

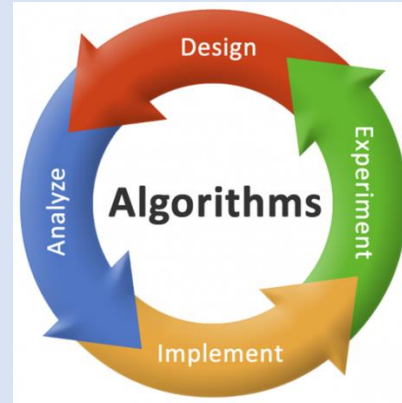
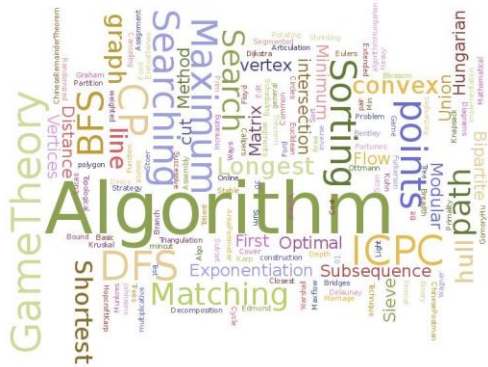
Backtracking

COP 3503

Spring 2025

Department of Computer Science
University of Central Florida

Most slides are by Dr. Steinberg



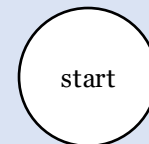
Introduction

- We just observed Divide and Conquer (DC) in designing an efficient running time algorithm.
- DC yields the correct results to problems.
- There can be more than one correct result, however one may be better than the other.
- In this lecture we will discuss another advanced technique called Backtracking

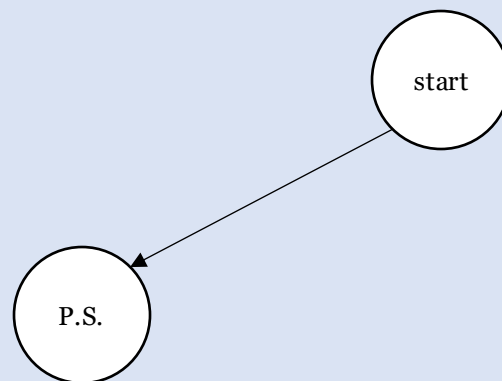
What kind of problems does Backtracking Solve?

- Combinatorial Problems
- Puzzles
- Finding Feasible Solutions
 - Finding the Optimal Solution

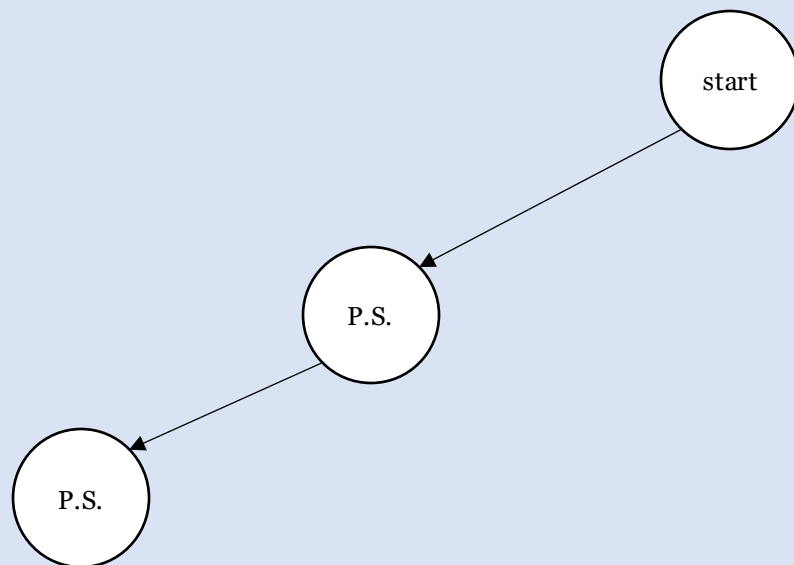
The Big Picture of Backtracking



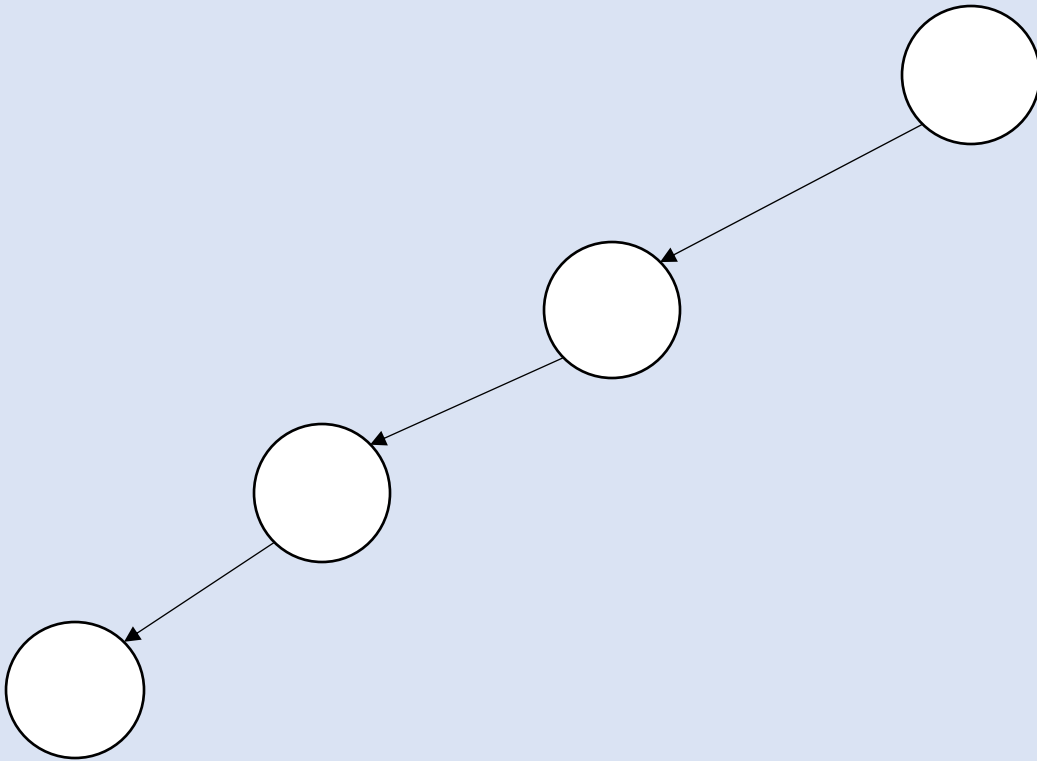
The Big Picture of Backtracking



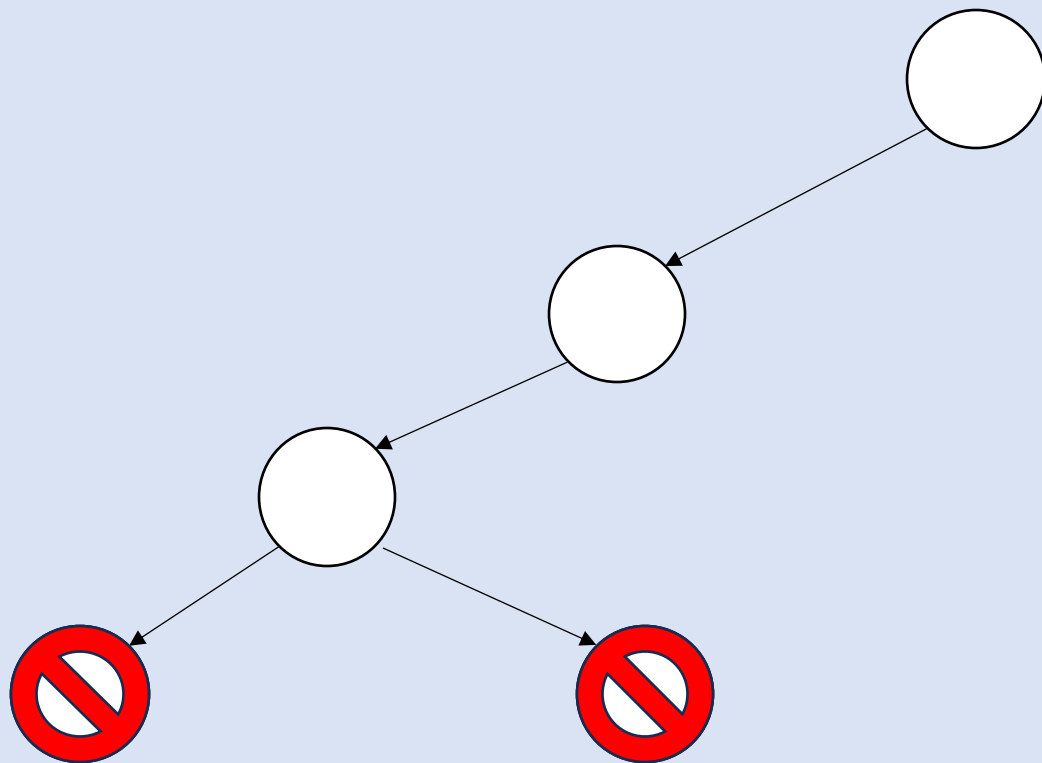
The Big Picture of Backtracking



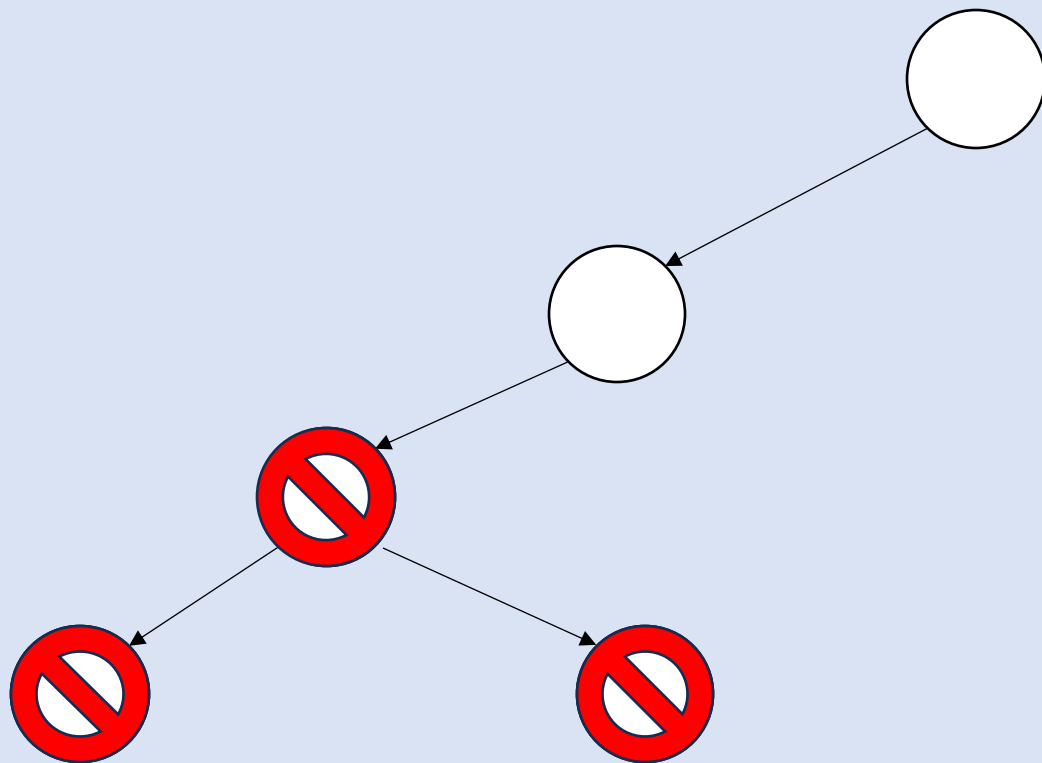
Quick Review of Tree Terminology



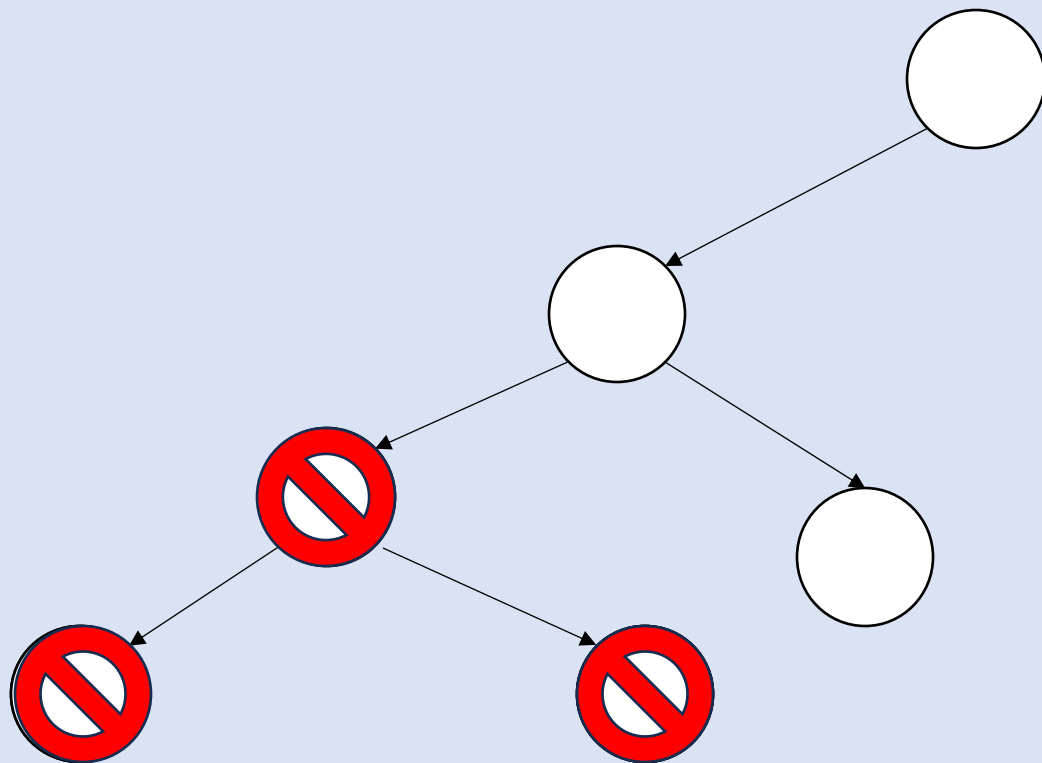
The Big Picture of Backtracking



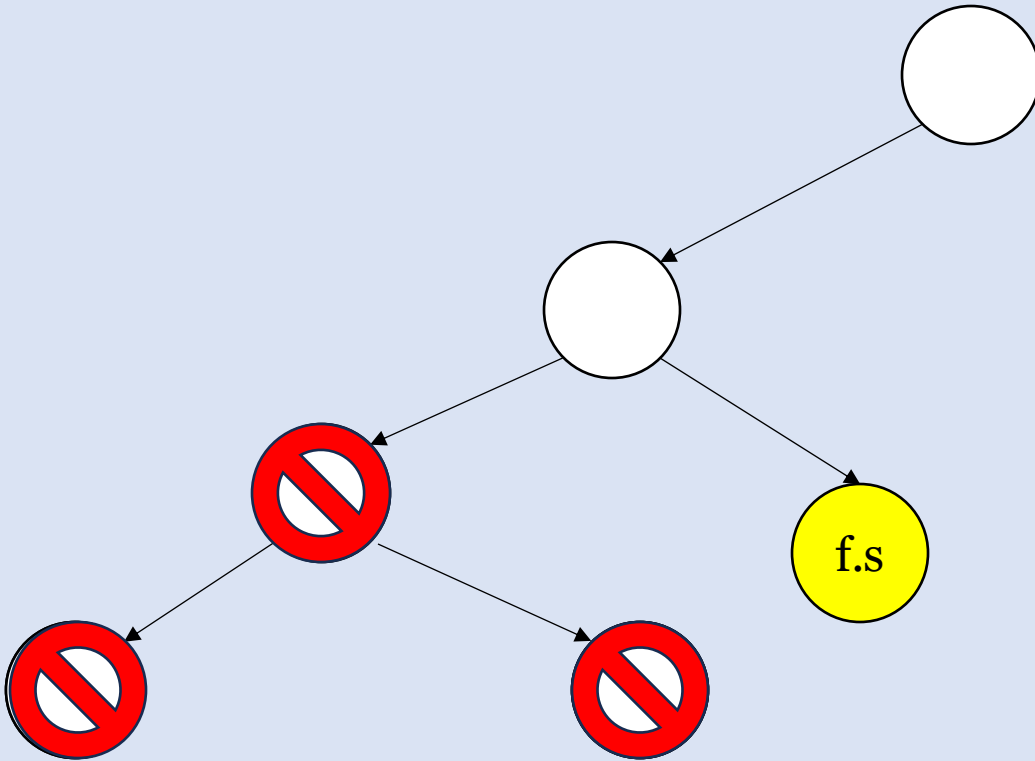
The Big Picture of Backtracking



The Big Picture of Backtracking



The Big Picture of Backtracking



Backtracking

- Uses a Search Tree
- The Root is the starting state before the search for solutions
- Nodes on the first level, choices made for the first component of the solution
- Nodes on the second level, choices for second component of the solution
- The pattern continues...
- Each node on the tree is considered a potential solution, if it corresponds to a partially constructed solution that may still lead to a full solution.
- Leaves in the tree are dead ends or complete solution.

The Search Tree Construction

- If the current node is considered a potential solution, its child is generated by adding the first remaining legitimate option for the next component of a solution
 - The algorithm moves onto the child
- If the current node is NOT considered a potential solution, then the algorithm “backtracks” to the parent to consider the next potential solution.
 - If no option is possible, then the algorithm backtracks up on more level in the search tree.
- If the potential solution turns out to be the complete solution, then the algorithm stops (assuming we are searching for one solution).

Based on observation of Backtracking what kind of problems can we apply this technique too?
From the description, are there limitations?

Let's Observe the Classic N-Queens Problem

Problem Definition

The n -queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal.

Problem Definition

The n -queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal.

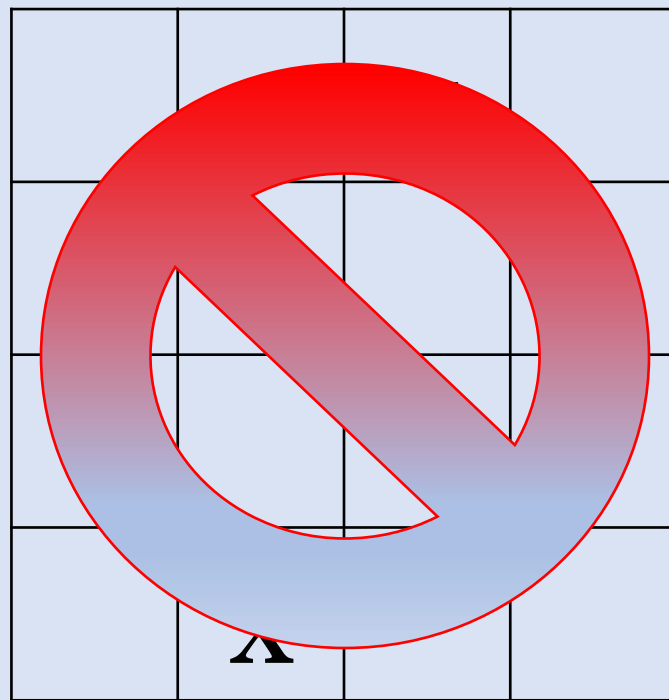
Sample
4 queens on 4 x 4

		X	
X			
			X
	X		

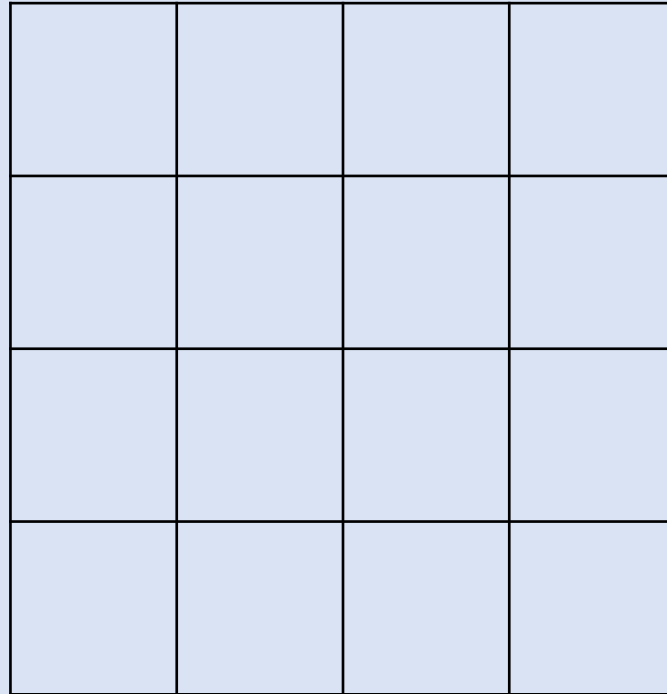
Problem Definition

		X	
X			X
	X		

Problem Definition



Search Tree Example of 4 queens on 4 x 4





x			



X			



X	X		

x			

x			
	x		

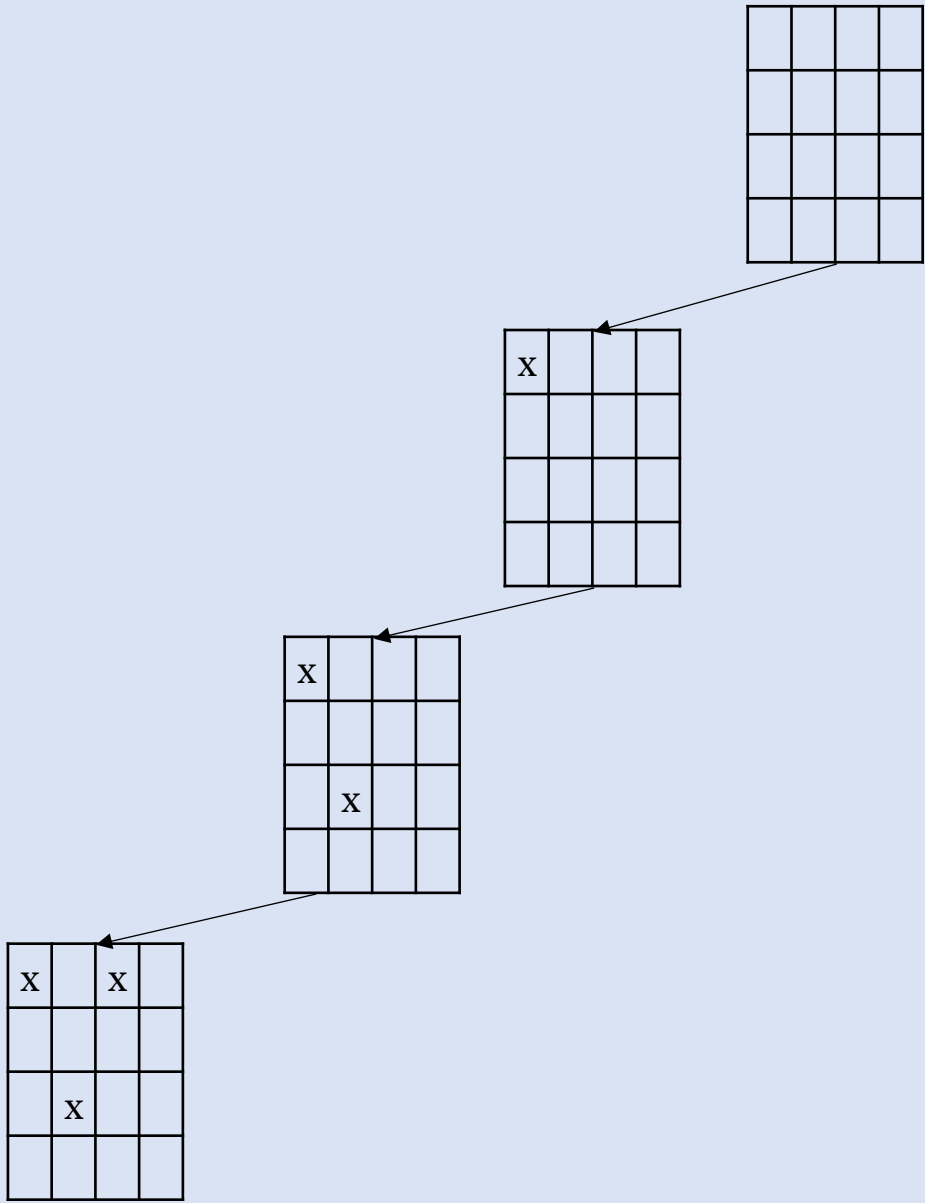


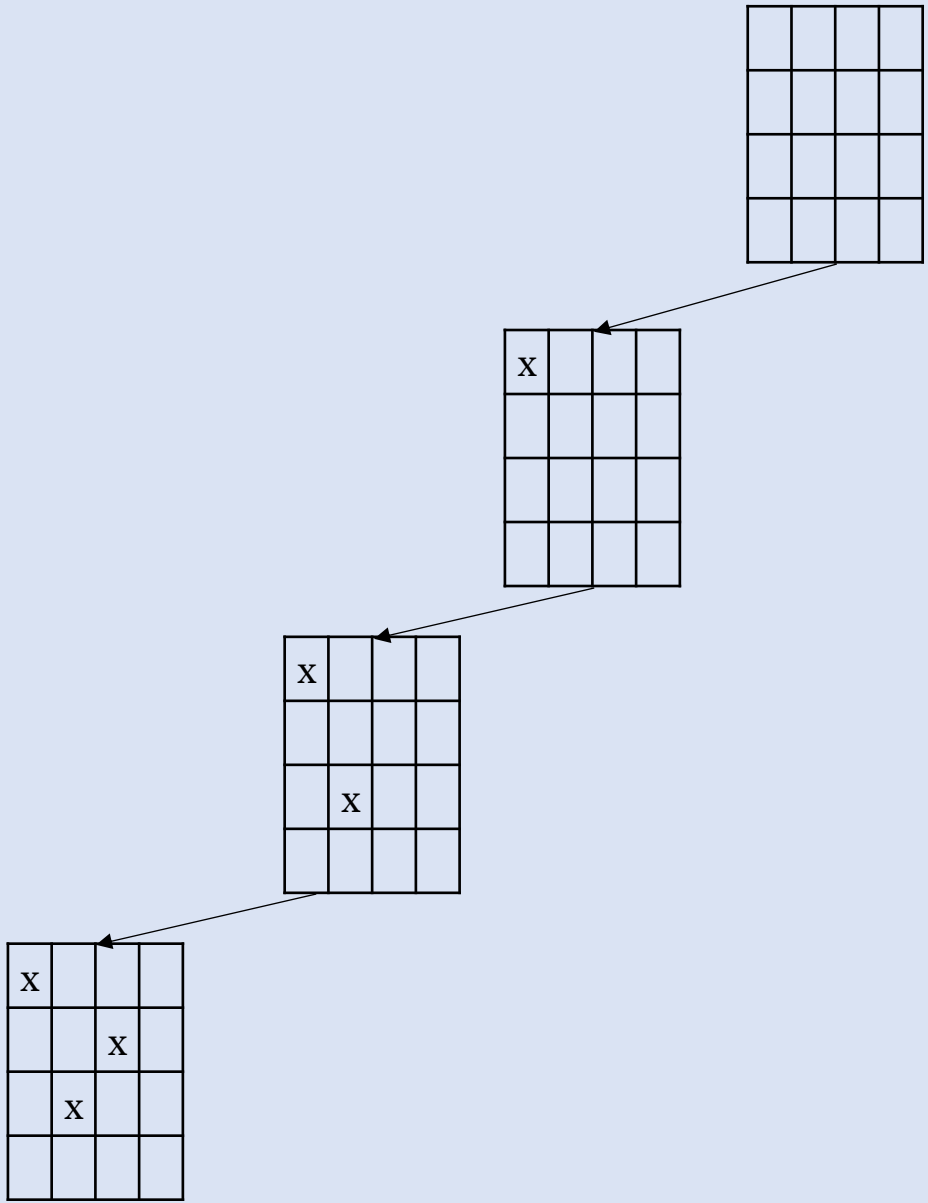


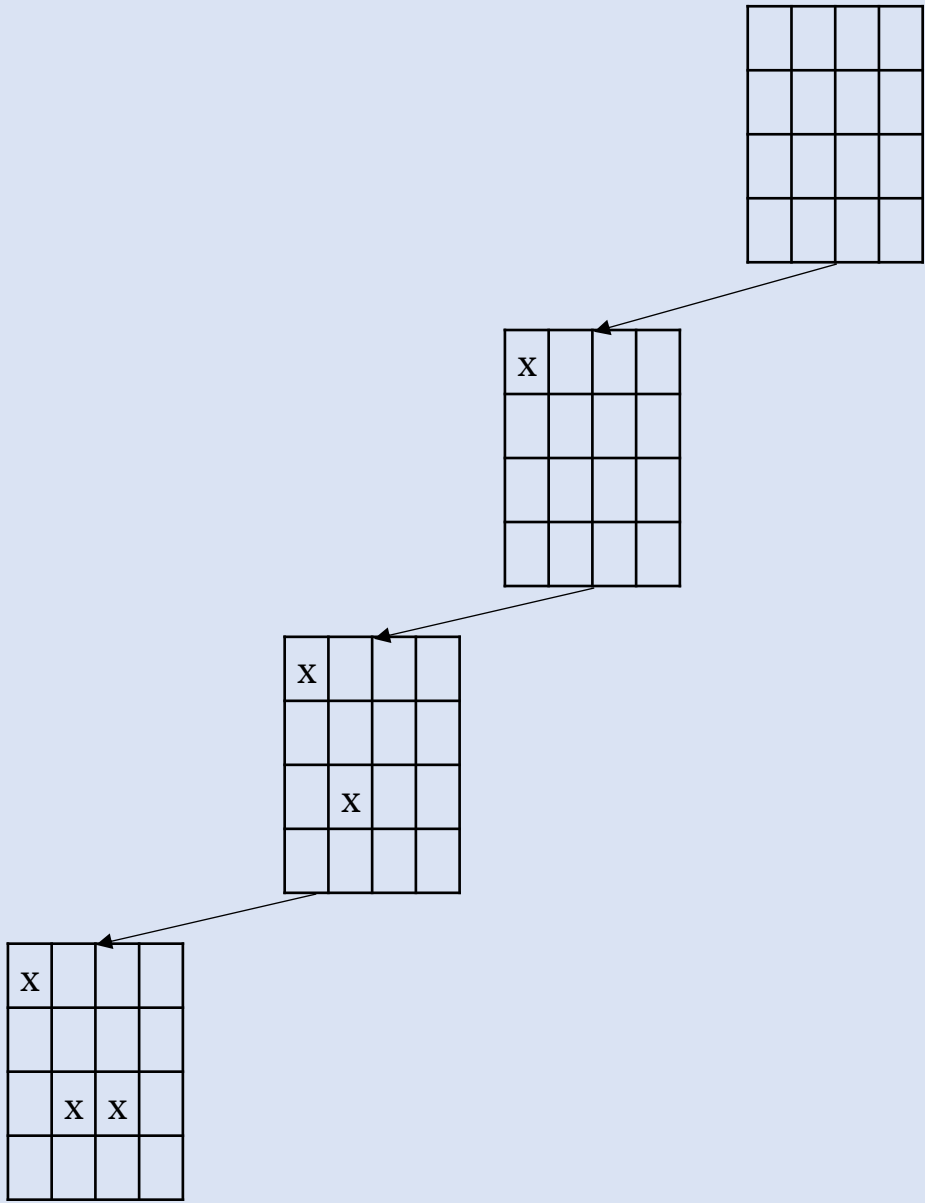
X			

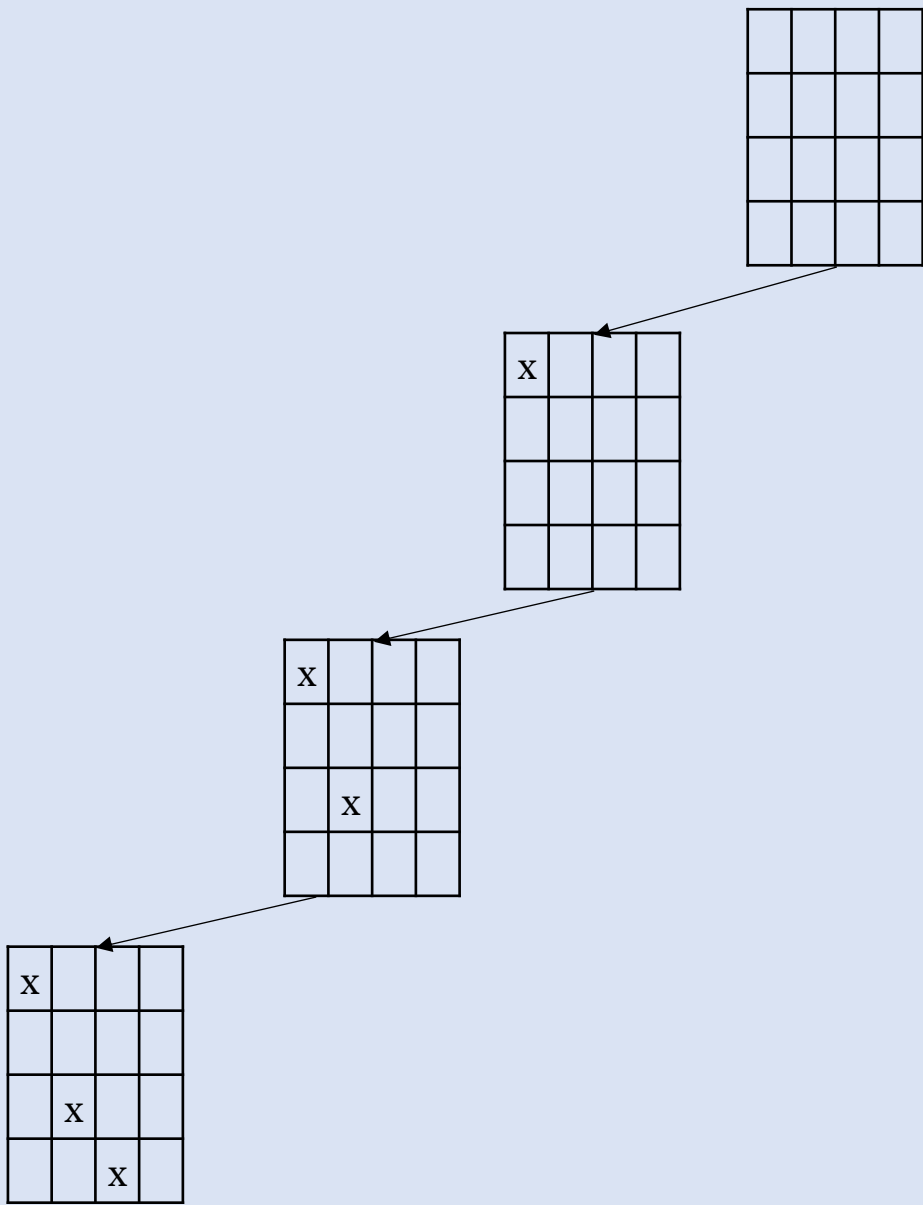


X			
	X		







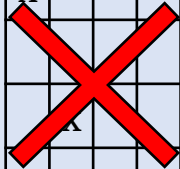


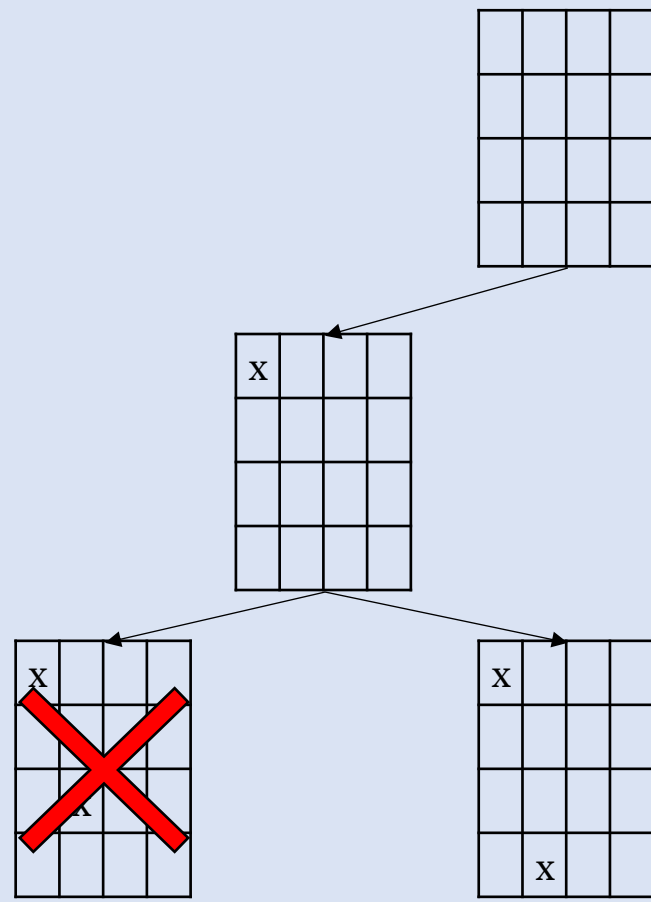


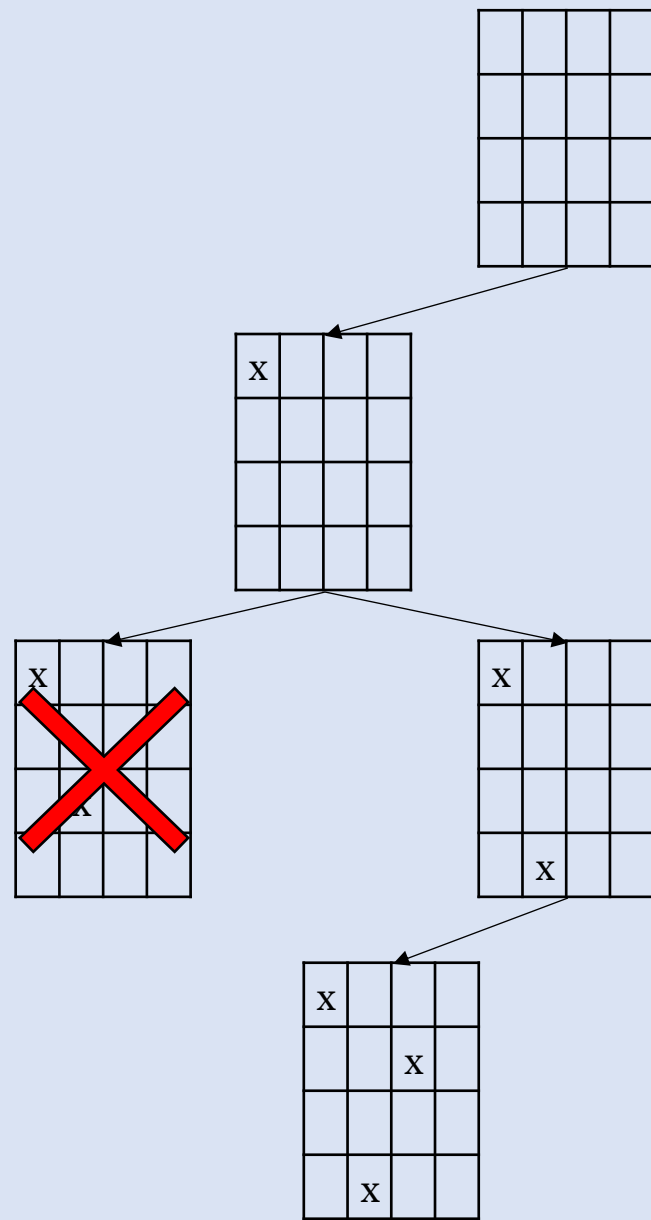
x			

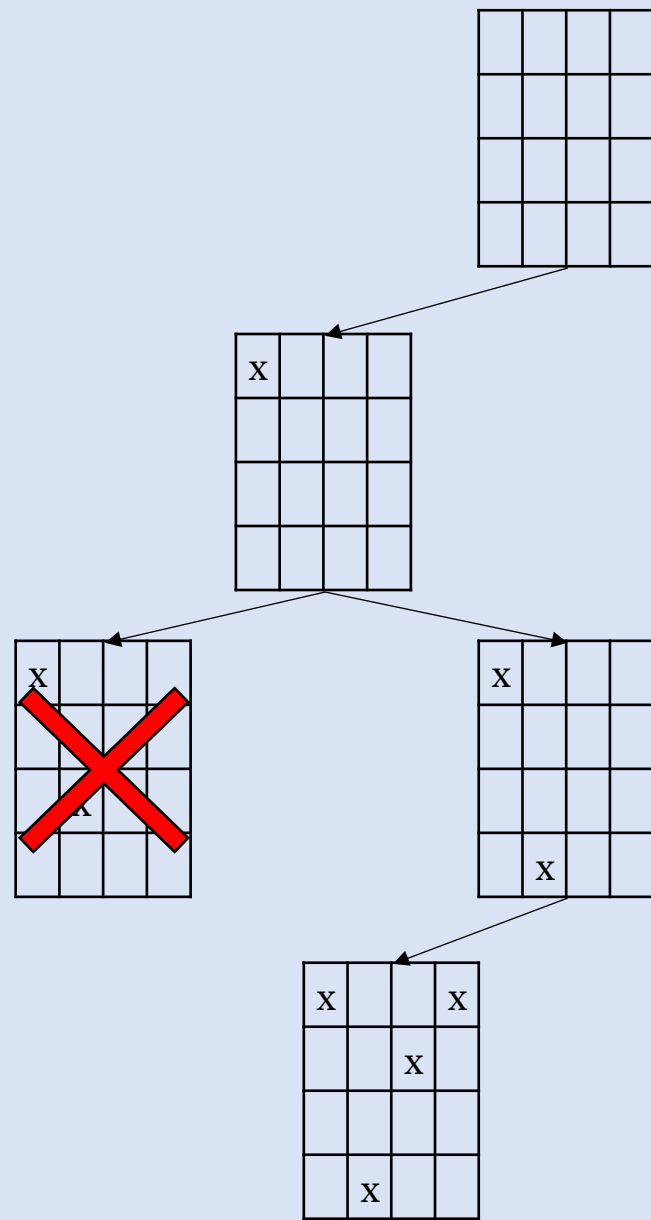


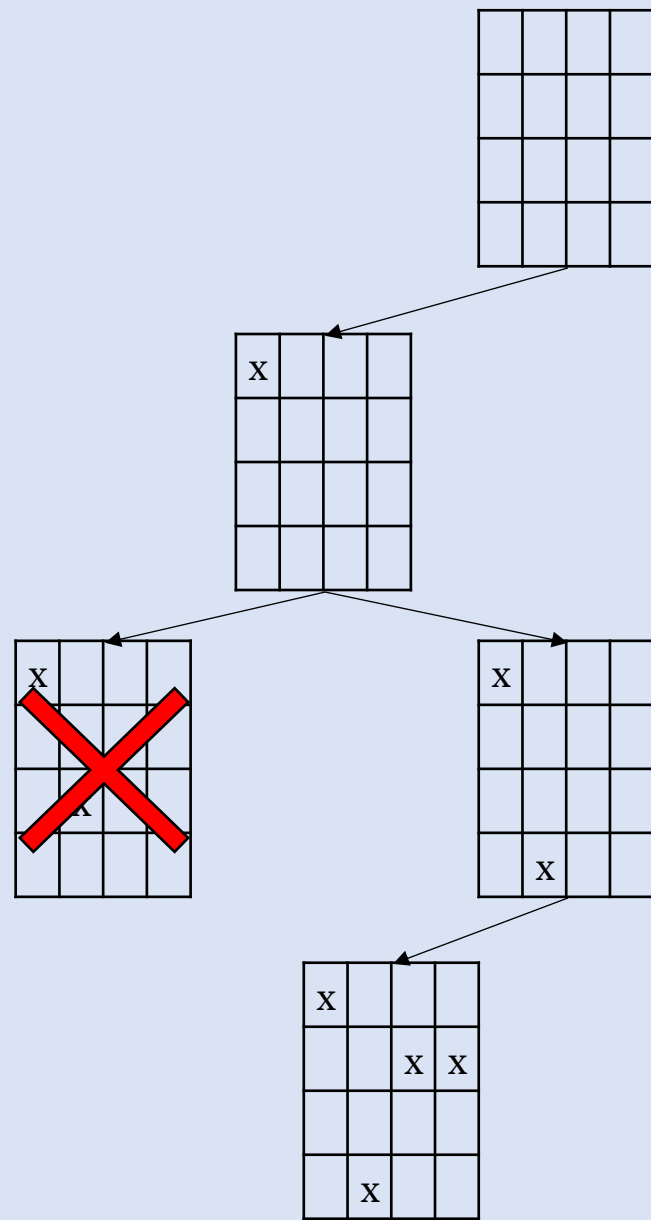
x			

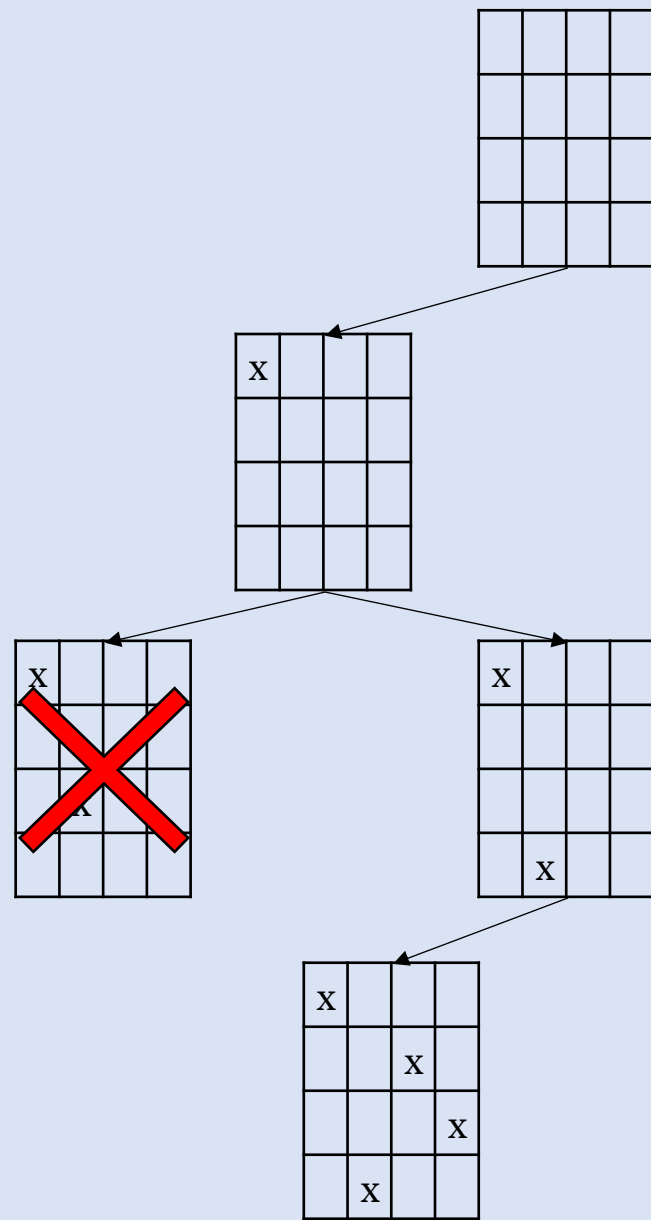


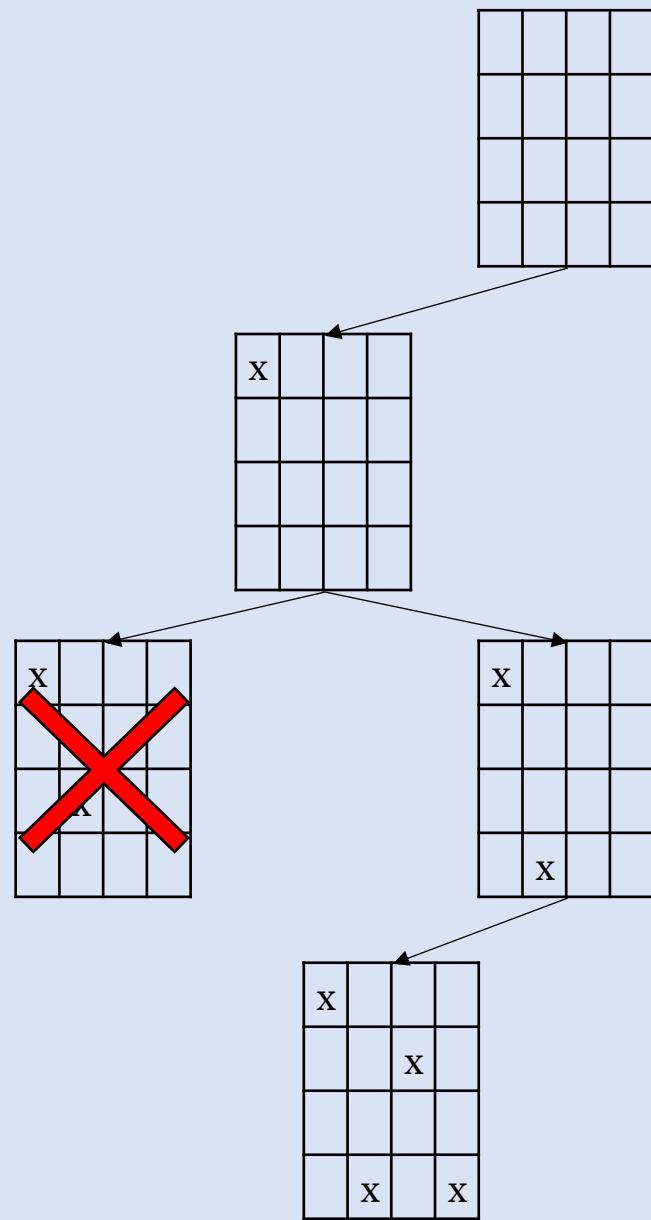


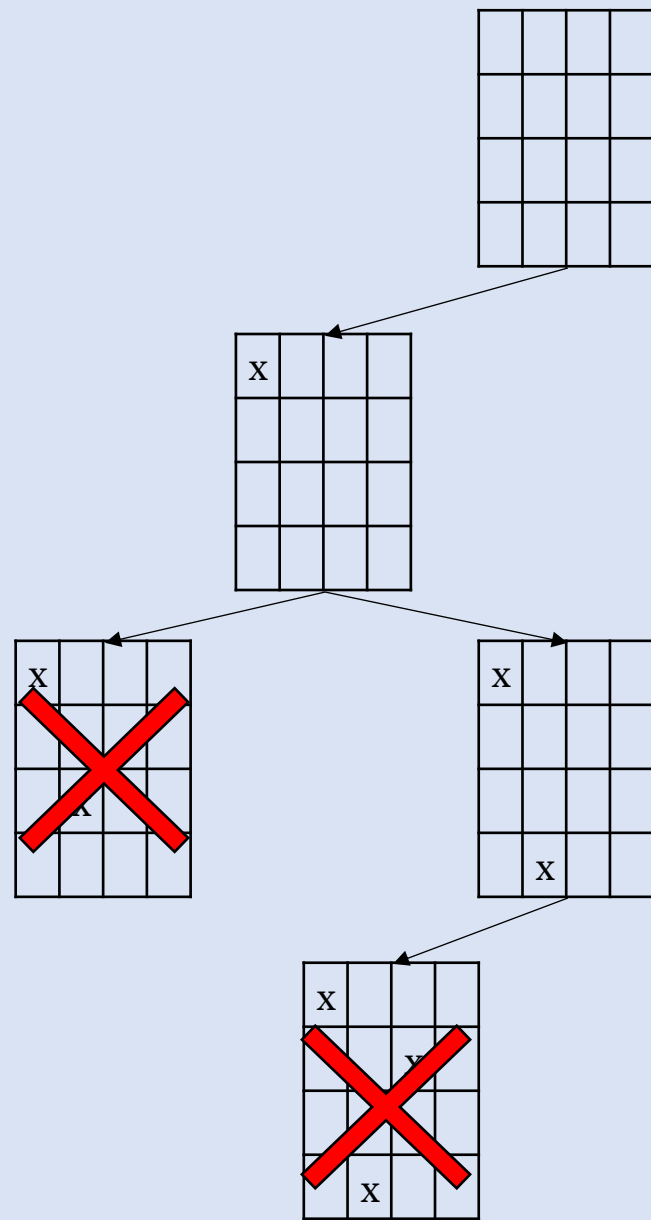


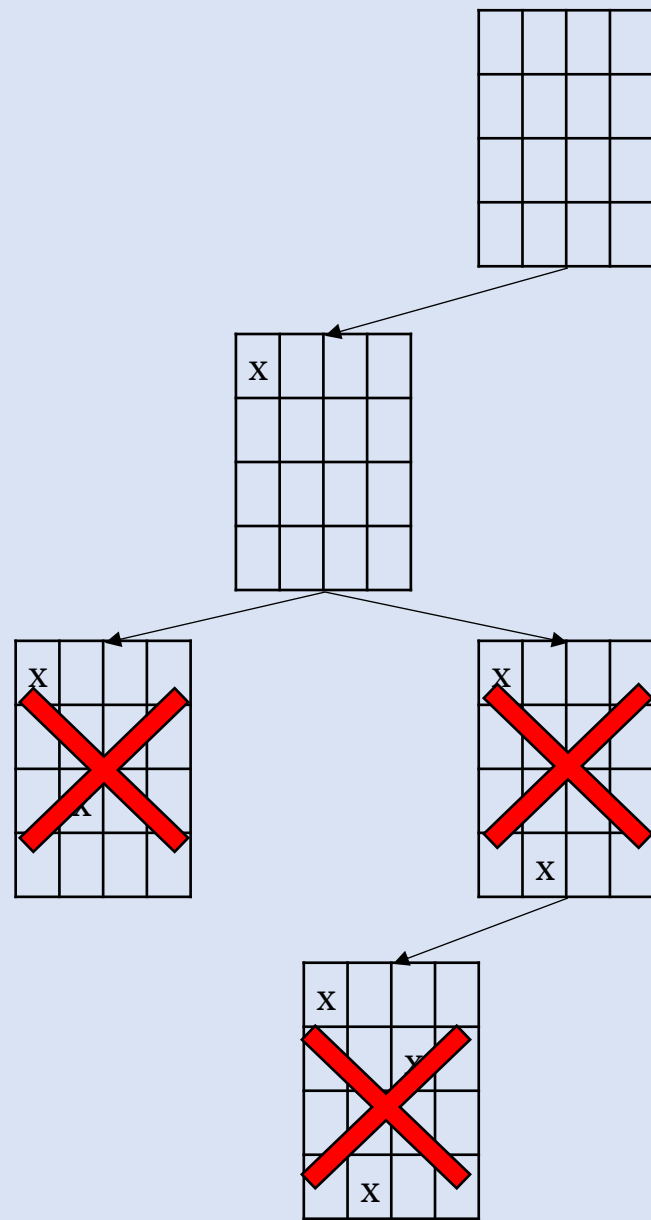


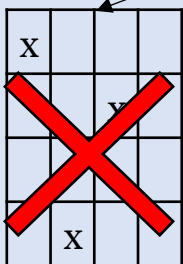
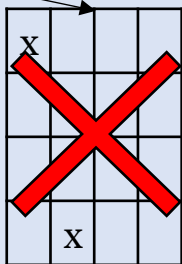
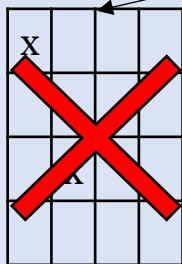
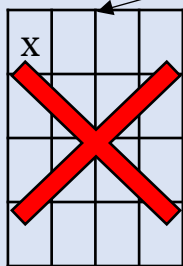
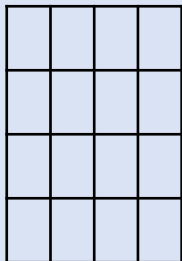


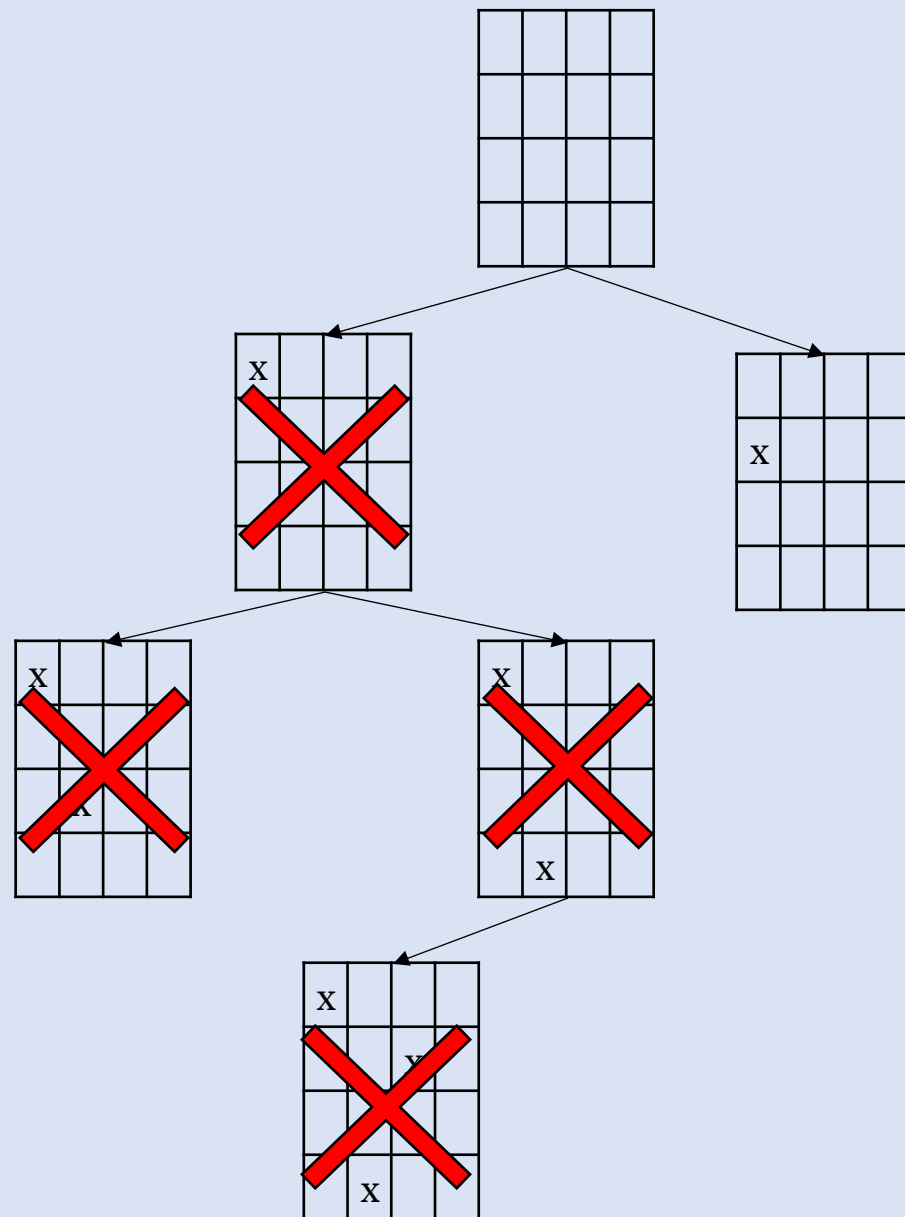


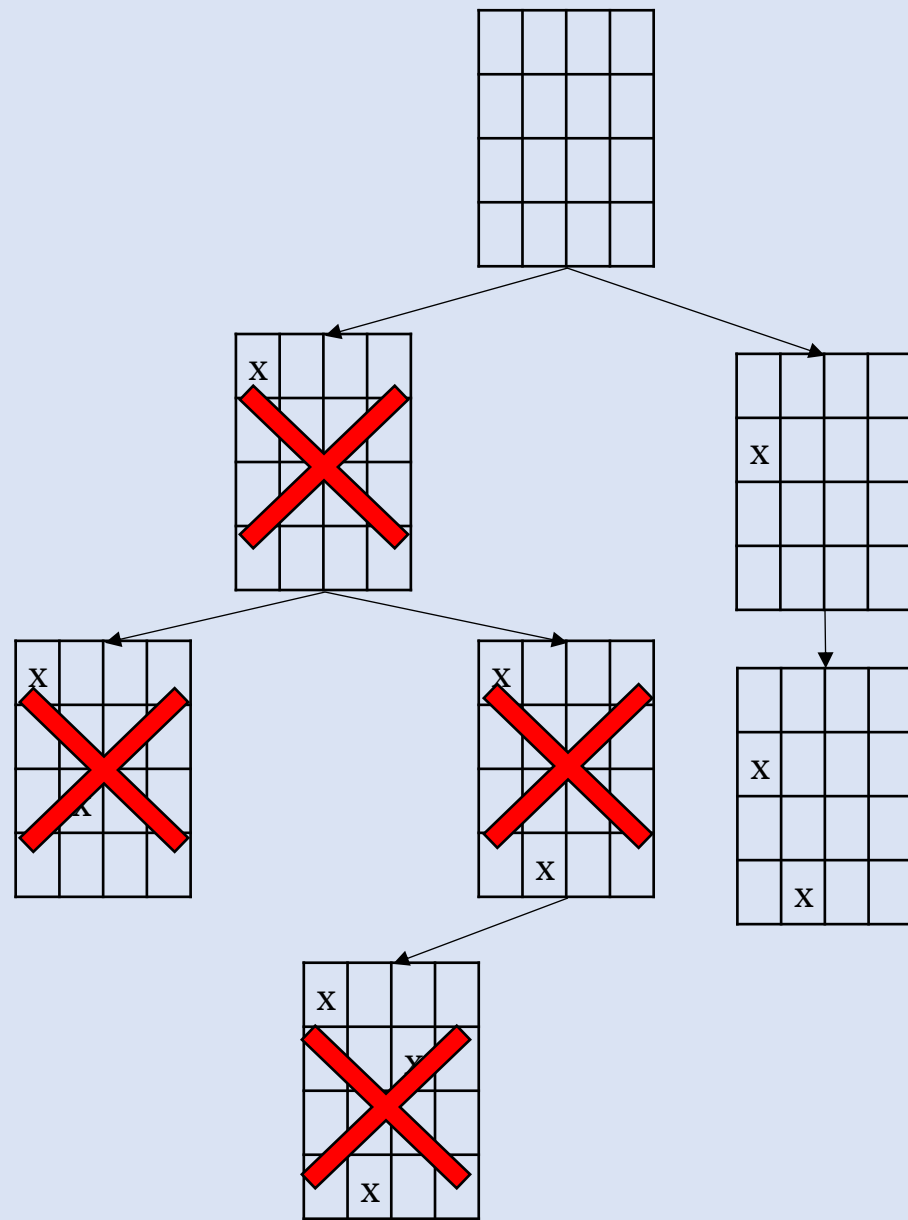


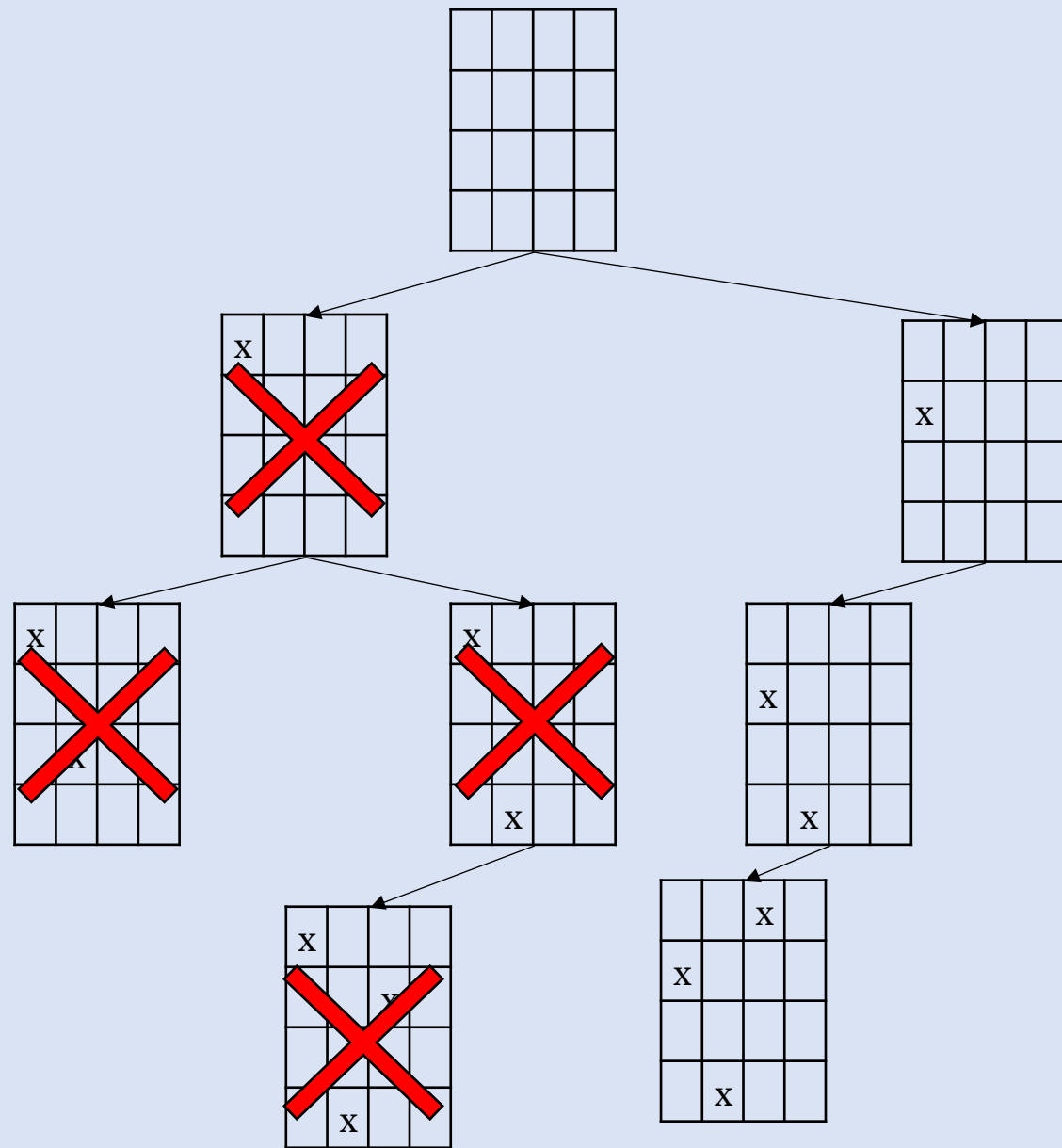


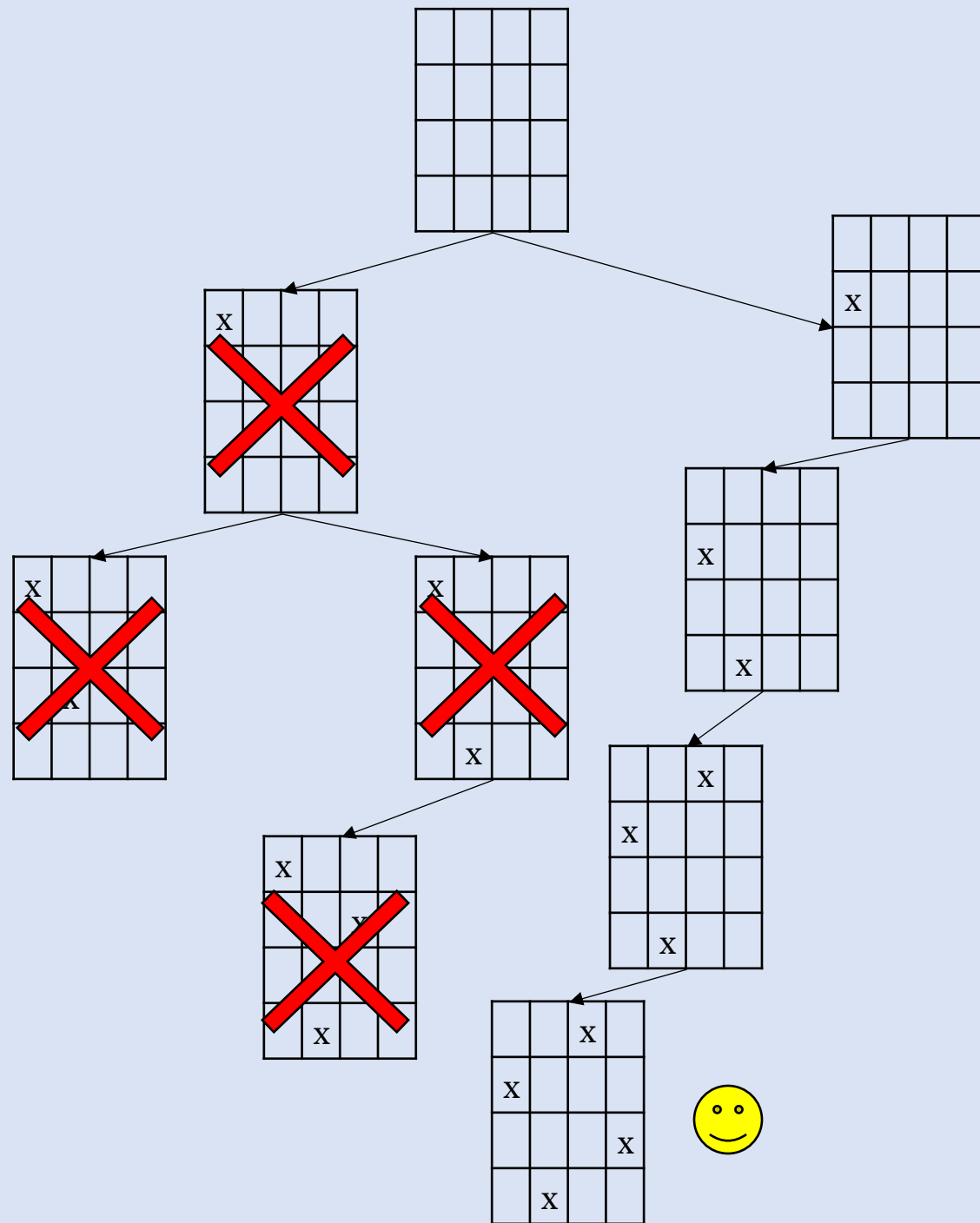












Let's code it up

Running Time Analysis

- The heart and soul of our running time analysis is all derived from recursive calls.
- Inside our setup method where all variables are set to their default values is the recursive method is called once.
- However, what about the recursive method itself?

The Recursive Method Analyzation

- When column (k) = 0 the recursive method is called one time from solveNQueen()
- When $k > 0$, there are $n(n - 1) \dots (n - k + 2)$ ways to place $k - 1$ queens in the first $k - 1$ columns in different rows.

$$n(n - 1) \dots (n - k + 2)$$

Ignoring Recursion Component of our Recursive Method

- Our recursive method has one for loop that goes up to n , which can be represented as $\Theta(n)$ for $k < n$, therefore, the worst-case is at most n , for $k = 1$, and $n[n(n - 1) \dots (n - k + 2)]$, for $1 < k < n$
- When $k = n - 1$, all rows except for one are occupied. This means our positionOk method should return true for one call.
- Our recursive method runs $O(n)$ when $k = n - 1$
- There are $n(n - 1) \dots 2$ ways to place $n - 1$ queens in the first $n - 1$ columns in different rows. Therefore the worst-case for our recursive call solveNQueen($n - 1$) is
$$n[n(n - 1) \dots 2]$$

Now let's put together our recursive analysis

- Combining the equations from the previous slides, we can find the worst-case time overall.

$$n[1 + n + n(n - 1) + \cdots + n(n - 1) \dots 2]$$

$$n * n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{1!} \right]$$

This you should
learn from
Calculus

$$\longrightarrow e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Now let's put together our recursive analysis

- Combining the equations from the previous slides, we can find the worst-case time overall.

That
Means...

This you should
learn from
Calculus

$$\longrightarrow e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Now let's put together our recursive analysis

$$n[1 + n + n(n - 1) + \cdots + n(n - 1) \dots 2]$$

$$n * n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{1!} \right]$$

This you should
learn from
Calculus $\longrightarrow e = \sum_{i=0}^{\infty} \frac{1}{i!}$

$$n * n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{1!} \right] = n * n! \sum_{i=0}^{\infty} \frac{1}{i!} = n * n! (e - 1)$$

Now let's put the other running time analysis

The Running Time
is $O(n * n!)!!!!$

$$n * n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{1!} \right] = n * n! \sum_{i=0}^n \frac{1}{i!} = n * n! (e - 1)$$