CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 15: Stacks and Queues

Announcements

- Midterm #1 on Friday

Recap

Stacks: Last In First Out (LIFO)

- Push (put item on top of the stack)
- Pop (take item off top of stack)
- Peek (peek at top of stack)

Queues: First in First Out (FIFO)

- Enqueue (put item on the end of the queue)
- Dequeue (take item off the front of the queue)
- Peek (peek at the front of the queue)

Queues

Thought Question: How could you use an array to build a queue?

Queues - ArrayList Implementation

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Queues - ArrayList Implementation

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

- `data.add(value);` — Amortized $\Theta(1)$
- `return data.remove(0);` — $\Theta(n)$:(
- `return data.get(0);` — $\Theta(1)$

Queues - ArrayList Implementation

```
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(0, value);
  }
  public E remove() { // dequeue
    return data.remove(data.size() - 1);
  }
  public E peek() {
    return data.get(data.size() - 1);
  }
}
```

Queues - ArrayList Implementation

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(0, value);
  }
  public E remove() { // dequeue
    return data.remove(data.size() - 1);
  }
  public E peek() {
    return data.get(data.size() - 1);
  }
}
```

$\Theta(n)$:(

$\Theta(1)$

$\Theta(1)$

Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

Why didn't we have to pay that cost with a list?

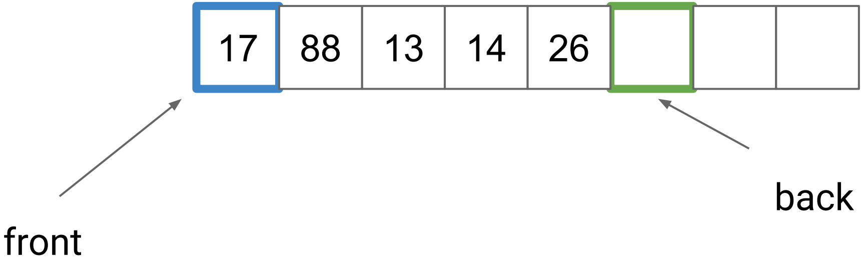
Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

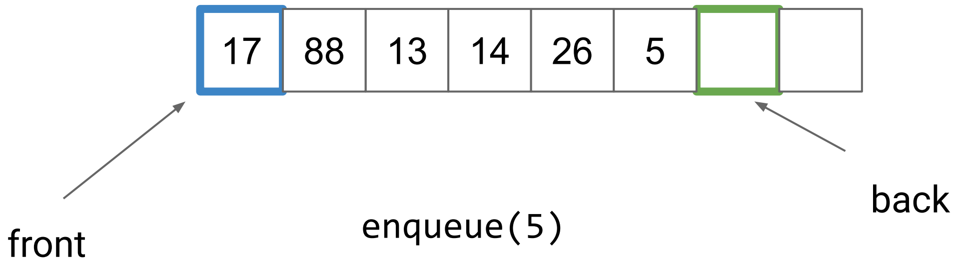
Why didn't we have to pay that cost with a list?

Update our values of "first" and "last"!

Queues



Queues



Queues


```
enqueue(5)
enqueue(4)
```

Queues


```
enqueue(5)
enqueue(4)
dequeue()
```

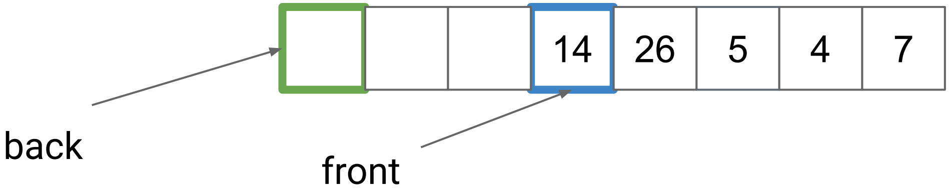
Queues


```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
```

Queues

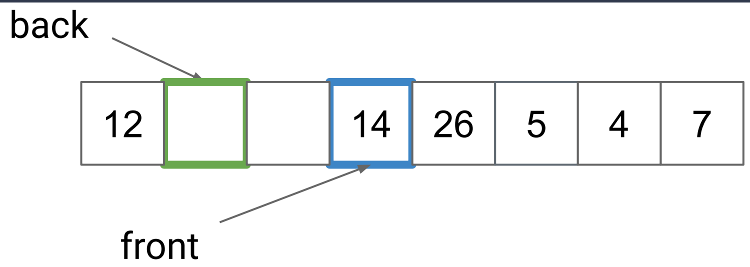

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
```

Queues



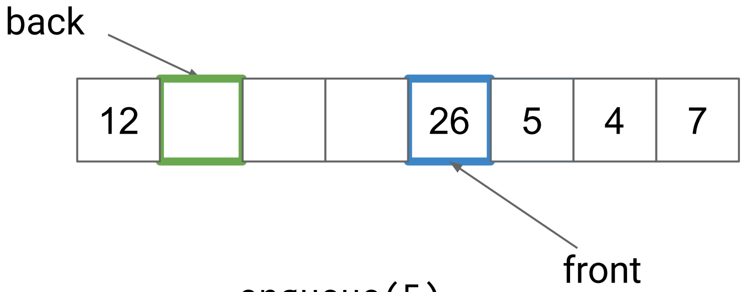

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
```

Queues



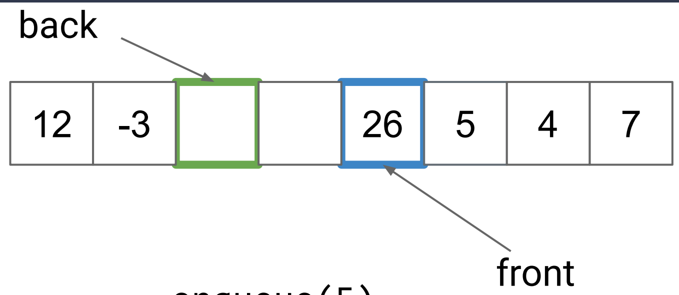

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
```

Queues




```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
```

Queues




```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
enqueue(-3)
```

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue

1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue

1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue

1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue

1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

What is the complexity?

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue Amortized $\Theta(1)$

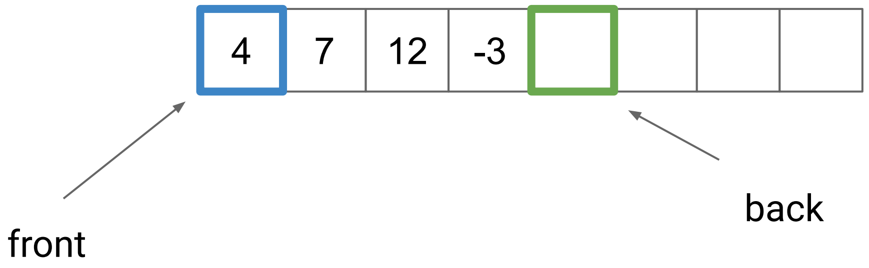
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue $\Theta(1)$

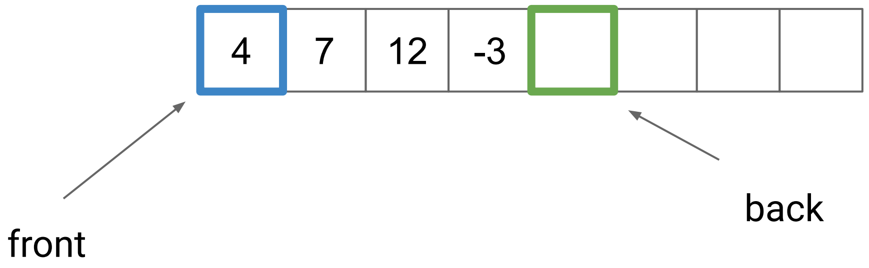
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

What is the complexity?

Why Ring Buffer?

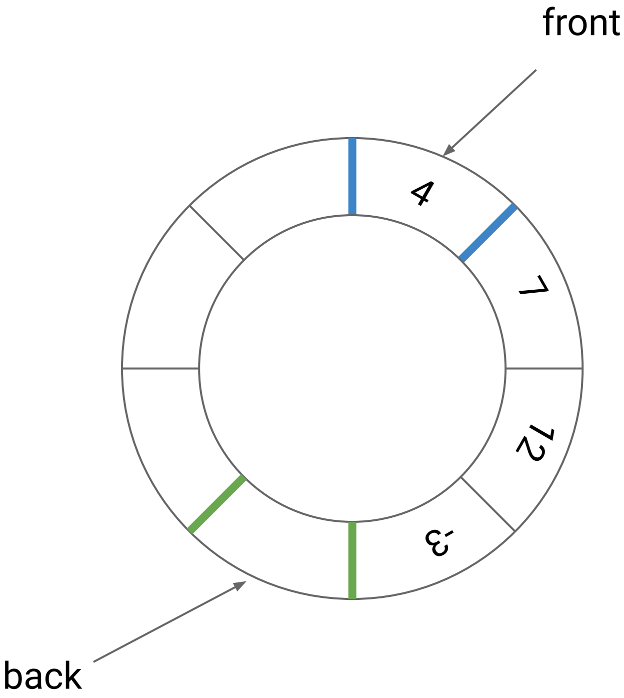


Why Ring Buffer?

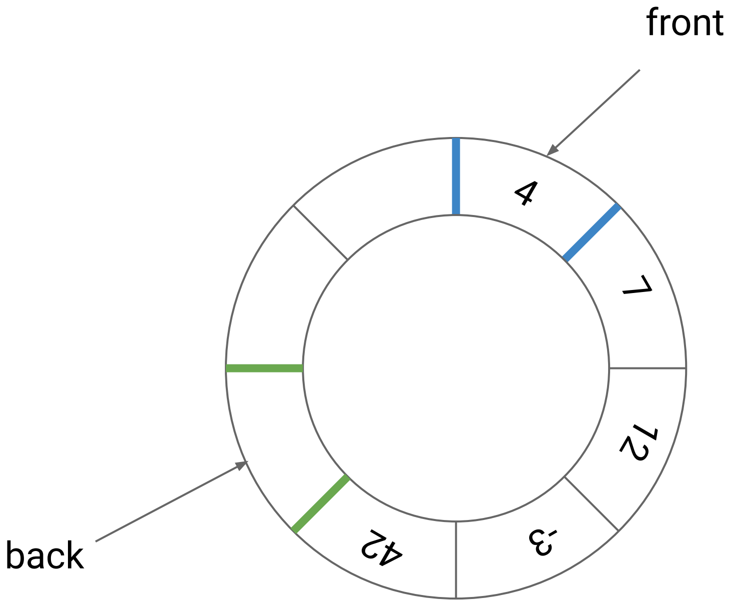


Conceptually, we can think of this as a ring…

Why Ring Buffer?

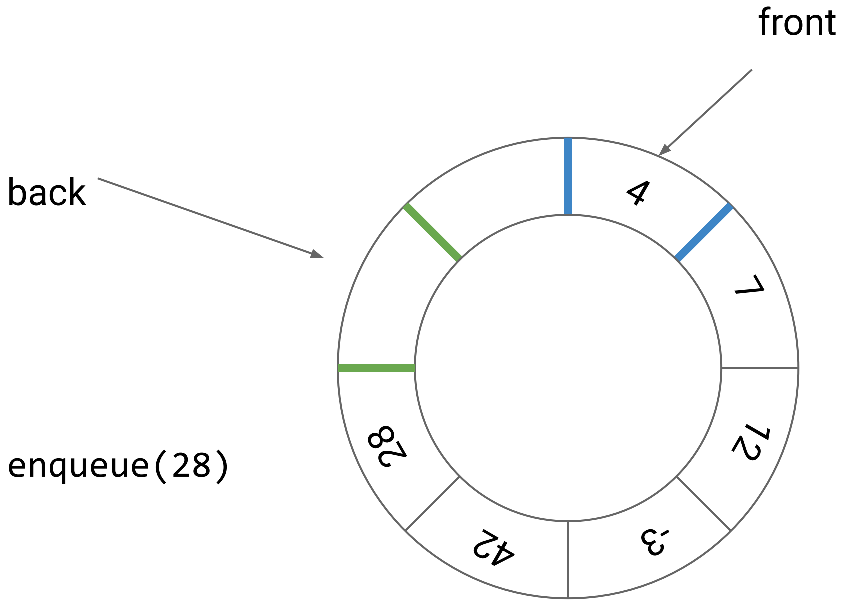


Why Ring Buffer?



`enqueue(42)`

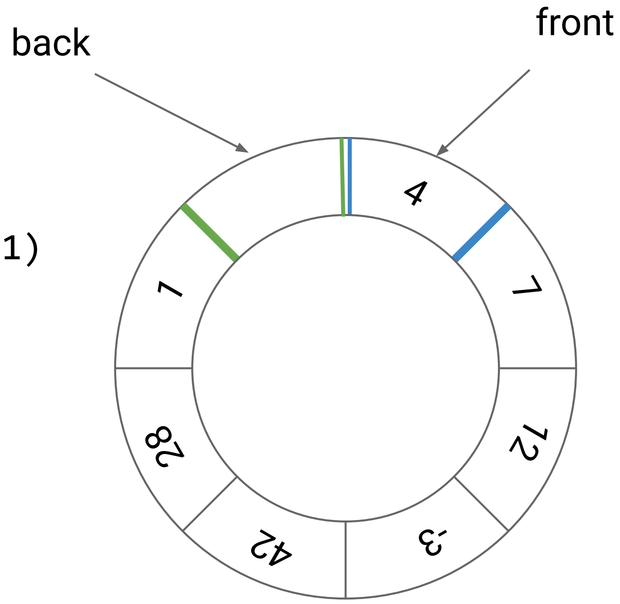
Why Ring Buffer?



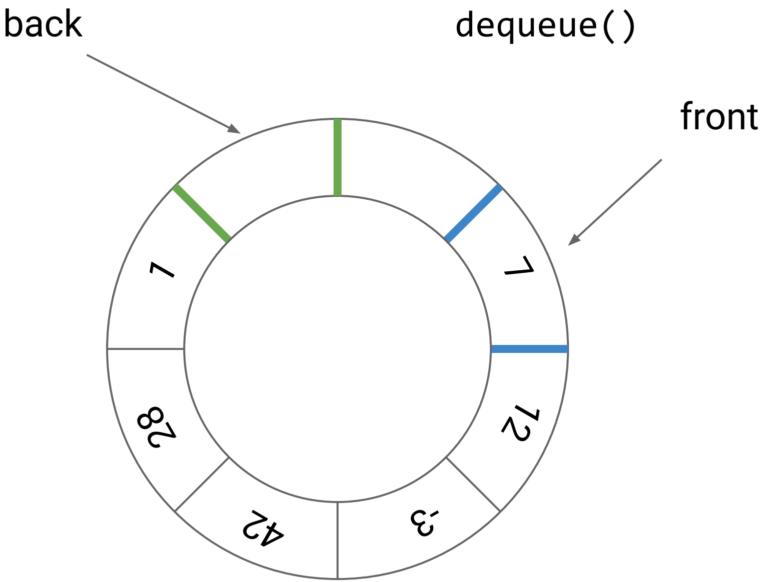
enqueue(28)

Why Ring Buffer?

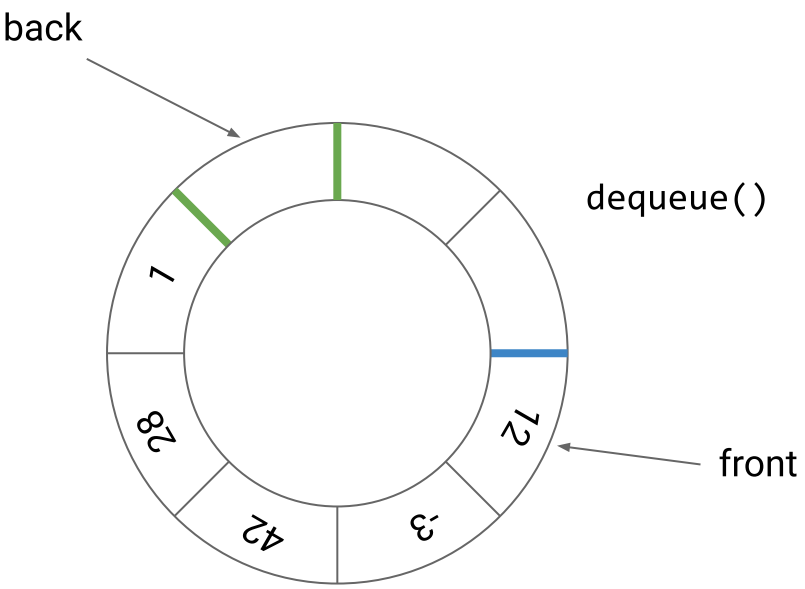
`enqueue(1)`



Why Ring Buffer?



Why Ring Buffer?



Applications of Stacks and Queue

Stack: Checking for balanced parentheses/braces

Queue: Scheduling packets for delivery

Both: Searching mazes

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

✓

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

✓   ✗

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}` ✓

`{())` ✗

`())` ✗

Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on `(`, decrement on `)`

Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on `(`, decrement on `)`

Problem: allows for `{(})`

Idea #2

Idea: Track nesting on a stack!

On `(` or `{`, push the symbol on the stack.

On `)` or `}`, pop the stack and check for a match.

Demo from last fall:

[https://odin.cse.buffalo.edu/teaching/cse-250/2022fa/slide/14b-QueueStackApps.html#/13]

Network Packets

Router: 1gb/s internal network, 100mb/s external

- 1 gb/s sent to the router, but only 100mb/s can leave.
- How do we handle this?

Queues

- Enqueue data packets in the order they are received.
- When there is available outgoing bandwidth, dequeue and send.

Avoiding Queueing Delays

- Limit size of queue; Packets that don't fit are dropped

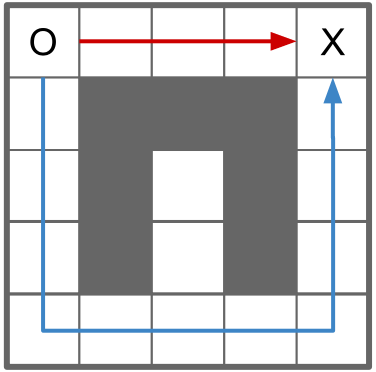
TCP: blocked packets are retried

UDP: application deals with dropped packets

Mazes

O is the start, **X** is the objective

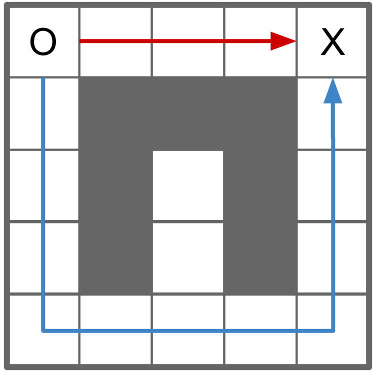
- There may be multiple paths
- Generally, we want the shortest



Approach 1: Take the first available route in one direction

- Right, Down, Left, or Up
- Down, Right, Up, or Left

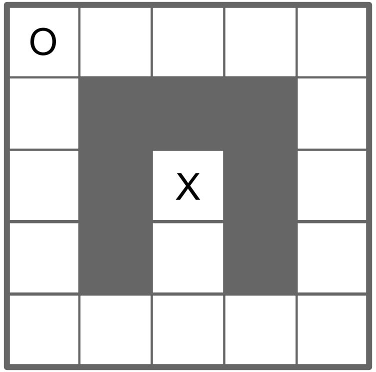
Mazes



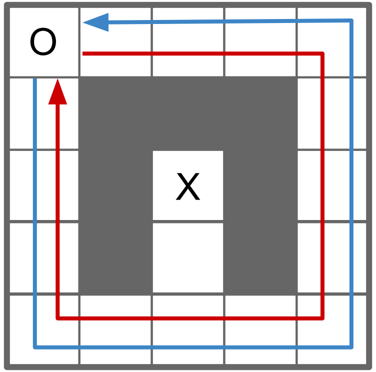
How do you know which one is best?

Is there anything wrong with this algorithm?

Mazes



Mazes



Priority order doesn't guarantee exploring the entire maze

Formalizing Maze-Solving

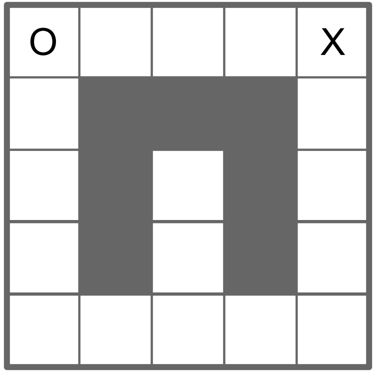
Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

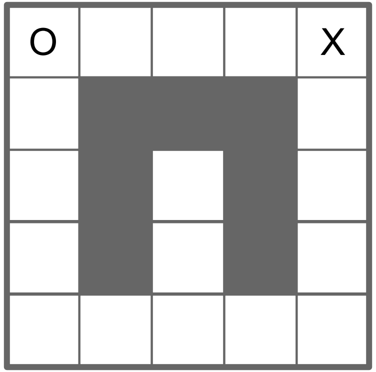
Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end.

How do we define the `steps` *function?*

Mazes

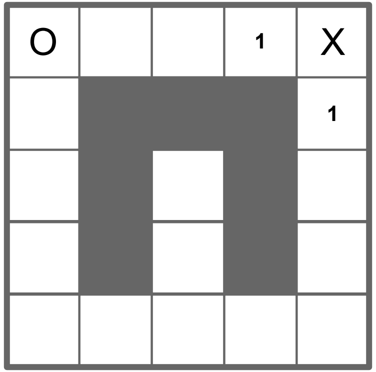


Mazes



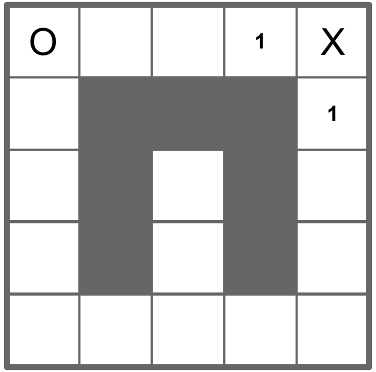
How many steps are required for the squares right next to X?

Mazes



How many steps are required for the squares right next to X?

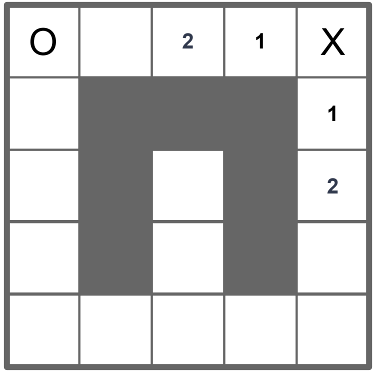
Mazes



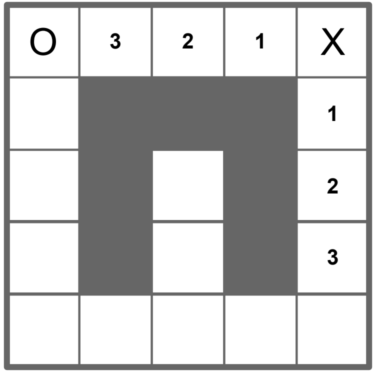
How many steps are required for the squares right next to X?

And the squares next to those?

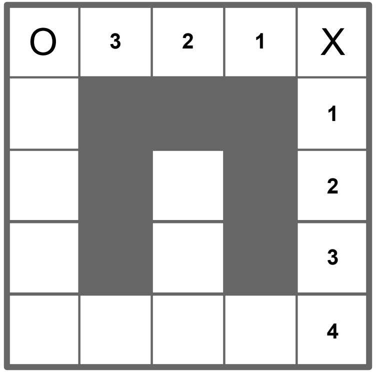
Mazes



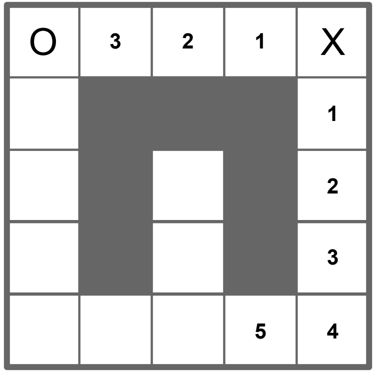
Mazes



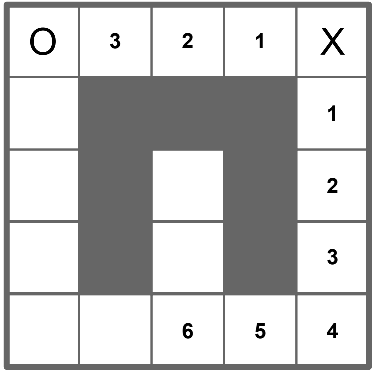
Mazes



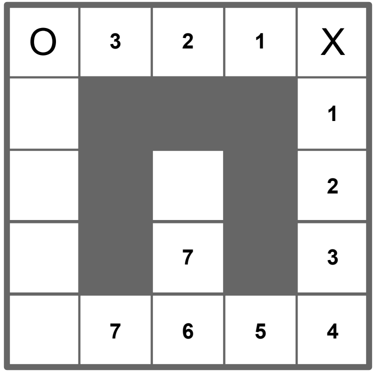
Mazes



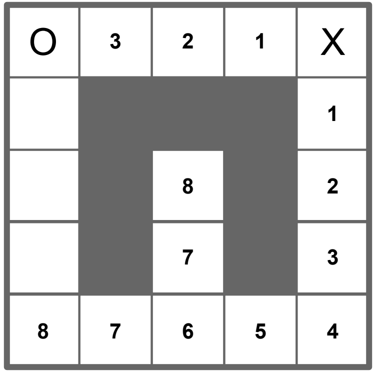
Mazes



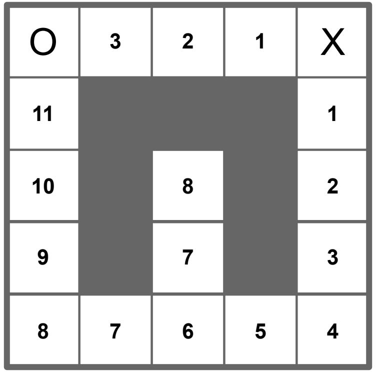
Mazes



Mazes



Mazes



Mazes

O	3	2	1	X
11	∞	∞	∞	1
10	∞	8	∞	2
9	∞	7	∞	3
8	7	6	5	4

Mazes

O	3	2	1	X
11	∞	∞	∞	1
10	∞	8	∞	2
9	∞	7	∞	3
8	7	6	5	4

So what is the number of steps from O to X?

Mazes

O	3	2	1	X
11	∞	∞	∞	1
10	∞	8	∞	2
9	∞	7	∞	3
8	7	6	5	4

So what is the number of steps from O *to* X?

4 (min of neighbors + 1)

Mazes

O	3	2	1	X
11	∞	∞	∞	1
10	∞	8	∞	2
9	∞	7	∞	3
8	7	6	5	4

Does this solution remind you of anything?

Mazes

O	3	2	1	X
11	∞	∞	∞	1
10	∞	8	∞	2
9	∞	7	∞	3
8	7	6	5	4

Does this solution remind you of anything?

Recursion!

Mazes

$$steps(\texttt{pos}, \texttt{dest}) = \begin{cases} 0 & \textbf{if } \texttt{pos} = \texttt{dest} \\ \infty & \textbf{if } is\_filled(\texttt{pos}) \\ 1 + min\_adjacent(\texttt{pos}, \texttt{dest}) & \textbf{otherwise} \end{cases}$$

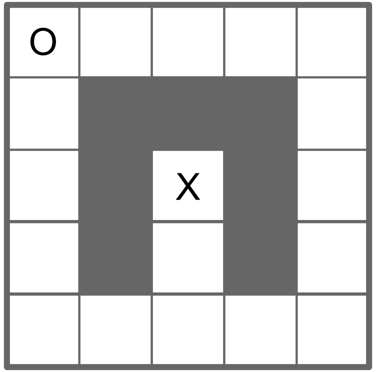
where...

$$min\_adjacent(\texttt{pos}, \texttt{dest}) = \min \begin{cases} steps(moveRight(\texttt{pos}), \texttt{dest}) \\ steps(moveDown(\texttt{pos}), \texttt{dest}) \\ steps(moveLeft(\texttt{pos}), \texttt{dest}) \\ steps(moveUp(\texttt{pos}), \texttt{dest}) \end{cases}$$

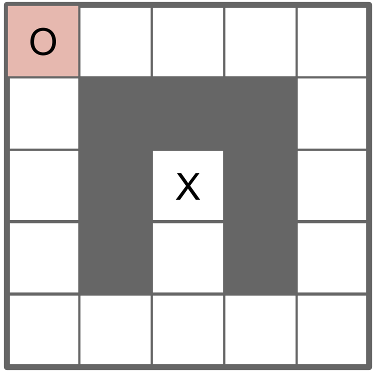
Mazes

```
steps(pos, dest):
  if pos == dest then return 0
  elif is_filled(pos) then return ∞
  else return 1 + min of
    steps(moveRight(pos, dest))
    steps(moveDown(pos, dest))
    steps(moveLeft(pos, dest))
    steps(moveUp(pos, dest))
```

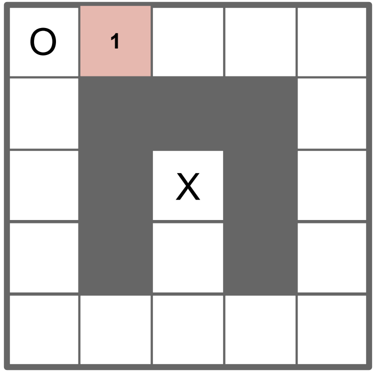
Mazes



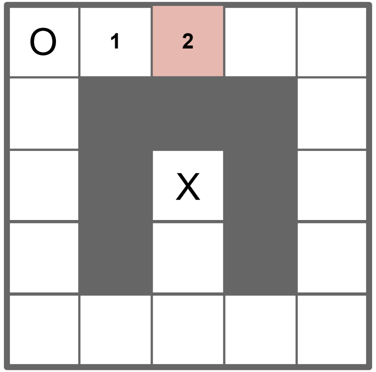
Mazes



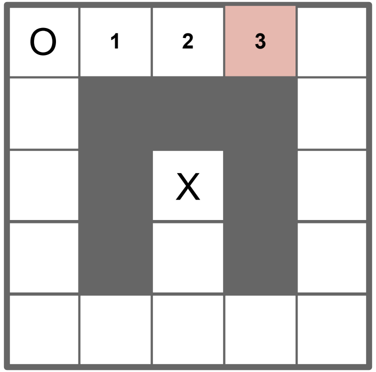
Mazes



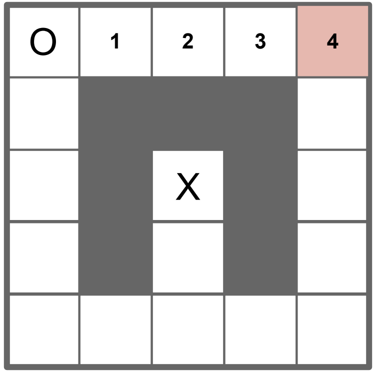
Mazes



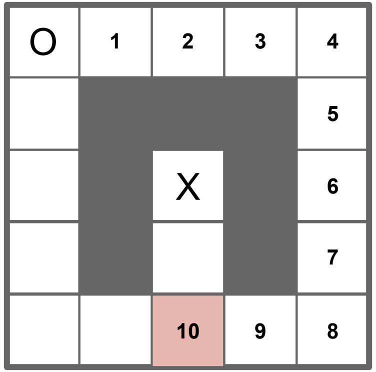
Mazes



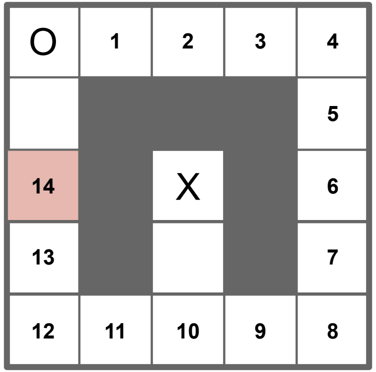
Mazes



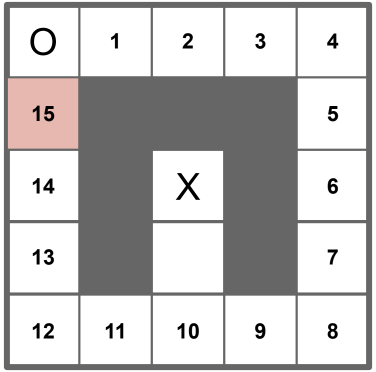
Mazes



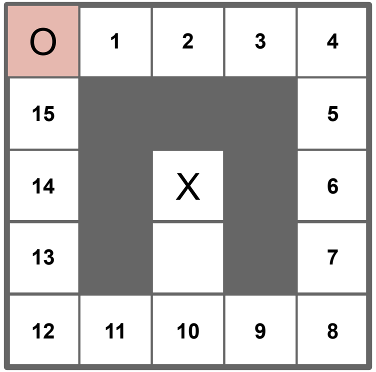
Mazes



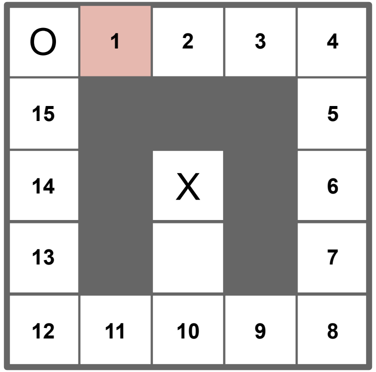
Mazes



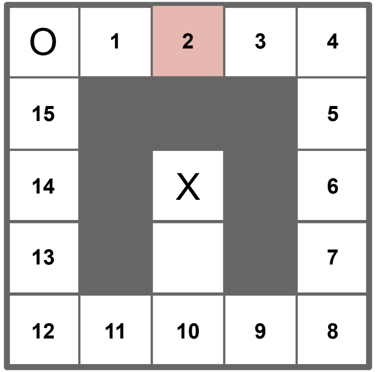
Mazes



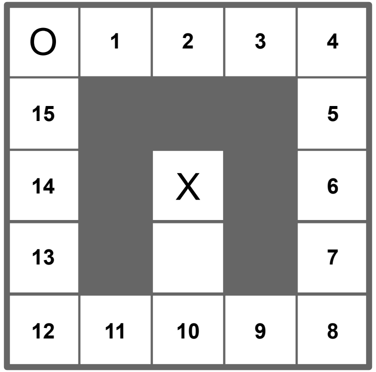
Mazes



Mazes

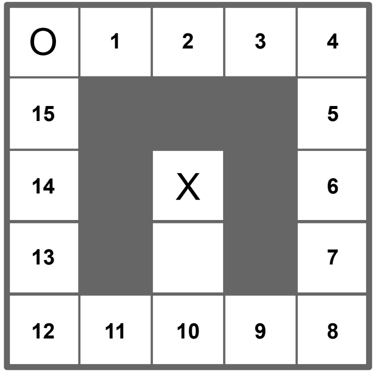


Mazes



Problem: Infinite loop!

Mazes



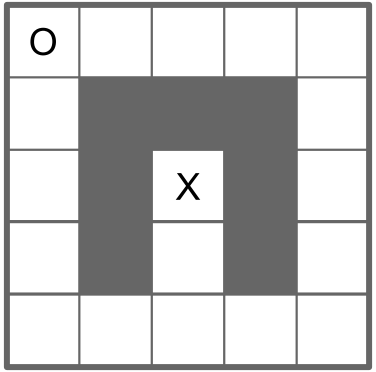
Problem: Infinite loop!

Insight: A path with a loop in it can't be shorter than one without the loop

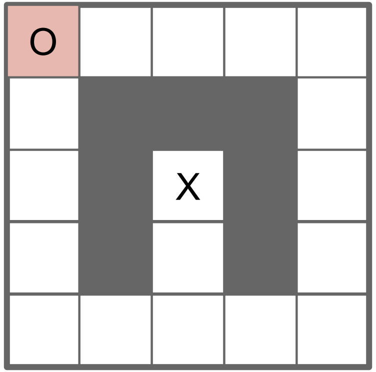
Mazes

```
steps(pos, dest):
  if pos == dest then return 0
  elif is_visited(pos) then return ∞
  elif is_filled(pos) then return ∞
  else
    Mark pos as visited
    return 1 + min of all 4 steps
```

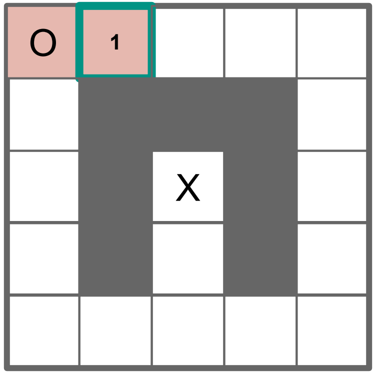
Mazes



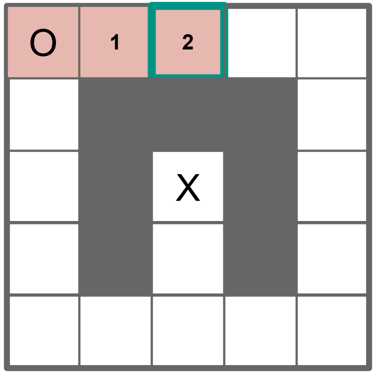
Mazes



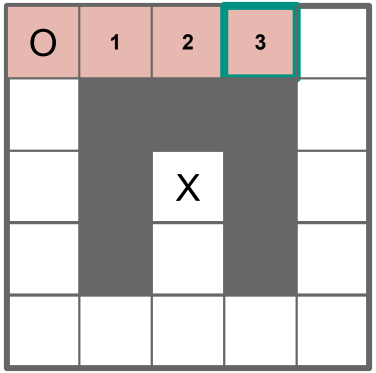
Mazes



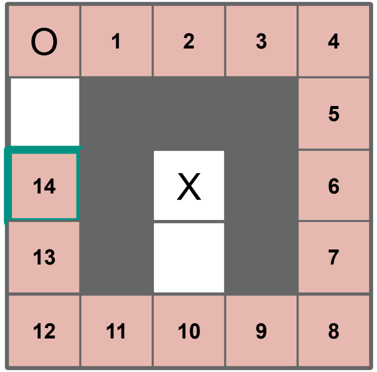
Mazes



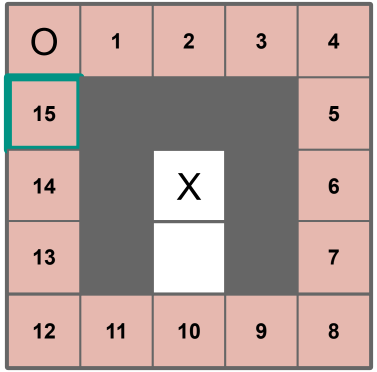
Mazes



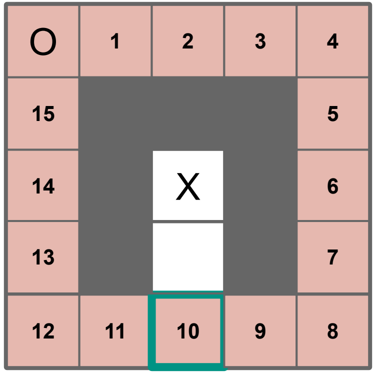
Mazes



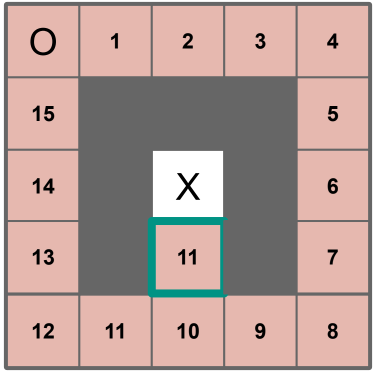
Mazes



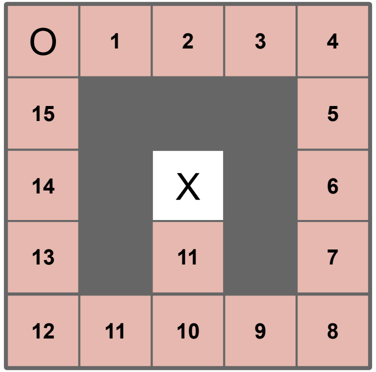
Mazes



Mazes

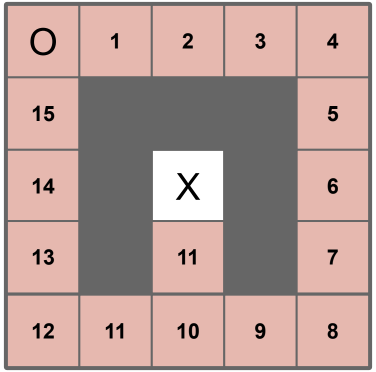


Mazes



Problem: The first time you visit a node may be from a longer path!

Mazes



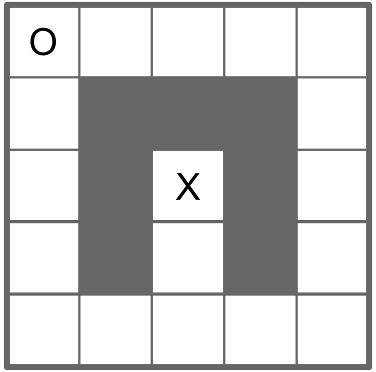
Problem: The first time you visit a node may be from a longer path!

Insight: Unmark nodes as you leave them

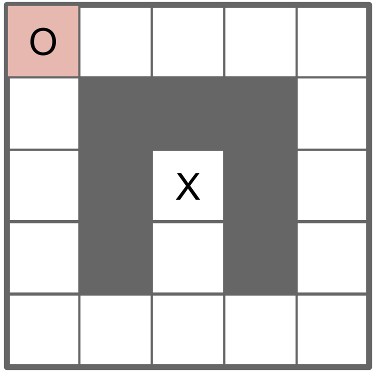
Mazes

```
steps(pos, dest):
  if pos == dest then return 0
  elif is_visited(pos) then return ∞
  elif is_filled(pos) then return ∞
  else
    Mark pos as visited
    min = 1 + min of all 4 steps
    Mark pos as unvisited
    return min
```

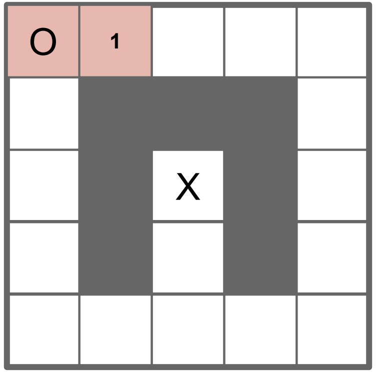
Mazes



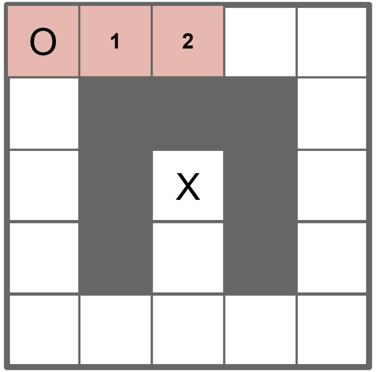
Mazes



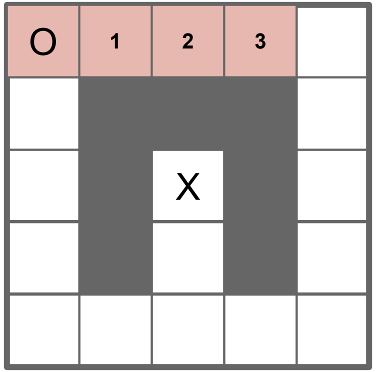
Mazes



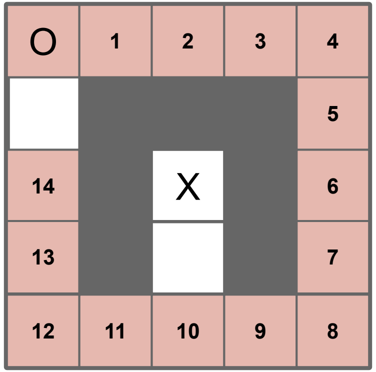
Mazes



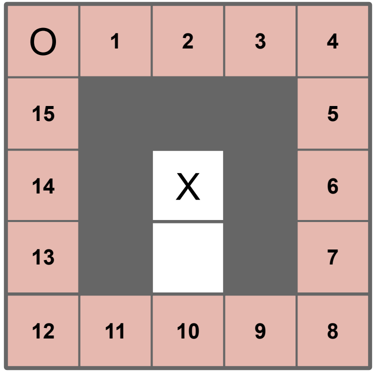
Mazes



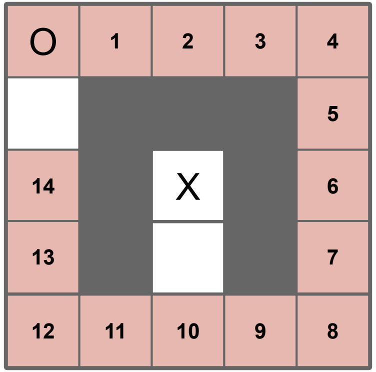
Mazes



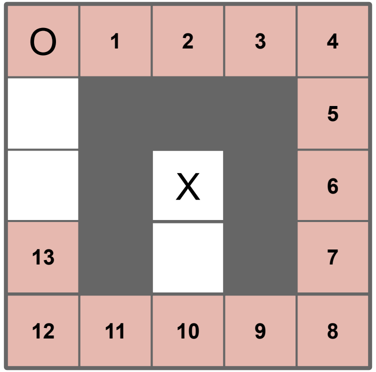
Mazes



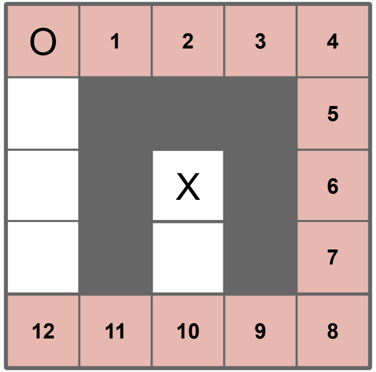
Mazes



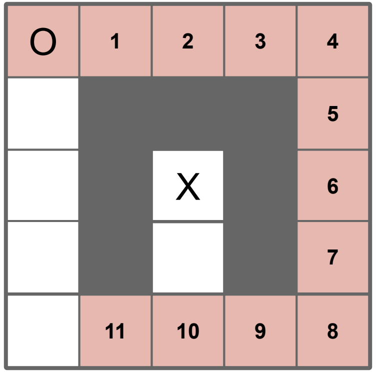
Mazes



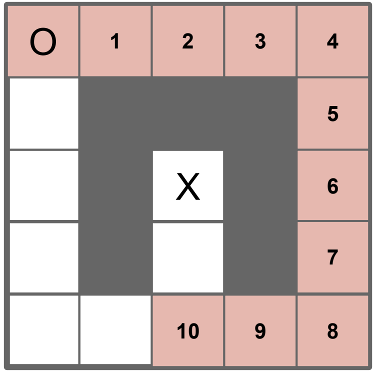
Mazes



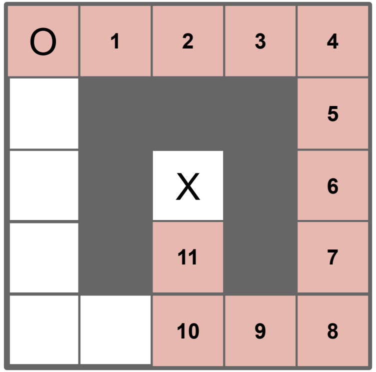
Mazes



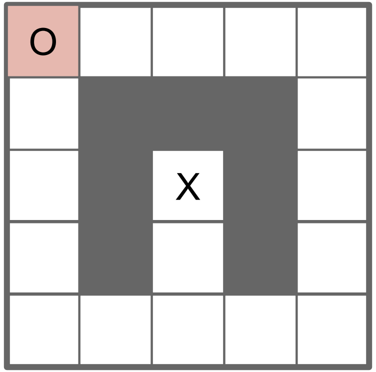
Mazes



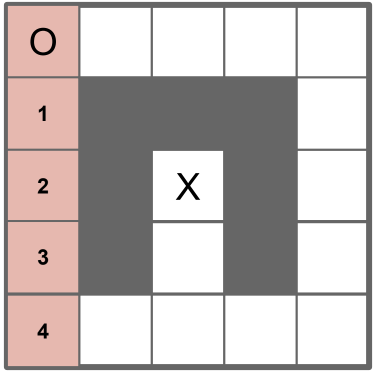
Mazes



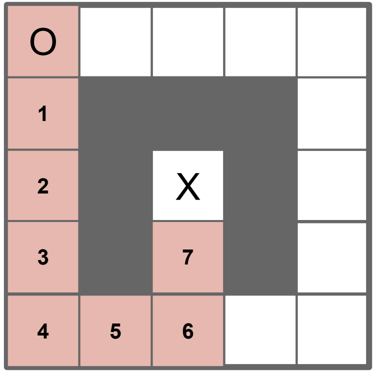
Mazes



Mazes



Mazes



Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end.

Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end. ✓

Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end. ✓

What path did we take?

Mazes

Idea: Keep track of the nodes marked visited…that's our path!

Mazes: Now with...some data structure?

```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∈ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.append(pos)
    bestPath = 1 + min of all 4 steps
    visited.removeLast()
    return bestPath
```

Mazes: Now with...some data structure?

```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∈ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.append(pos)
    bestPath = 1 + min of all 4 steps
    visited.removeLast()
    return bestPath
```

What could this data structure be??

Mazes: Now with...Stacks!

```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∈ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.push(pos)
    bestPath = 1 + min of all 4 steps
    visited.pop()
    return bestPath
```

A stack!

Queues?

Thought Experiment: Can we do something similar with queues?