CS 91 USACO

Bronze Division

Unit 2: 1-D Data Structures

LECTURE 10: HEAP, ARRAYHEAP, PRIORITY QUEUE, AND REVIEW OF SOLUTION SPACE

DR. ERIC CHOU

IEEE SENIOR MEMBER

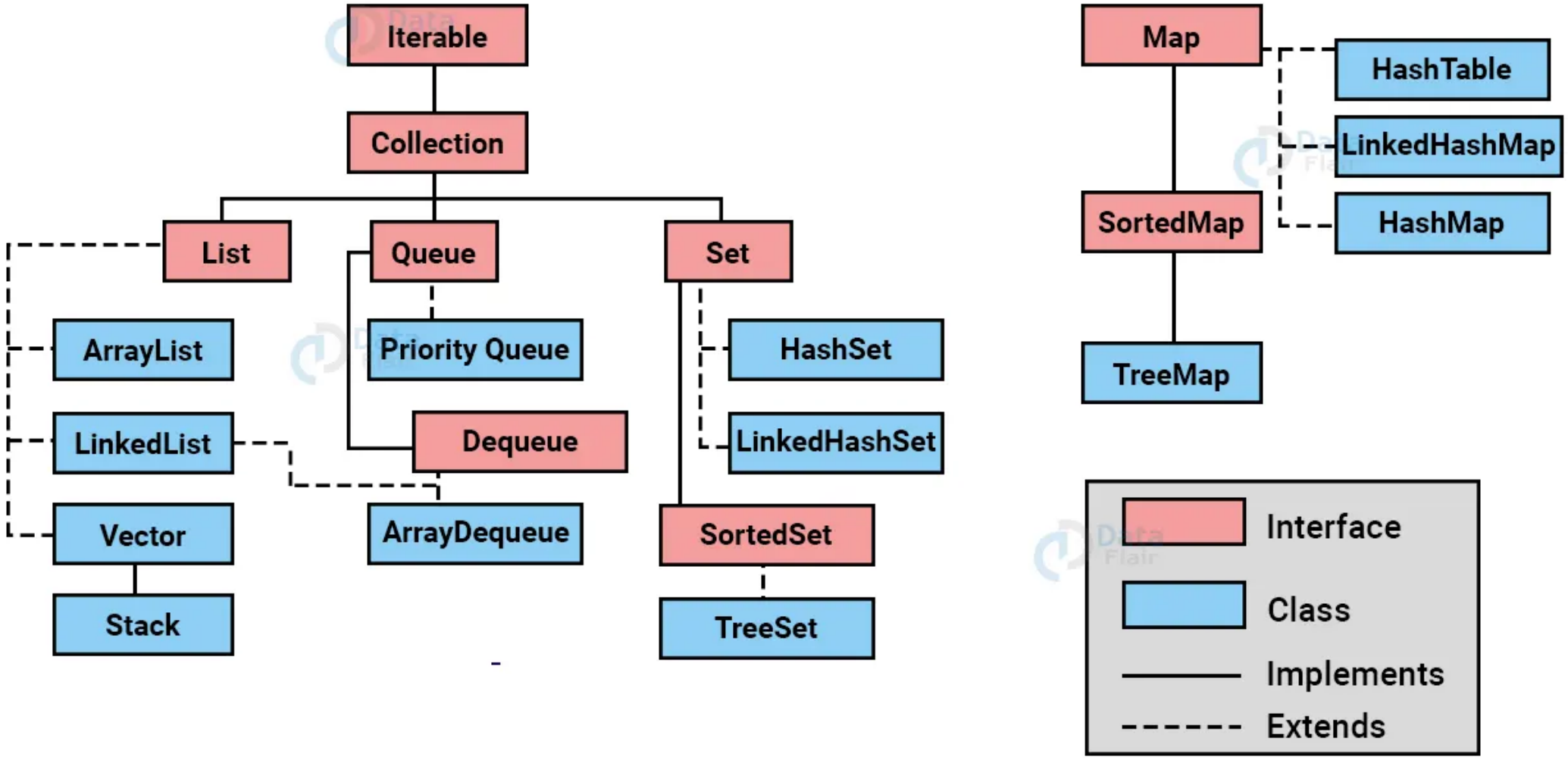
Objectives

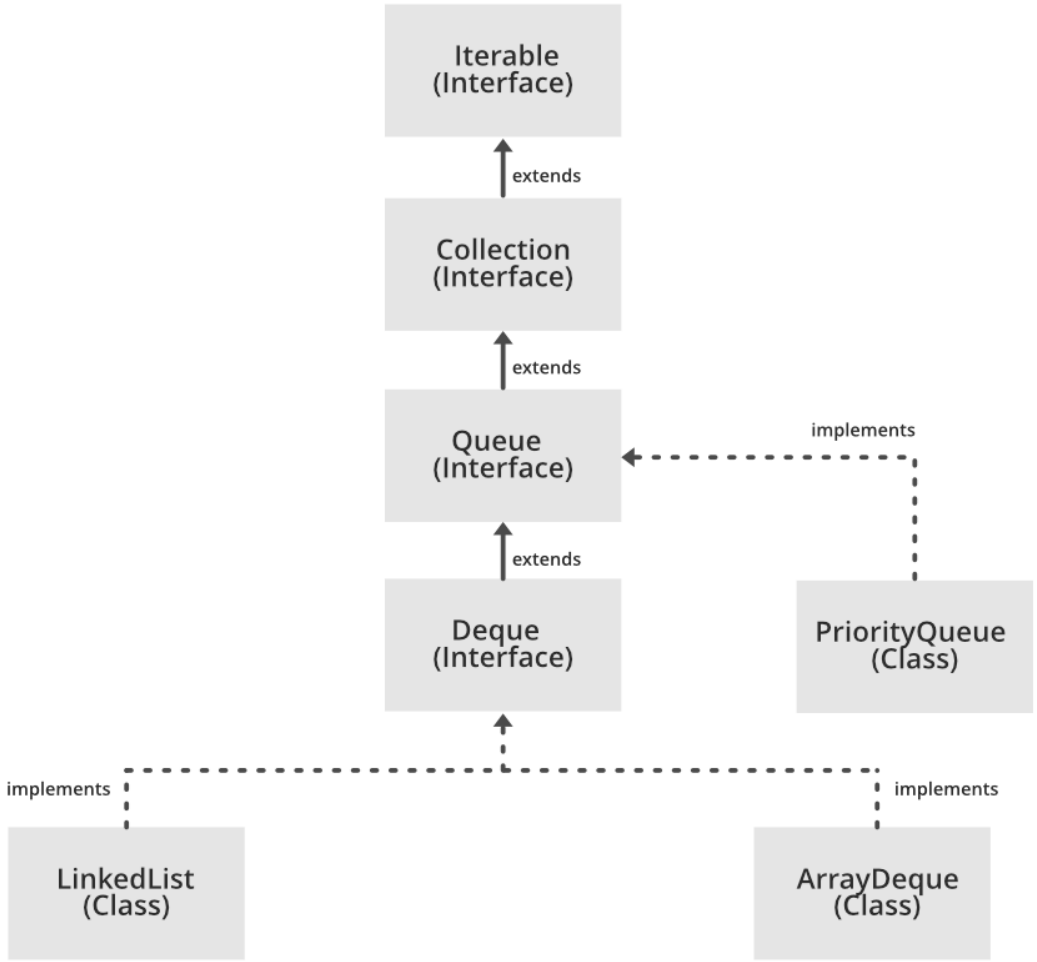
- Priority Queue
- Heap Sort
- ArrayHeap
- Review of Solution Space and Data Structures
- Practice Exam: Milk2 and Transform

Priority Queue

SECTION 1

Hierarchy of Collection Framework in Java



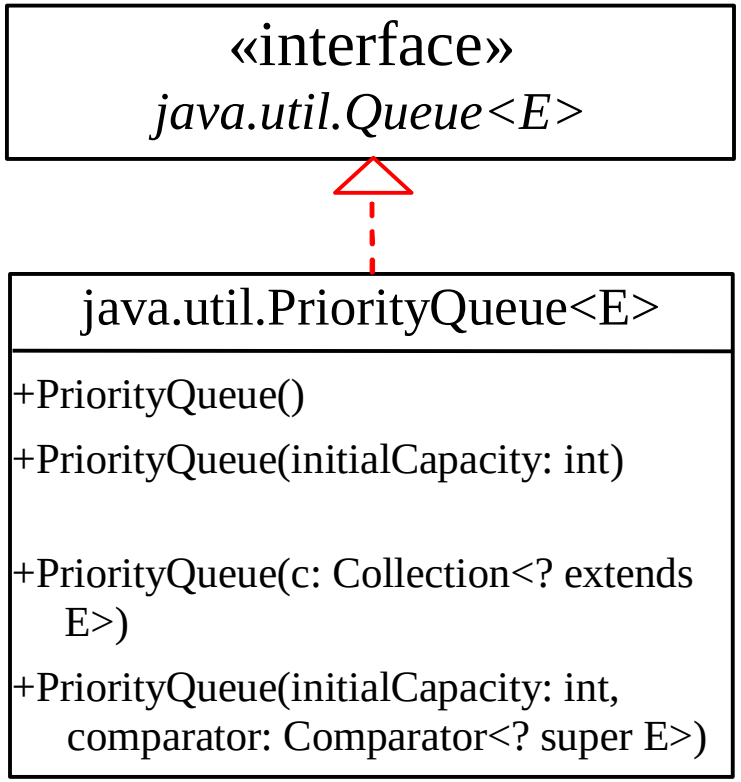

Iterable
(Interface)
extends
Collection
(Interface)
extends
Queue
(Interface)
implements
extends
Deque
(Interface)
PriorityQueue
(Class)
implements
implements
LinkedList
(Class)
ArrayDeque
(Class)

Priority Queue

- A regular queue is a first-in and first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.
- For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.

PriorityQueue UML

PriorityQueue->Heap/Comparable/Queue->LinkedList



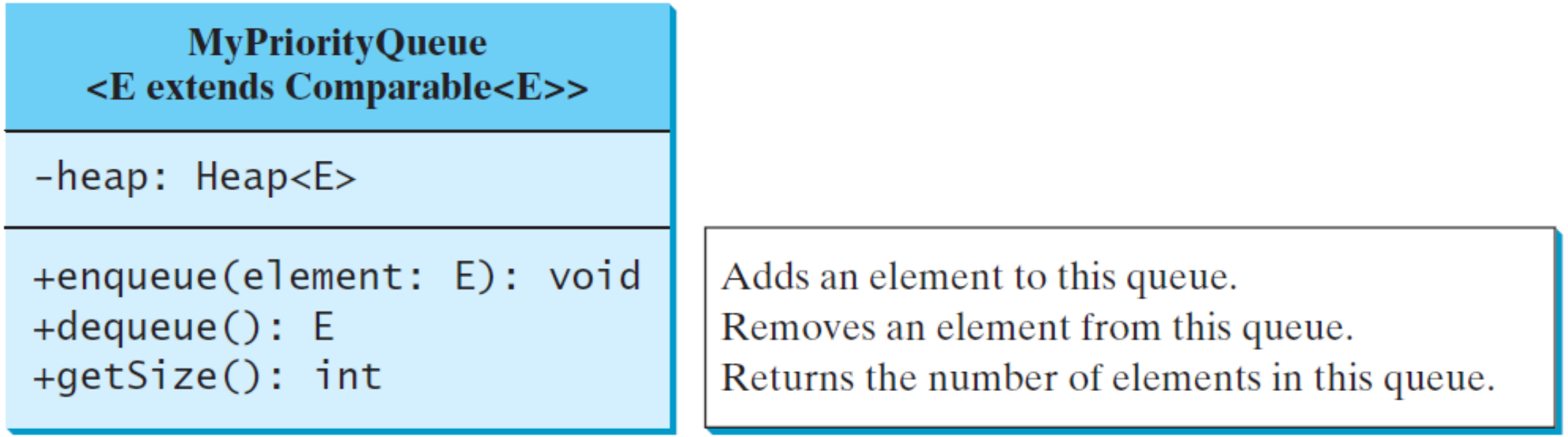
Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

Creates a priority queue with the specified initial capacity and the comparator.

Demo Program:



Good Data Structure for Priority Queue

- Add
- Remove
- Adjust

Priority Queue in JDK

- As a conclusion, in java JDK we have **PriorityQueue**, an unbounded priority queue based on a priority heap.
- The elements of the priority queue are ordered according to their natural ordering, or by a comparator provided at queue construction time, depending on which constructor is used.

Demonstration Program

USEPQ.JAVA

Demonstration Program

USEPQSORT.JAVA

Heap Sort

SECTION 2

HeapSort

- Add all elements from an array (arraylist) into a heap.
- Then, remove all min elements from the heap and add it to a new array (or arraylist).

ArrayHeap

SECTION 3

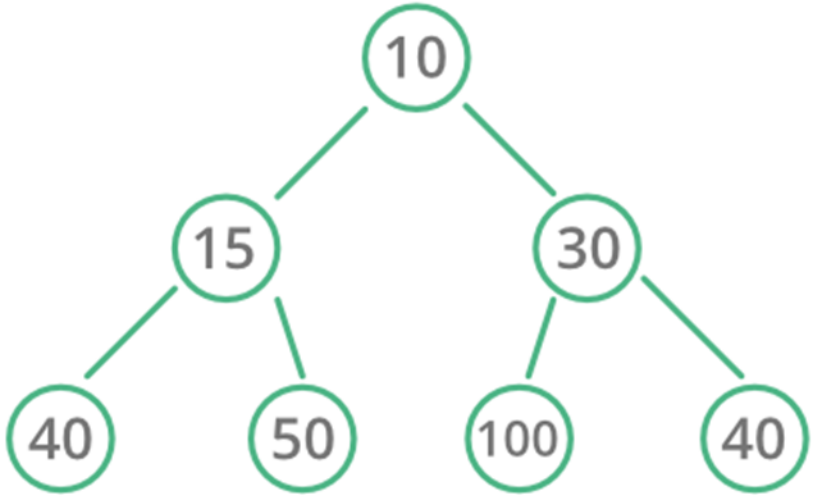
ArrayHeap

- To implement the heaps as arrays:

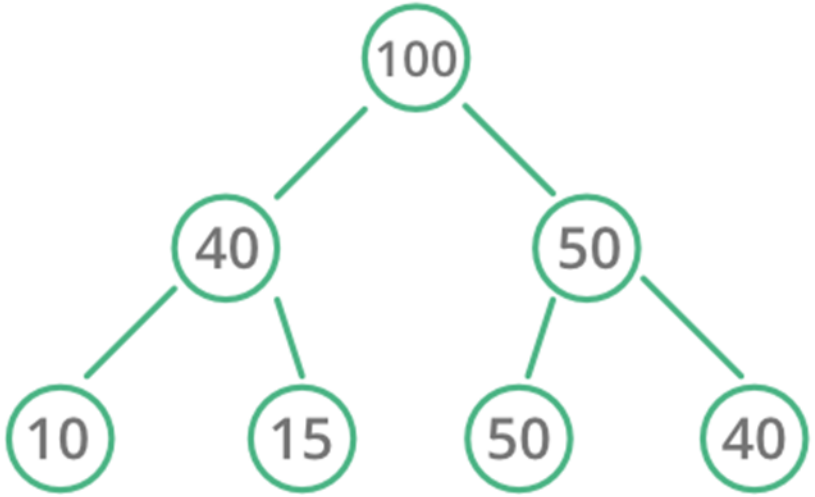
```
We put the root at array[0]
```

- Then we traverse each level from left to right,and so the left child of the root would go into array[1],its right child would be into array[2],etc.
- For the node at array[i], we can get left child using this formula **(2i + 1)**, for the right child we can use this one **(2i + 2)**,and for parent item **floor((i — 1) / 2)**.This works just with complete binary trees.

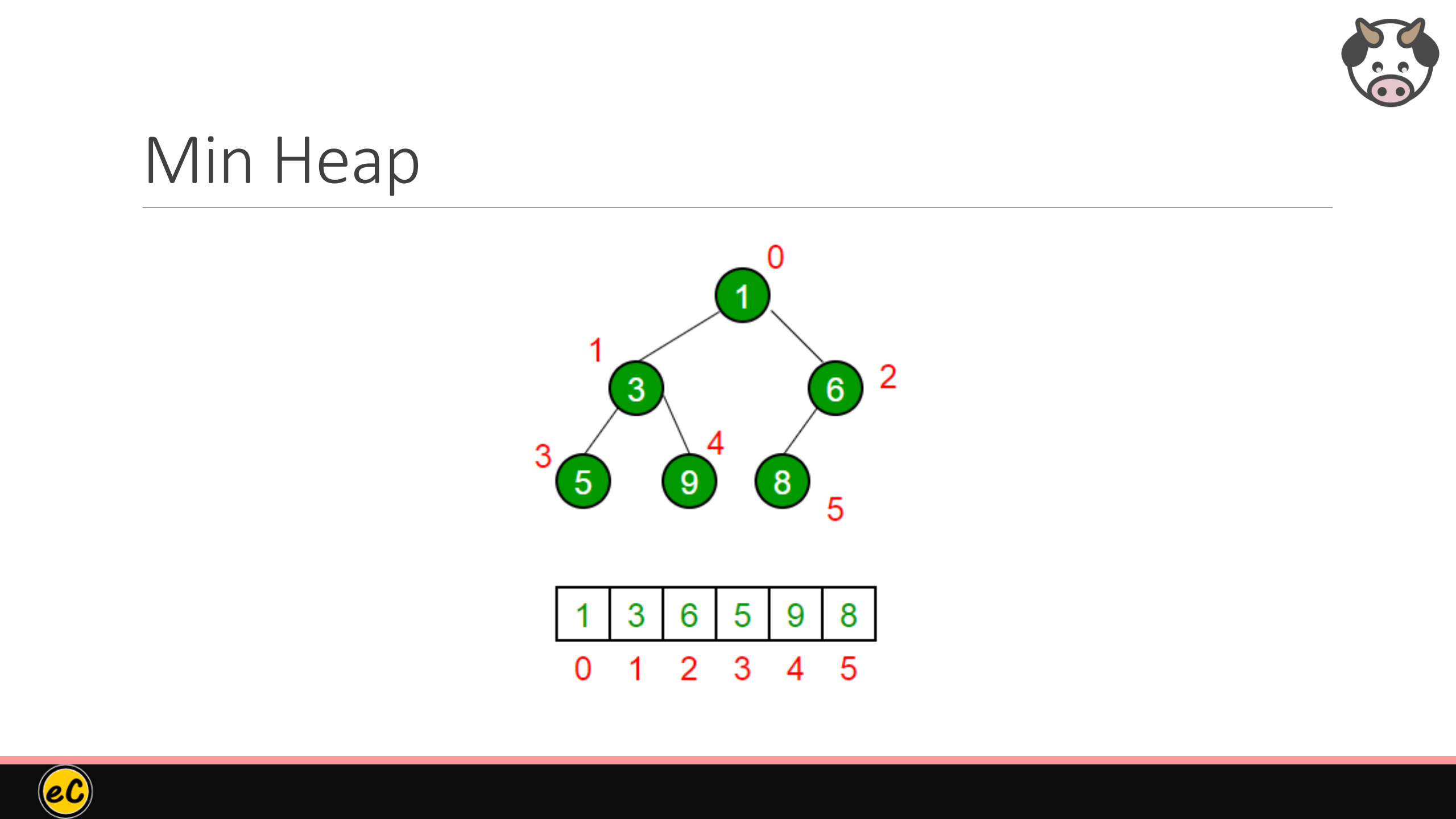
Heap Data Structure



Min Heap



Max Heap



Min Heap

Add and Shift Up

```java
boolean add(int x){
    // Add x at the end of the array list
    arrayHeap.add(x);
    // Sift up
    siftUp();
    return true;
}
```

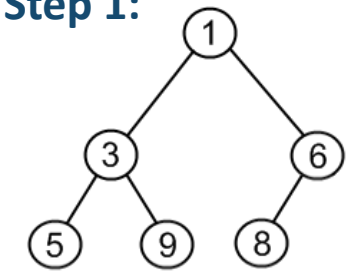
```java
private void siftUp(){
    int p = arrayHeap.size()-1;  // Position to sift up
    while (p != 0){
        int parent = (p-1)/2;
        if (get(p) >= get(parent)){
            // We are done
            break;
        }
        else{
            // Do a swap
            Integer temp = arrayHeap.get(parent);
            arrayHeap.set(parent, arrayHeap.get(p));
            arrayHeap.set(p, temp);
            // Move up
            p = parent;
        }
    }
}
```

Remove Min

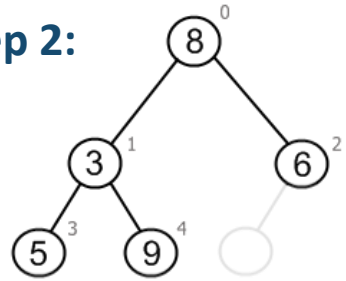
```
public int removeMin(){
    if (isEmpty()){
        String message = "Priority Queue is empty.";
                        throw new RuntimeException(message);
     }
    else{
        int val = arrayHeap.get(0);
        // Replace root by last leaf
        int lastPos = arrayHeap.size()-1;
        arrayHeap.set(0, arrayHeap.get(lastPos));
        // Remove the last leaf
        arrayHeap.remove(arrayHeap.size()-1);
        siftDown();
        return val;
    }
}
```

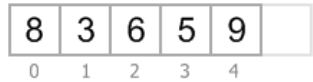
```java
private void siftDown(){
    int p = 0; // Position to sift down
    int size = arrayHeap.size();
    while (2*p + 1 < size){
        int leftChildPos = 2*p + 1;
        int rightChildPos = leftChildPos + 1;
        int minChildPos = leftChildPos;
        // Is there a right child?
        if (rightChildPos < size){
            // Which child is smaller
            if (get(rightChildPos) < get(leftChildPos)){
                minChildPos = rightChildPos;
            }
        }
        // If less than children we are done,
        // otherwise swap node with smaller child
        if (get(p) <= get(minChildPos))
            break;
        else{
            // Do the swap
            Integer temp = arrayHeap.get(p);
            arrayHeap.set(p, arrayHeap.get(minChildPos));
            arrayHeap.set(minChildPos, temp);
        }
        // Go down to the child position
        p = minChildPos;
    }
}
```

Step 1:

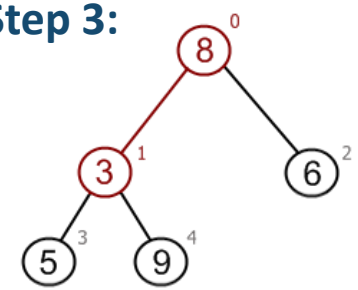


Step 2:

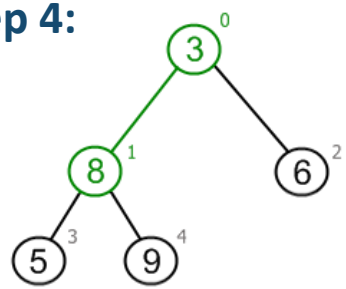




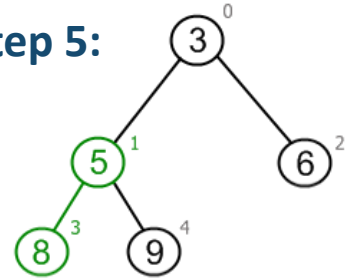
Step 3:



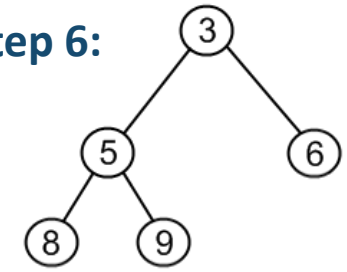
Step 4:



Step 5:



Step 6:



Heap as Priority Queue

- Insertion Time O(log n)
- Deletion Time O(log n)
- Access Time O(1)
- Heap is an array. Heap is also a priority queue. Heap is useful for sorting.

Demonstration Program

USEPMAXHEAP.JAVA

Demonstration Program

USEPPOINTPQ.JAVA + POINT.JAVA

Review of Complete Search

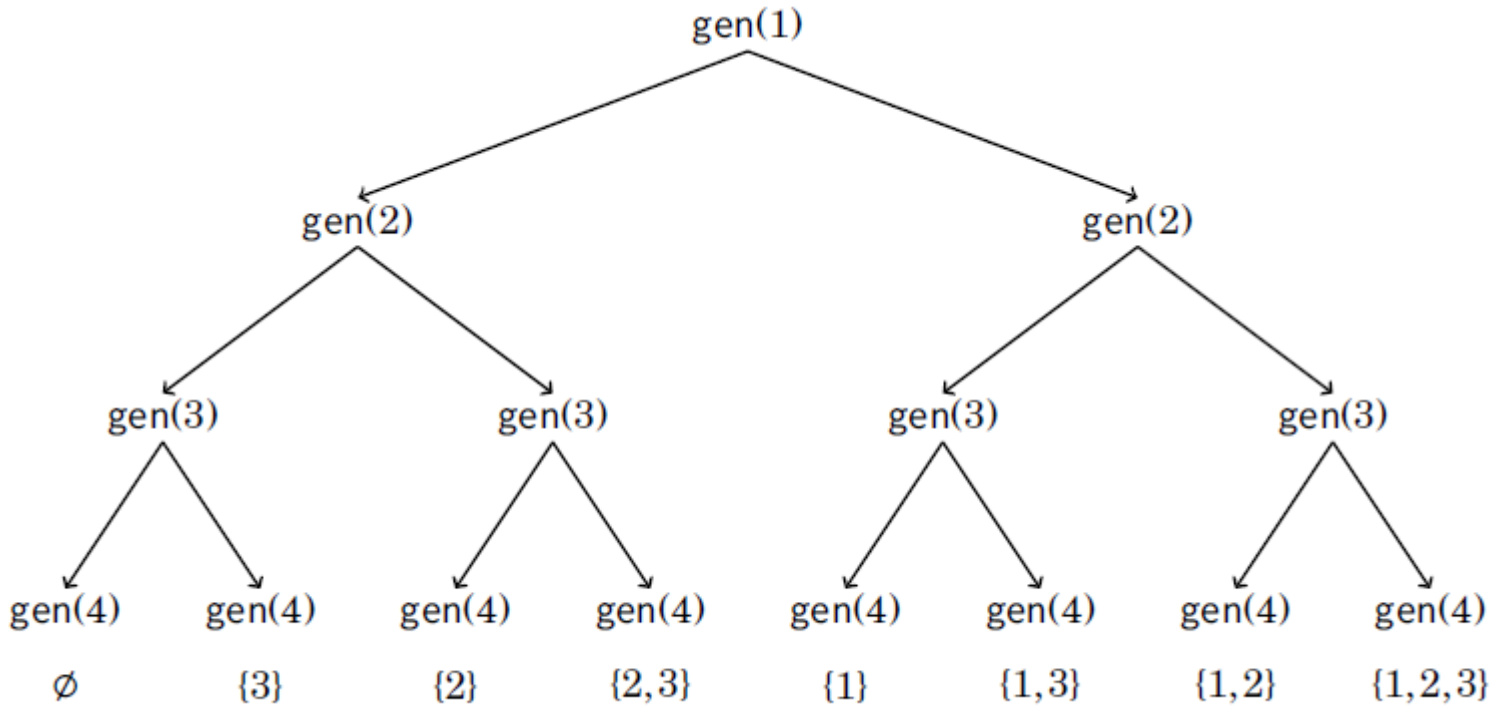
SECTION 4

Techniques for Complete Search

- Generating Solution space:
  - Subset,
  - Combination,
  - Permutation,
  - Pairing,
  - Grouping
- Backtracking

Generating subsets

Method1: Binomial Theorem



Back Tracking

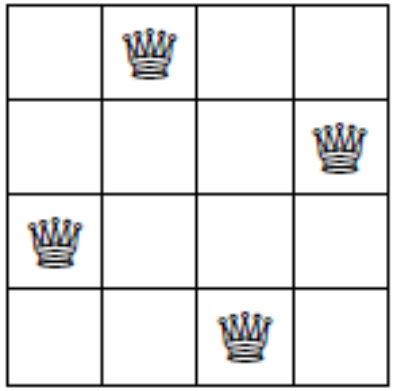
SECTION 5

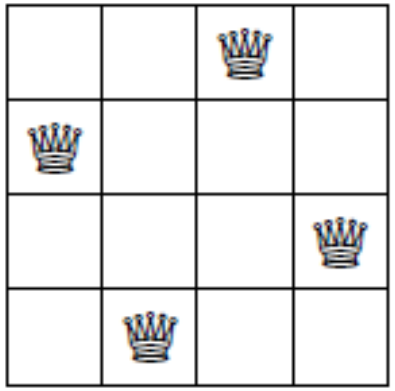
Backtracking

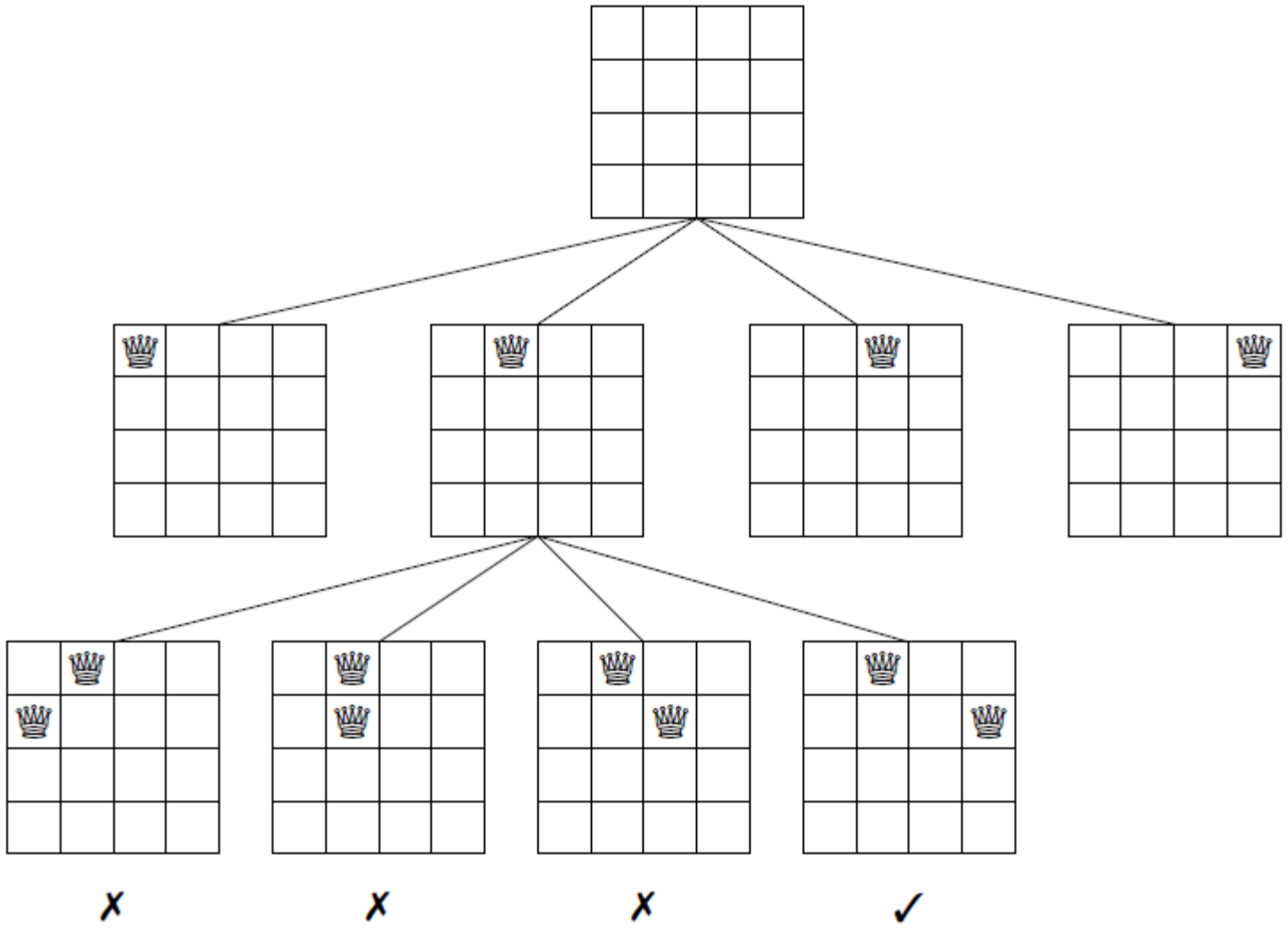
- A backtracking algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.
- Main Idea: Explore all the possible legal moves.

Queen's Problem

As an example, consider the **queen problem** where the task is to calculate the number of ways we can place n queens to an n x n chessboard so that no two queens attack each other. For example, when n = 4, there are two possible solutions to the problem:







```
void search(int y) {
    if (y == n) {
        c++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (r1[x] || r2[x+y] || r3[x-y+n-1]) continue;
        r1[x] = r2[x+y] = r3[x-y+n-1] = 1;
        search(y+1);
        r1[x] = r2[x+y] = r3[x-y+n-1] = 0;
    }
}
```

The code assumes that the rows and columns of the board are numbered from 0. The function places a queen to row y where $0 \le y < n$. Finally, if $y = n$, a solution has been found and the variable c is increased by one.

The array `r1` keeps track of the columns that already contain a queen, and the arrays `r2` and `r3` keep track of the diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, the rows and diagonals of the 4×4 board are numbered as follows:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

`r1`

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

`r2`

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

`r3`

Let $q(n)$ denote the number of ways to place n queens to an $n \times n$ chessboard. The above backtracking algorithm tells us that, for example, $q(8) = 92$. When n increases, the search quickly becomes slow, because the number of the solutions increases exponentially. For example, calculating $q(16) = 14772512$ using the above algorithm already takes about a minute on a modern computer[1].

Practice: Milking Cows (milk2)

SECTION 6

Problem Statement

- Three farmers rise at 5 am each morning and head for the barn to milk three cows. The first farmer begins milking his cow at time 300 (measured in seconds after 5 am) and ends at time 1000. The second farmer begins at time 700 and ends at time 1200. The third farmer begins at time 1500 and ends at time 2100. The longest continuous time during which at least one farmer was milking a cow was 900 seconds (from 300 to 1200). The longest time no milking was done, between the beginning and the ending of all milking, was 300 seconds (1500 minus 1200).

Problem Statement

- Your job is to write a program that will examine a list of beginning and ending times for N (1 <= N <= 5000) farmers milking N cows and compute (in seconds):
  - The longest time interval at least one cow was milked.
  - The longest time interval (after milking starts) during which no cows were being milked.
- **Note:** Milking from time 1 through 10, then from time 11 through 20 counts as two different time intervals.

INPUT FORMAT (file milk2.in):

- **Line 1:** The single integer, N
- **Lines 2..N+1:** Two non-negative integers less than 1,000,000, respectively the starting and ending time in seconds after 0500

SAMPLE INPUT:

```
3
300 1000
700 1200
1500 2100
```

OUTPUT FORMAT (milk2.out):

- A single line with two integers that represent the longest continuous time of milking and the longest idle time.

SAMPLE OUTPUT:

```
900 300
```

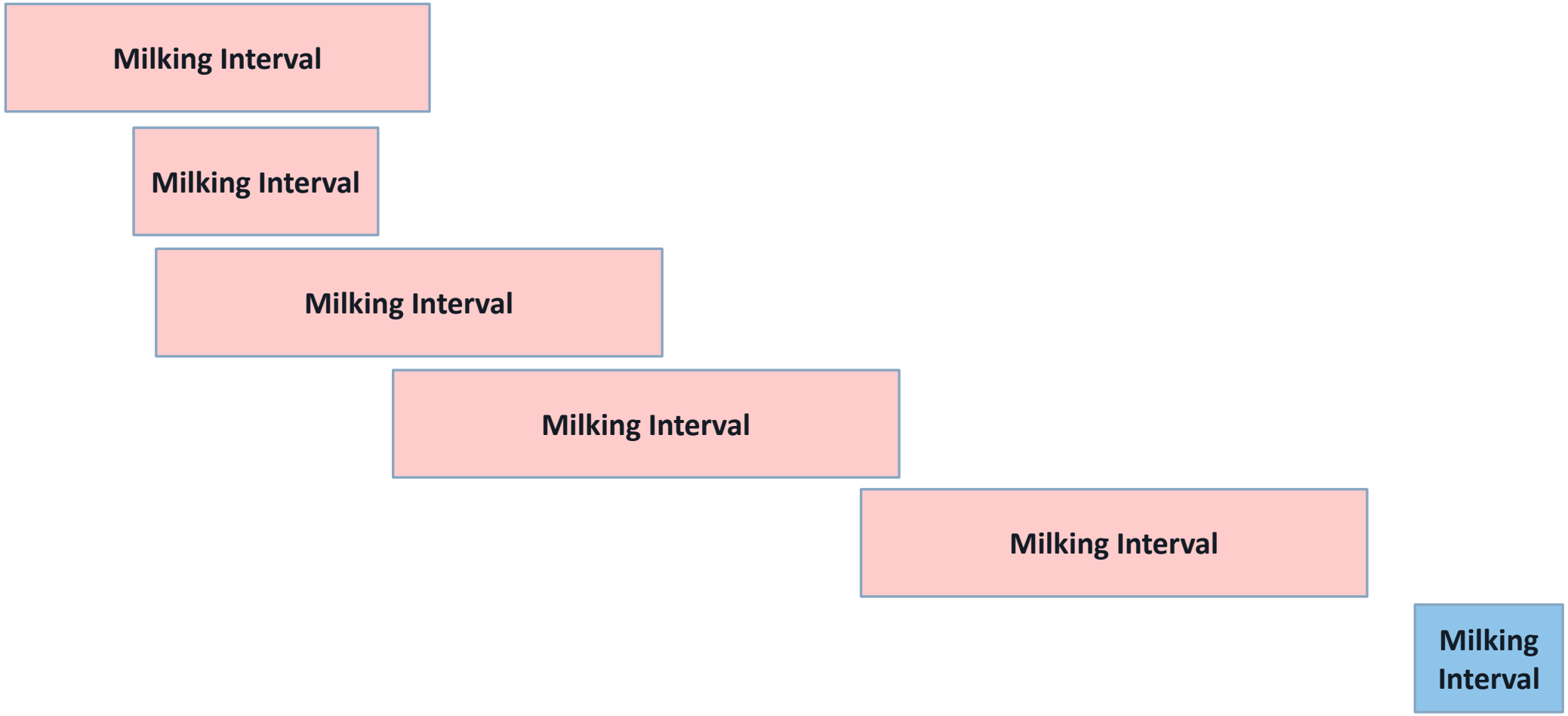
Nature of the Problem

- Linear Solution Space
- Merge of elements

Nature of the Problem

- Interval Management
- Add a new milking interval
- Convert all milking intervals into Interval objects and add these objects into an interval list one by one.
- Sort all milking intervals by starting time. (***O(log n))***)
- Merging the milking intervals
- Find the longest interval and the longest bye-interval.

Merging Intervals



Bye-Intervals

Bye Interval	Milking Interval	Bye Interval	Milking Interval	Bye Interval	Milking Interval

Practice: Transformations (transform)

SECTION 7

Problem Statement

- A square pattern of size N x N (1 <= N <= 10) black and white square tiles is transformed into another square pattern. Write a program that will recognize the minimum transformation that has been applied to the original pattern given the following list of possible transformations:

Problem Statement

- #1: 90 Degree Rotation: The pattern was rotated clockwise 90 degrees.
- #2: 180 Degree Rotation: The pattern was rotated clockwise 180 degrees.
- #3: 270 Degree Rotation: The pattern was rotated clockwise 270 degrees.
- #4: Reflection: The pattern was reflected horizontally (turned into a mirror image of itself by reflecting around a vertical line in the middle of the image).
- #5: Combination: The pattern was reflected horizontally and then subjected to one of the rotations (#1-#3).
- #6: No Change: The original pattern was not changed.
- #7: Invalid Transformation: The new pattern was not obtained by any of the above methods.

Problem Statement

- In the case that more than one transform could have been used, choose the one with the minimum number above.

INPUT FORMAT (file transform.in):

- **Line 1:** A single integer, N
- **Line 2..N+1:** N lines of N characters (each either `@' or `-'); this is the square before transformation
- **Line N+2..2*N+1:** N lines of N characters (each either `@' or `-'); this is the square after transformation

SAMPLE INPUT:

```
3
@-@
---
@@-
@-@
@--
--@
```

OUTPUT FORMAT (transform.out):

- A single line containing the number from 1 through 7 (described above) that categorizes the transformation required to change from the `before' representation to the `after' representation.

SAMPLE OUTPUT:

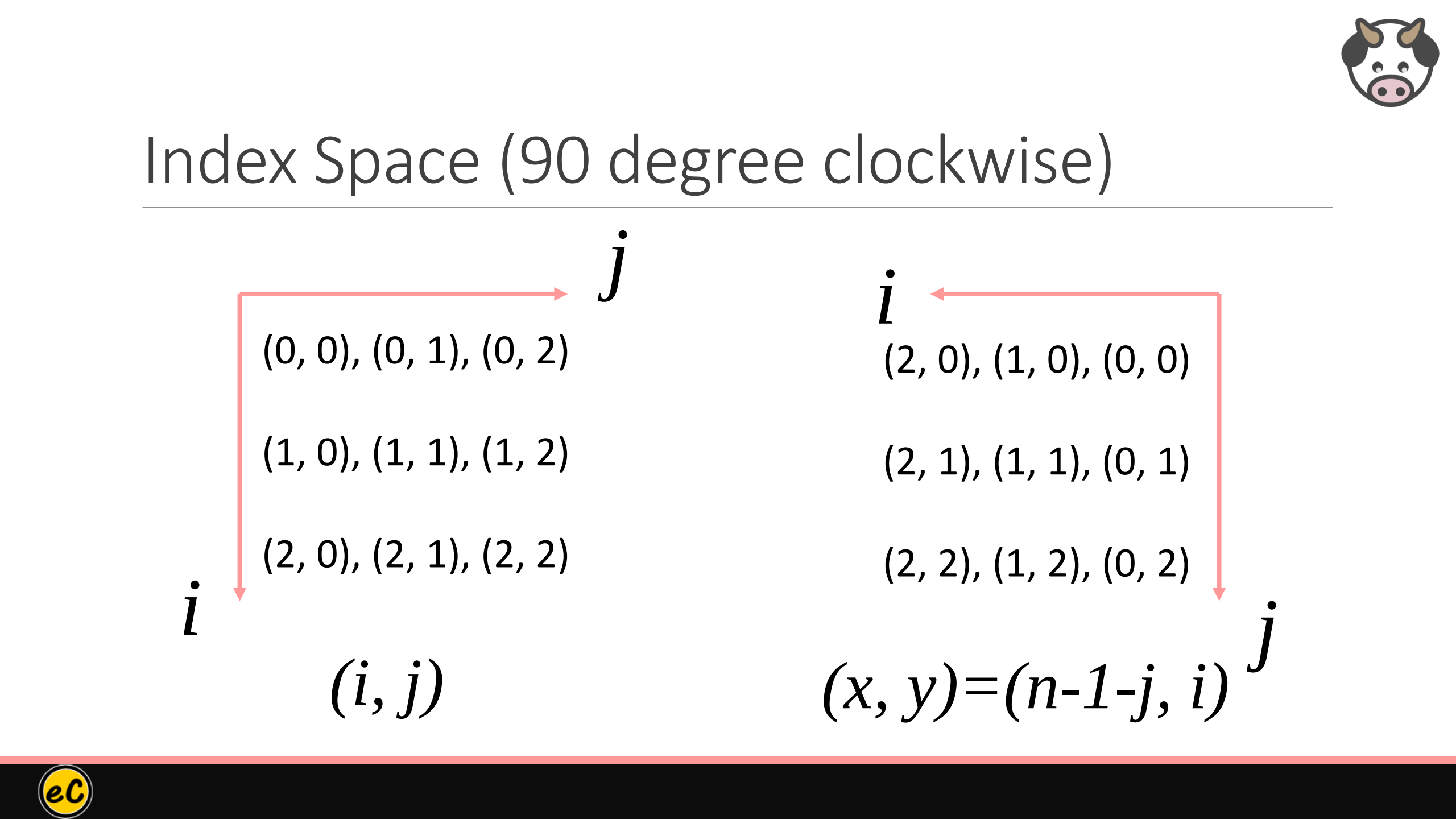
```
1
```

Nature of the Problem

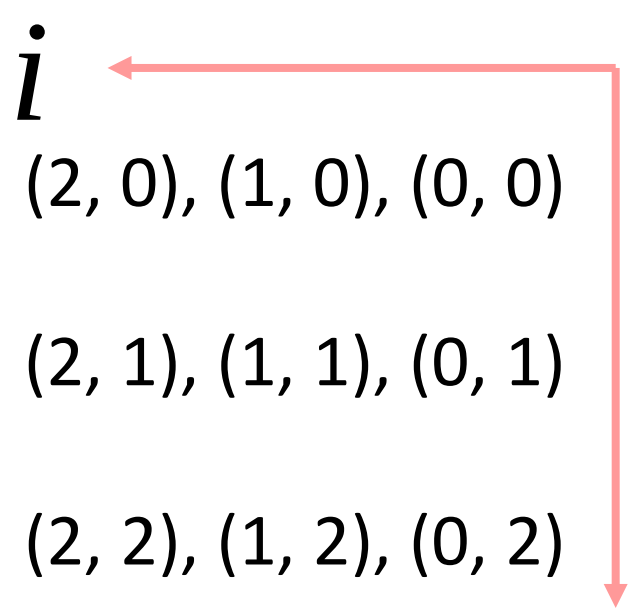
- Try out the possible rule-spaces

Transformation Rules

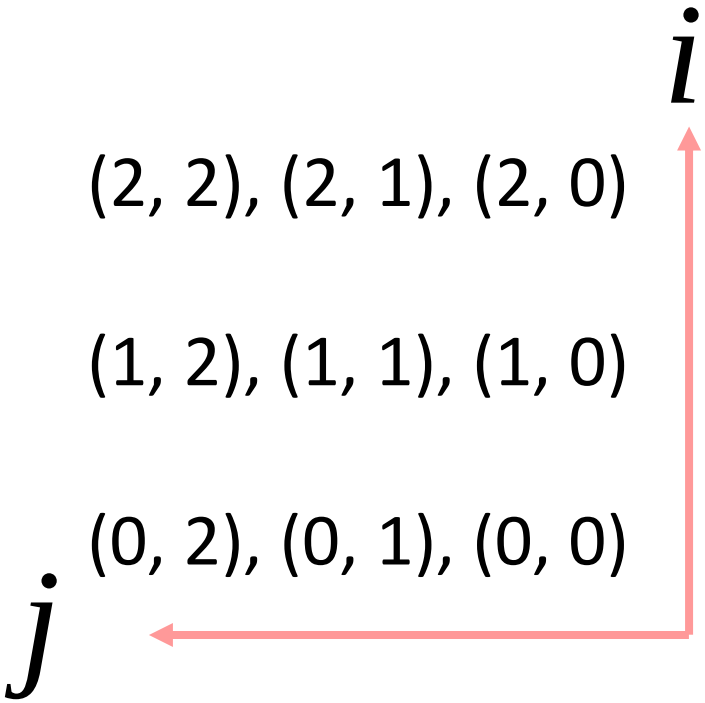
- Rule 1: 90 Degree Rotation: The pattern was rotated clockwise 90 degrees.
- Rule 2: 180 Degree Rotation: The pattern was rotated clockwise 180 degrees.
- Rule 3: 270 Degree Rotation: The pattern was rotated clockwise 270 degrees.
- Rule 4: Reflection: The pattern was reflected horizontally (turned into a mirror image of itself by reflecting around a vertical line in the middle of the image).
- Rule 5: Combination: The pattern was reflected horizontally and then subjected to one of the rotations (#1-#3).
- Rule 6: No Change: The original pattern was not changed.
- Rule 7: Invalid Transformation: The new pattern was not obtained by any of the above methods.


Index Space (90 degree clockwise)
j
i
(0, 0), (0, 1), (0, 2)
(1, 0), (1, 1), (1, 2)
(2, 0), (2, 1), (2, 2)
(i, j)
i
(2, 0), (1, 0), (0, 0)
(2, 1), (1, 1), (0, 1)
(2, 2), (1, 2), (0, 2)
j
(x, y)=(n-1-j, i)

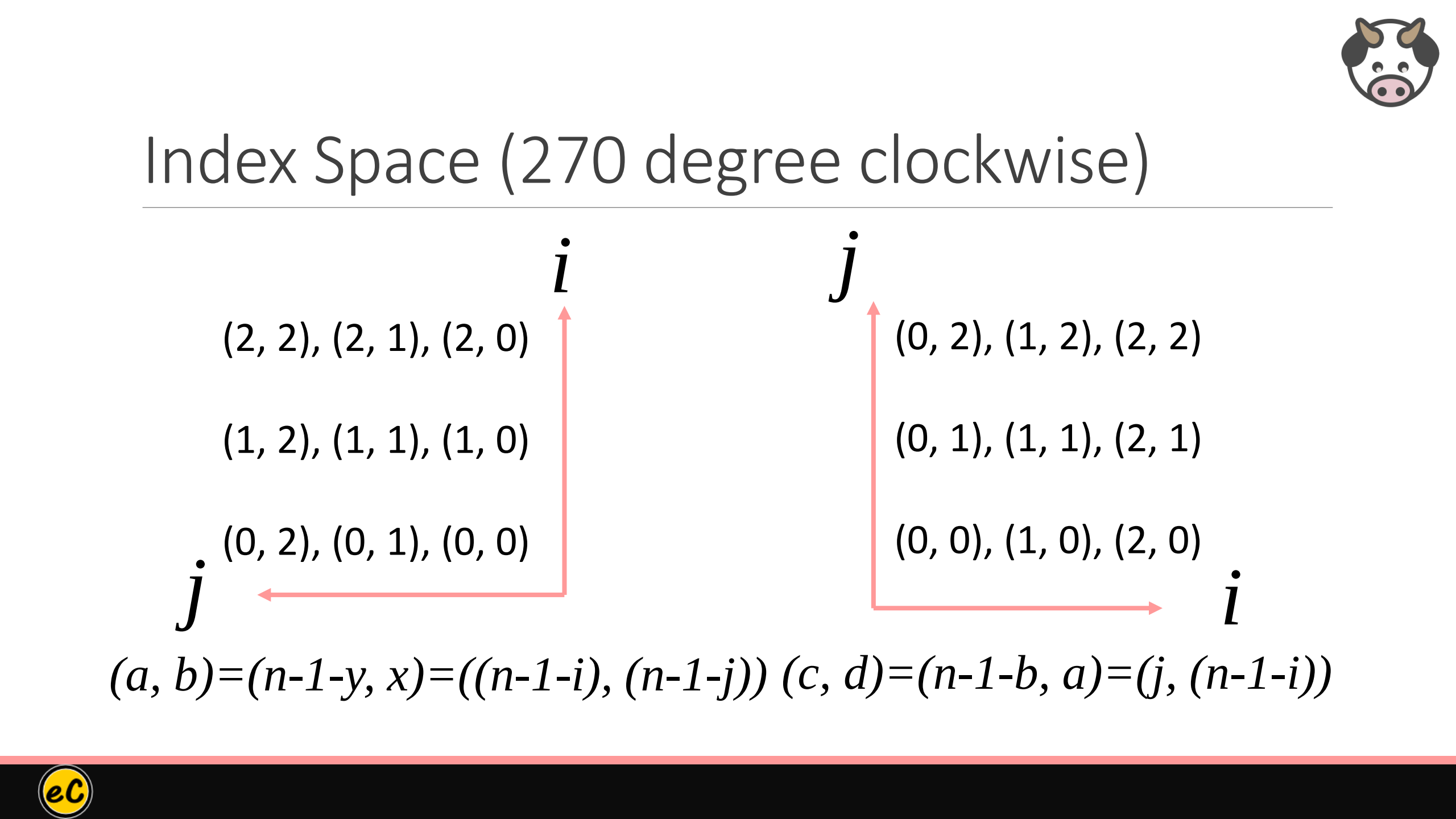
Index Space (180 degree clockwise)



$(x, y)=(n\text{-}1\text{-}j, i)$

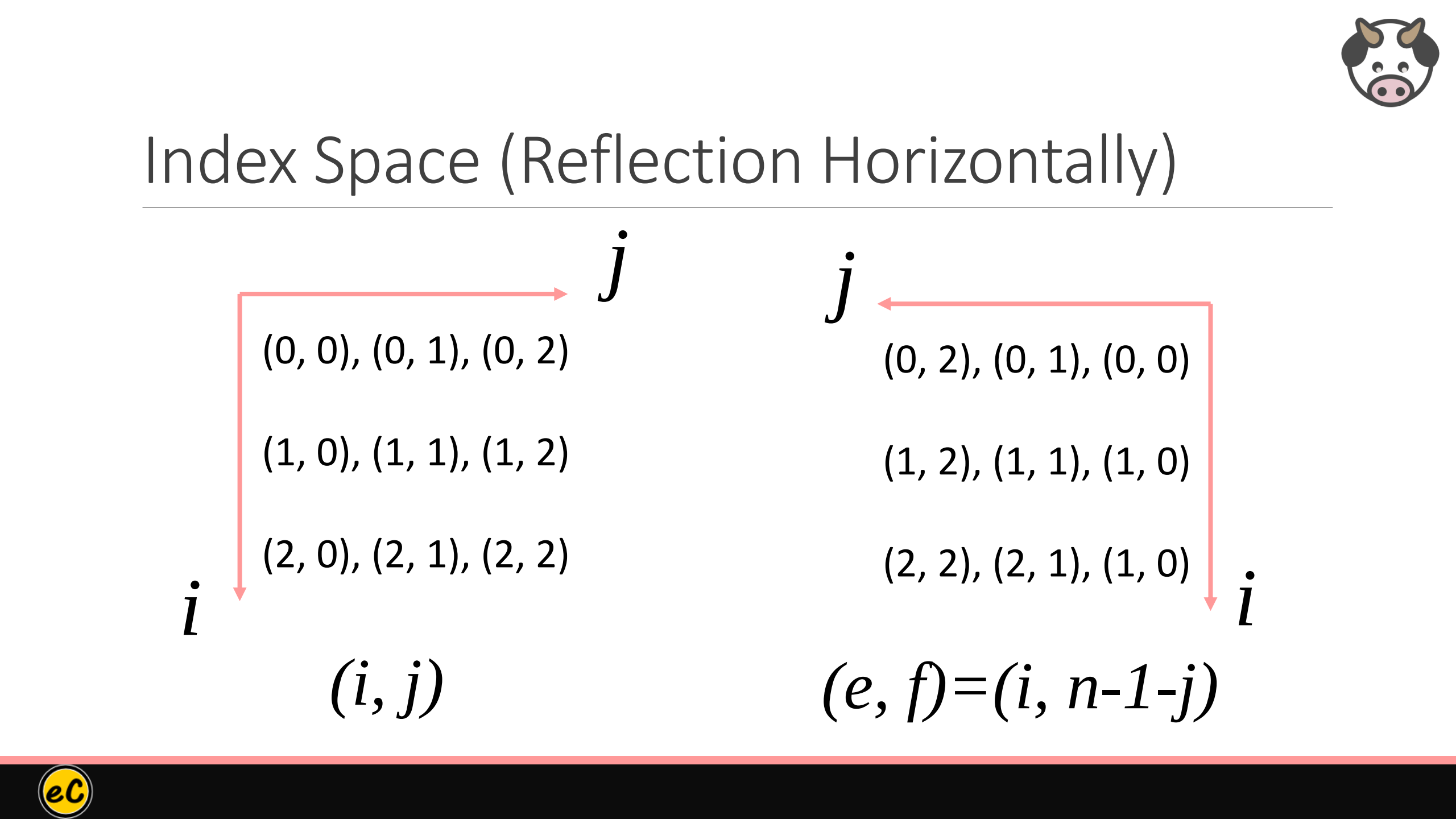


$(a, b)=(n\text{-}1\text{-}y, x)=((n\text{-}1\text{-}i), (n\text{-}1\text{-}j))$



Index Space (270 degree clockwise)

$(a, b)=(n\text{-}1\text{-}y, x)=((n\text{-}1\text{-}i), (n\text{-}1\text{-}j))$ $(c, d)=(n\text{-}1\text{-}b, a)=(j, (n\text{-}1\text{-}i))$



Index Space (Reflection Horizontally)

j

(0, 0), (0, 1), (0, 2)

(1, 0), (1, 1), (1, 2)

(2, 0), (2, 1), (2, 2)

i

(i, j)

j

(0, 2), (0, 1), (0, 0)

(1, 2), (1, 1), (1, 0)

(2, 2), (2, 1), (1, 0)

i

$(e, f)=(i, n\text{-}1\text{-}j)$

Rule 5

- Apply the horizontal flection and then, apply rotate once to check if valid, then twice, then third times. If there is a match within 3 times. Then return turn. Otherwise return false.

Nature of the Problem

- Use simple index generator for clockwise 90 degree and horizontal reflection.
- Rule checking rules 1, 2, 3, 4, 5, and 6 one by one for isValid() checking. If none of these rules matches, return 7.