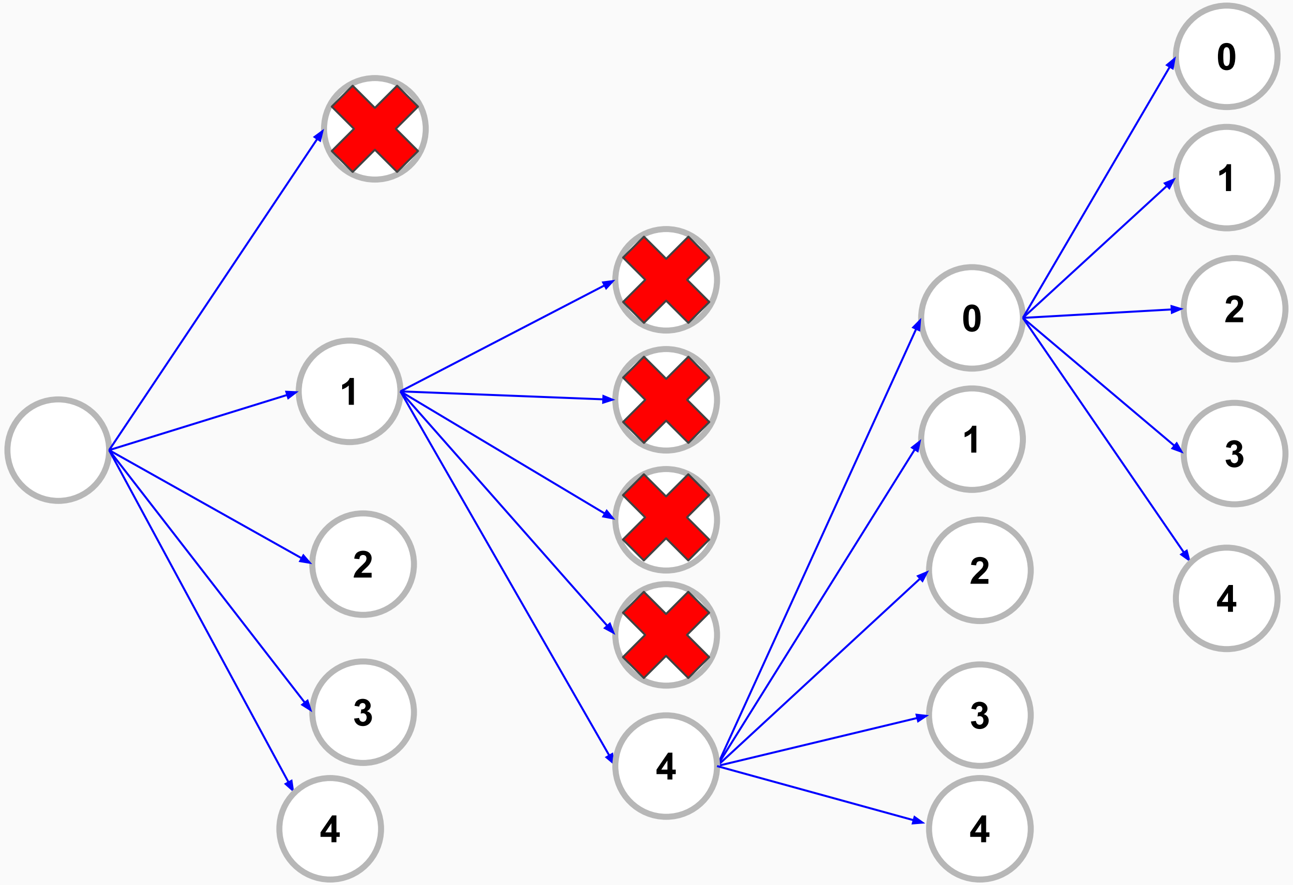
Approach 2 (Time complexity simulation)

For **N = 5**



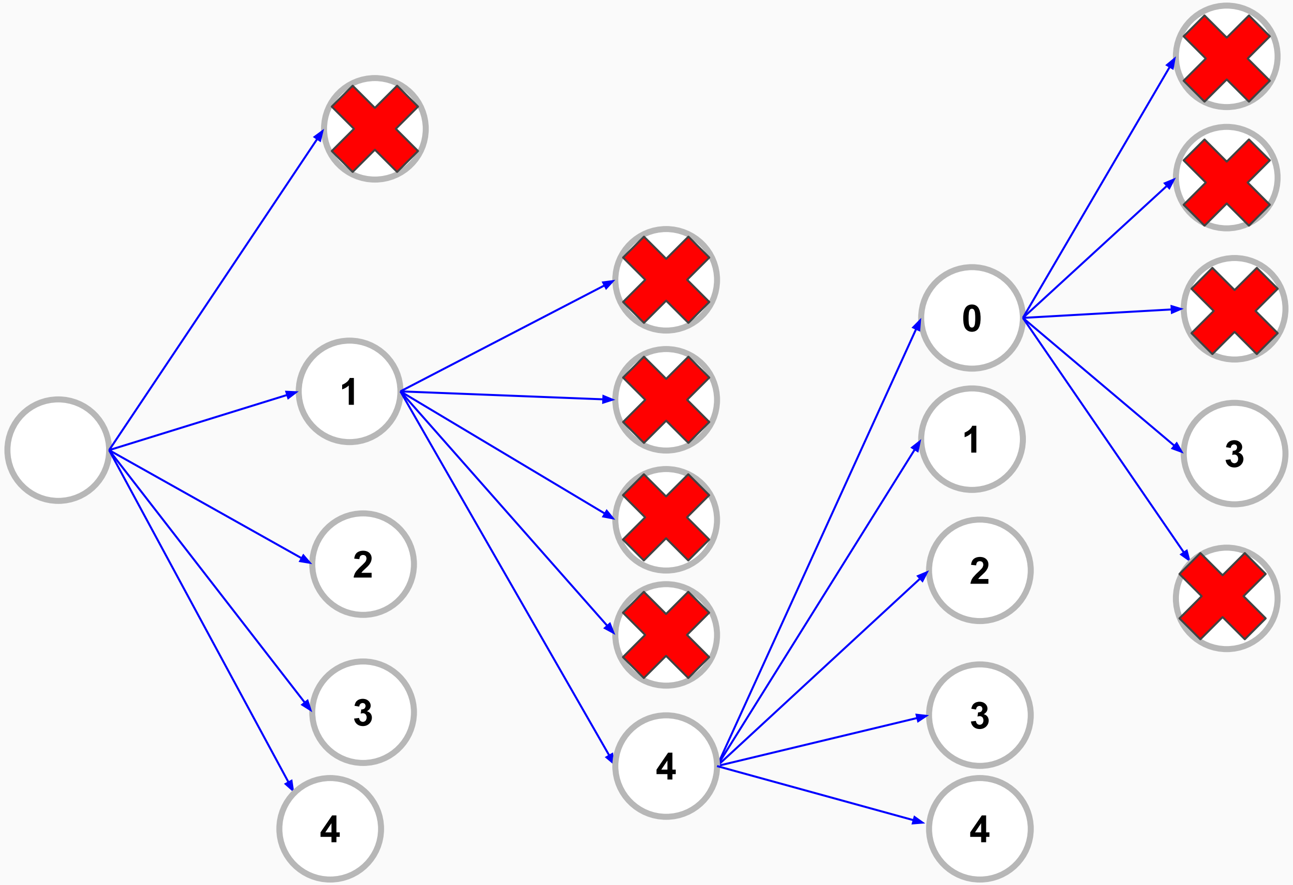
Approach 2 (Time complexity simulation)

For **N = 5**



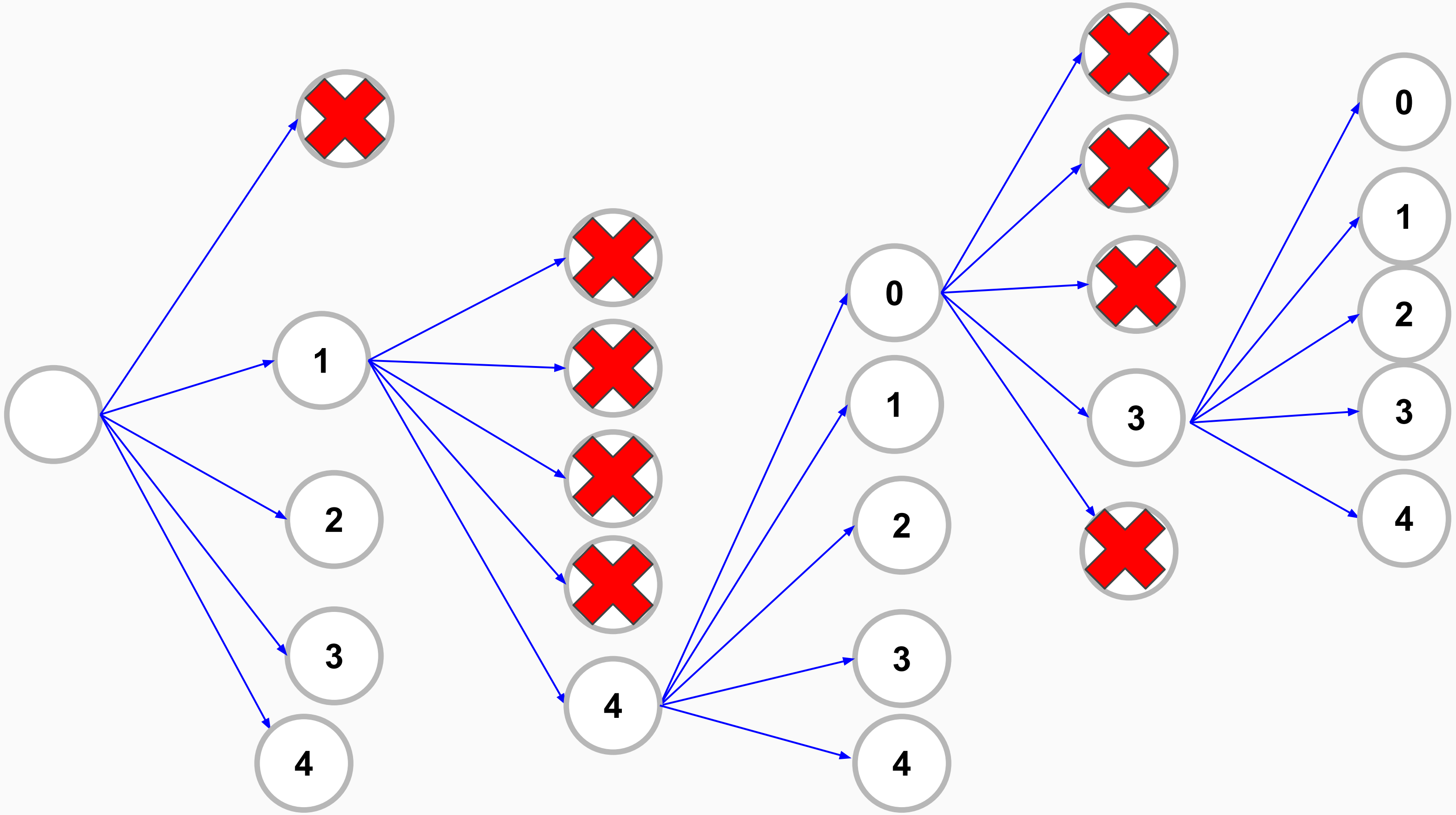
Approach 2 (Time complexity simulation)

For **N = 5**



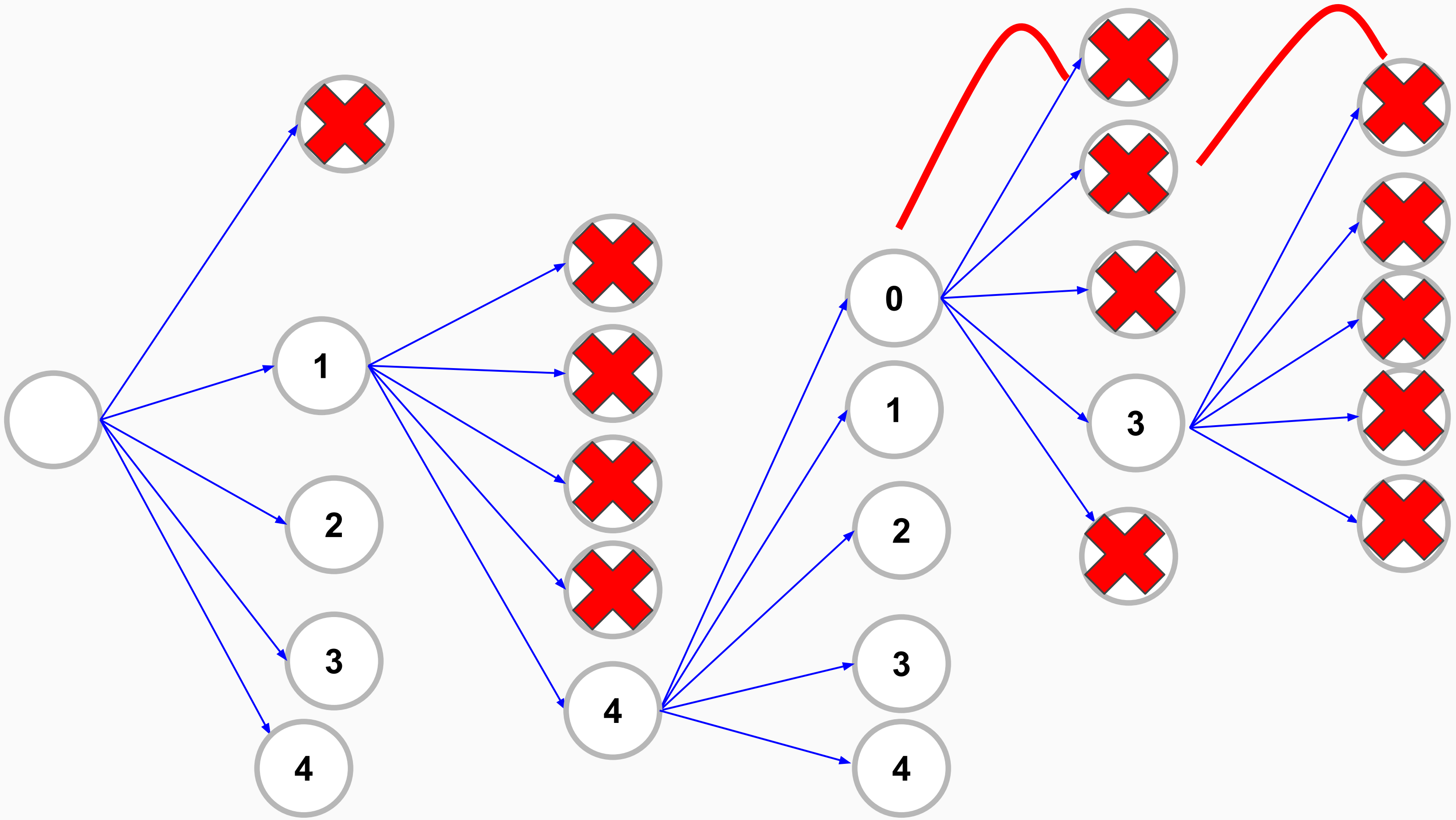
Approach 2 (Time complexity simulation)

For **N = 5**



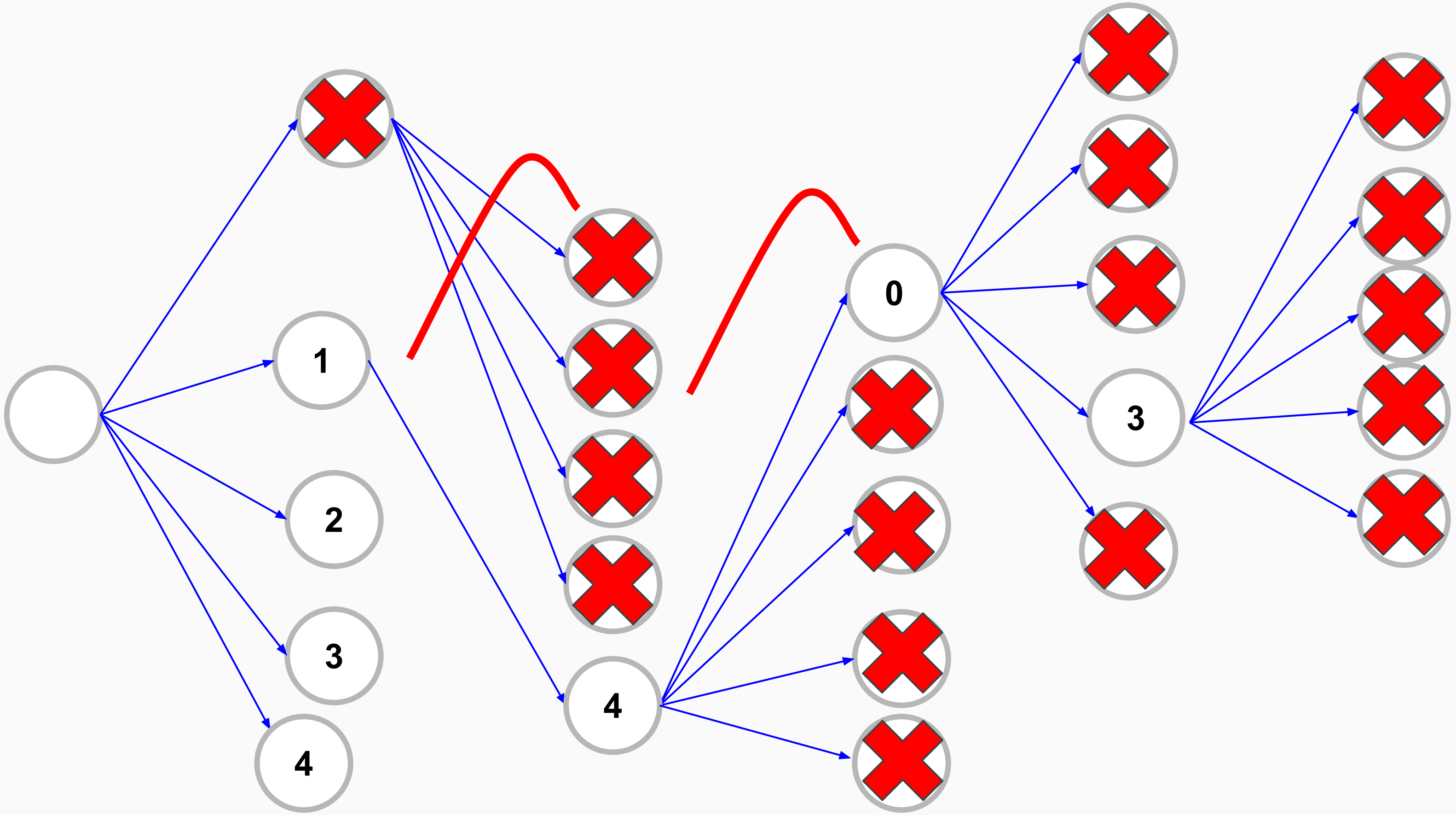
Approach 2 (Time complexity simulation)

For **N = 5**



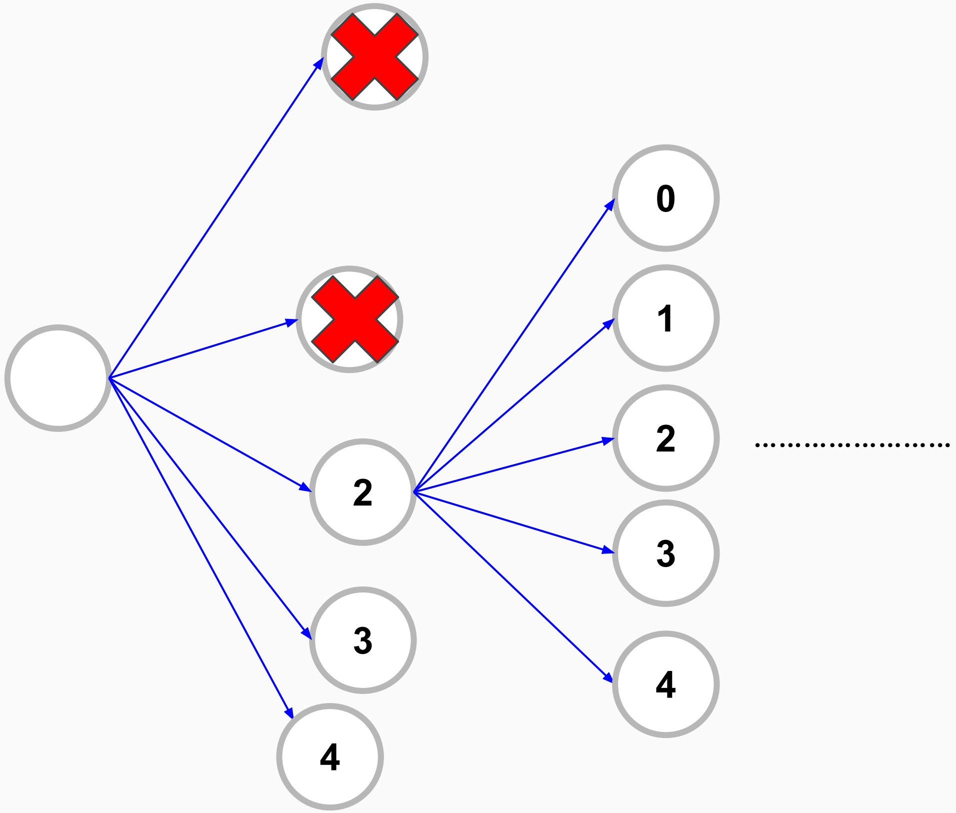
Approach 2 (Time complexity simulation)

For **N = 5**



Approach 2 (Time complexity simulation)

For **N = 5**



Time complexity analysis

- We notice that for each node it makes n other recursive calls. Which means to say that the upper bound definitely $O(n^n)$.
- Since we are solving this by assigning a queen column wise. We note that when we assign the queen in the first column, we have n options after that we have n - 1 options as you cannot place a queen in the same column as the first queen, then n - 2, n - 3… In this case the worst case time complexity is $O(n!)$
- However, since we already have m pre-assigned queens, they do not have any option in that case, hence for our algorithm a tighter bound for worst case time complexity is $O((n-m)!)$

Lab 4 Demo