Q1 Excel with Integrity Pledge

0 Points

I will complete this exam in a fair, honest, respectful, responsible and trustworthy manner. This means that I will complete the exam as if the professor was watching my every action. I will act according to the professor's instructions, and I will neither give nor receive any aid or assistance other than what is authorized. I know that the integrity of this exam and this class is up to me, and I pledge to not take any action that would break the trust of my classmates or professor, or undermine the fairness of this class.

I promise to complete this exam in keeping with the Excel with Integrity Pledge.

Fill in your Name and today’s Date below:

Your name, today's date

Q2 Instructions

0 Points

You have 90 minutes to complete the exam. Gradescope will not allow you to return to your exam after your 90 minutes has expired. Work to maximize points. If you get stuck, work through other problems and come back to it.

In general, if you think you've spotted a typo in the exam, do your best to answer in the spirit of the question. Keep in mind that some questions have interesting code examples with intentional bugs for you to find as part of the question. We won't answer any questions during the exam.

In general, assume that any necessary libraries (JUnit, ArrayList, List, and so on) have been imported.

You can use your notes and a compiler. However, the test is designed as if you were taking it during lecture without a compiler and it's highly suggested you not rely on your compiler and use pen & paper to figure out your answers.

Stay calm – you can do this!

Q2.1 Reference

0 Points

The following is a helpful Java reference that you can use during your exam.

```
class ArrayList<E>

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>,
Collection<E>, List<E>, RandomAccess
```

```
Interface List<E>

void add(int index, E element)
Inserts the specified element at the specified position
in this list.

boolean add(E e)
Appends the specified element to the end of this list.

E get(int index)
Returns the element at the specified position in this list.

int indexOf(Object o)
Returns the index of the first occurrence of the specified
element in this list, or -1 if this list does not contain
the element.
```

```
E remove(int index)
Removes the element at the specified position in this list.

boolean remove(Object o)
Removes the first occurrence of the specified element from
this list, if it is present.

int size()
Returns the number of elements in this list.
```

Q3 Queue

6 Points

Consider implementing the interface FixedLengthQueue:

```
interface FixedLengthQueue<E> {
    int getMaxElements();
    int size();
    boolean tryEnqueue(E elt);
    E dequeue();
}
```

A FixedLengthQueue is like a regular Queue, but it has some fixed number of elements it can hold.

If elements are enqueued (by **tryEnqueue()**) when the maximum number is already stored, **tryEnqueue()** has no effect, no element is added, and returns **false**. If the element is added, the method returns **true**.

dequeue() must throw some exception (any exception is allowed) if the queue is empty.

The ALFLQueue generic class implements the FixedLengthQueue interface as follows.

```
class ALFLQueue<E> implements FixedLengthQueue<E> {
    ArrayList<E> contents;
    final int maxElements;    // the fixed maximum number of
                              // elements the queue can hold

    public ALFLQueue(int maxElements) {
        this.contents = new ArrayList<E>();
        this.maxElements = maxElements;
    }

    public int size() { return this.contents.size(); }

    public int getMaxElements() { return this.maxElements; }

    public E dequeue() {
        // make sure to answer below!
    }

    public boolean tryEnqueue(E elt) {
        // make sure to answer below!
    }
}
```

Q3.1 A

2 Points

dequeue() must **throw** some exception (any exception is allowed) if the queue is empty.

Provide an implementation of **dequeue()** that matches the description above.

Q3.2 B

2 Points

If elements are enqueued (by **tryEnqueue()**) when the maximum number is already stored, **tryEnqueue()** has no effect, no element is added, and returns **false**. If an element is successfully added, the method returns **true**.

Provide an implementation of **tryEnqueue()** that matches the description above.

Q3.3 C

1 Point

Changing the type of the **contents** field from ArrayList to List would cause a compile error.

- ○ true
- ◉ false

Q3.4 D

1 Point

If we change the interface FixedLengthQueue to the version below, we would need to edit ALFLQueue to avoid introducing extra compile errors as we must use **E** for generic types:

```
interface FixedLengthQueue<Item> {
    int getMaxElements();
    int size();
    boolean tryEnqueue(Item elt);
    Item dequeue();
}
```

- ○ true
- ◉ false

Q4 ArrayList

6 Points

Consider this AList interface, implementation, and tests:

```
interface AList<E> {
  void add(E element);
  String concatAll();
}
public class AListImplementation<E> implements AList<E> {
  E[] elements;
  int size;
  public AListImplementation() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }
  public String concatAll() {
    String toReturn = "";
    for(int i = 0; i < this.elements.length; i += 1) {
      toReturn += this.elements[i];
    }
    return toReturn;
  }
  public void add(E element) {
    expandCapacity();
    this.elements[this.size] = element;
    this.size += 1;
  }
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }
    E[] expanded = (E[])(new Object[currentCapacity * 2]);
    for(int i = 0; i < this.elements.length; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

```
public class TestAList {
  @Test
  public void testThreeAdds() {
    AList<Integer> a1 = new AListImplementation<Integer>();
    a1.add(1); a1.add(2); a1.add(3);
    assertEquals("123", a1.concatAll());
  }
  @Test
  public void testFourAdds() {
    AList<Integer> a1 = new AListImplementation<Integer>();
    a1.add(1); a1.add(2); a1.add(3); a1.add(4);
    assertEquals("1234", a1.concatAll());
  }
}
```

You run the tests, and see that **testFourAdds()** passes, and **testThreeAdds()** fails with the following message:

org.junit.ComparisonFailure: expected:<123[]> but was:<123[null]>

Consider each of the following changes of **AListImplementation** independently: **each change is applied in isolation**, without any of the others.

Q4.1 A

1 Point

Change line 9 to:

```
this.elements = (E[])(new Object[3]);
```

Which tests would pass?

- ○ **Both** tests would pass
- ○ **Neither** test would pass
- ◉ Only **testThreeAdds()** would pass
- ○ Only **testFourAdds()** would pass

Q4.2 B

1 Point

Add this new line after line 29 (before the curly brace):

```
this.size += 1;
```

Which tests would pass?

- ○ **Both** tests would pass
- ◉ **Neither** test would pass
- ○ Only **testThreeAdds()** would pass
- ○ Only **testFourAdds()** would pass

Q4.3 C

1 Point

Change line 28 to use:

```
i < this.size
```

instead of

```
i < this.elements.length
```

Which tests would pass?

- ○ **Both** tests would pass
- ○ **Neither** test would pass
- ○ Only **testThreeAdds()** would pass
- ◉ Only **testFourAdds()** would pass

Q4.4 D

1 Point

Change line 27 to:

```
String[] expanded = new String[currentCapacity + 10];
```

Which tests would pass?

- ○ **Both** tests would pass
- ◉ **Neither** test would pass
- ○ Only **testThreeAdds()** would pass
- ○ Only **testFourAdds()** would pass

Q4.5 E

1 Point

Change line 14 to use:

```
i < this.size
```

instead of

```
i < this.elements.length
```

Which tests would pass?

- (●) **Both** tests would pass
- ( ) **Neither** test would pass
- ( ) Only **testThreeAdds()** would pass
- ( ) Only **testFourAdds()** would pass

Q4.6 F

1 Point

Change line 9 to:

```
this.elements = (E[])(new Object[4]);
```

Which tests would pass?

- ( ) **Both** tests would pass
- ( ) **Neither** test would pass
- ( ) Only **testThreeAdds()** would pass
- (●) Only **testFourAdds()** would pass

Q5 Interfaces

7 Points

Consider this class and interface hierarchy:

```
interface AnotherADT<T> {
  boolean methodOne();
  int methodTwo(T value);
}
class ImplementationAlpha<T> implements AnotherADT<T> {
  /* questions about this class body below */
}
class ImplementationBeta<T> implements AnotherADT<T> {
  /* questions about this class body below */
}
```

Assume that both **ImplementationAlpha** and **ImplementationBeta** successfully implement the **AnotherADT** interface.

Answer **true** or **false** for each of the following:

Q5.1 A

1 Point

ImplementationAlpha and **ImplementationBeta** must define at least two methods with the same name and parameter list.

- (●) true
- ( ) false

Q5.2 B

1 Point

This statement compiles successfully:

```
AnotherADT b = new ImplementationBeta();
```

- (●) true
- ( ) false

Q5.3 C

1 Point

Any field declared in **ImplementationAlpha** must be declared in **ImplementationBeta** with the same name and type

- ○ true
- ◉ false

Q5.4 D

1 Point

This statement compiles successfully:

```
AnotherADT a = new ImplementationAlpha();
a.methodOne();
```

- ◉ true
- ○ false

Q5.5 E

1 Point

Any method declared in **ImplementationAlpha** must be declared in **ImplementationBeta** with the same name and parameter list

- ○ true
- ◉ false

Q5.6 F

1 Point

This statement compiles successfully:

```
ImplementationAlpha a = new AnotherADT();
```

○ true

◉ false

Q5.7 G
1 Point

This statement compiles successfully:

```
AnotherADT s = new AnotherADT();
```

○ true

◉ false

Q6 LinkedList
5.5 Points

Consider these classes and tests for LList. We assume that there is a dummy node for the linked list.

```
class Node<T> {
    T data;
    Node<T> next;
    public Node(T item, Node<T> n) { this.data = item; this.next = n; }
}
class LList<T> {
    public final Node<T> front;
    public LList() {
        this.front = new Node<T>(null, null);
    }
    public void prepend(T item) {
        this.front.next = new Node<T>(item, this.front.next);
    }
    public void removeFirst() {
        // Write your answer in the space below
    }
    public T getFirst() { return this.front.next.data; }
    /* Return the first index where the item appears in
          the list, or -1 otherwise */
    public int find(T item) {
        Node<T> current = this.front;
        int index = 0;
        while(current.next != null) {
            if(item.equals(current.next.data)) { return index; }
            current.next = current.next.next;
            index += 1;
        }
        return -1;
    }
}



public class TestLList {
    @Test
    public void testLList() {
        LList<String> sl = new LList<String>();
        sl.prepend("a");
        sl.prepend("b");
        sl.prepend("c");
        sl.prepend("d");
        assertEquals("d", sl.getFirst());
        sl.removeFirst();
        assertEquals("c", sl.getFirst());
        sl.removeFirst();
        assertEquals("b", sl.getFirst());
        sl.removeFirst();
        assertEquals("a", sl.getFirst());
    }
    @Test
    public void testFind() {
        /* fill in the answer below */
    }
}
```

Q6.1 A

2 Points

Fill in an implementation of **removeFirst()** that passes the given test (and should pass other related tests as well).

You can assume that **removeFirst()** will only be called when there are elements present in the list.

Q6.2 B

2 Points

The given version of **find()** is buggy. Write the body of a test that demonstrates that **find()** is incorrectly implemented in the answer sheet.

Do not use null as an input to **find()**, or an element of a list, in your test.

Q6.3 C

0.5 Points

How many total **Node objects** are created in **testLList** (including in all the methods it calls) before the first use of **assertEquals()**?

Write your answer as a number (no extra spaces).

5

Q6.4

1 Point

Given the implementation of **LList** with the methods presented, assume all the issues have been fixed, would **LList** be more appropriate for implementing a **Stack** or a **Queue**?

- (●) Stack
- ( ) Queue

Q7 DFS

3 Points

The mazes below were solved with a **stack worklist/DFS**. The **asterisks mark the path** from the finish back to the start by following previous references.

In the three mazes below, various orders have been chosen for adding the available neighbors to the worklist. Answer which order matches each maze solution.

For each, choose from the answers below. You may use the answers more than once. A reminder that on the page, West is left, East is right, North is up, and South is down

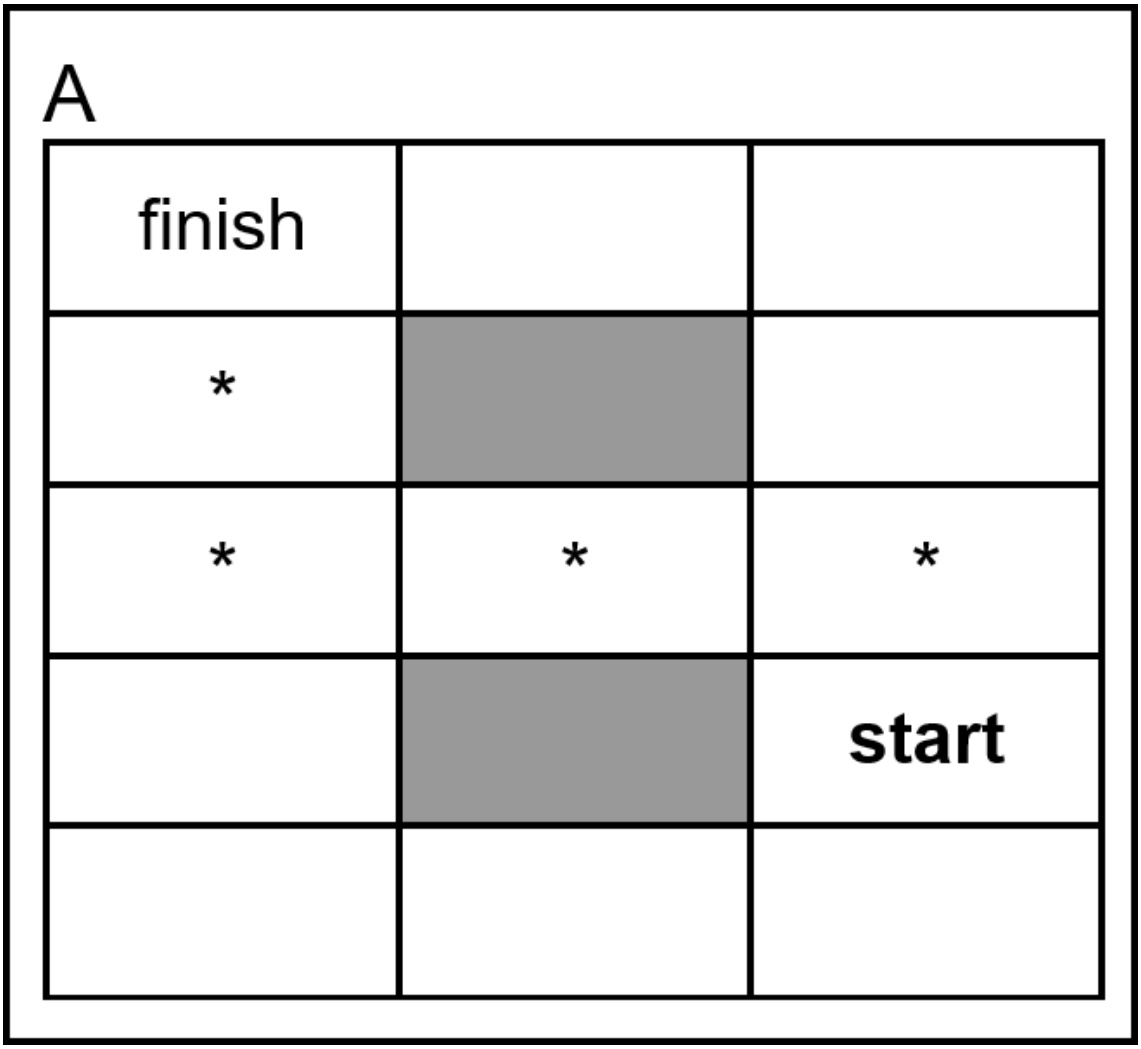
DFS/BFS Worklist Algorithm

```
initialize wl to be a new empty worklist (stack _or_ queue)
add the start square to wl
mark the start as visited
while wl is not empty:
  let current = remove the first element from wl (pop or dequeue)
  if current is the finish square
    return current
  else
    for each neighbor of current that isn't a wall and isn't visited IN SOME ORDER
      mark the neighbor as visited
      set the previous of the neighbor to current
      add the neighbor to the worklist (push or enqueue)

if the loop ended, return null (no path found)
```

Q7.1 A

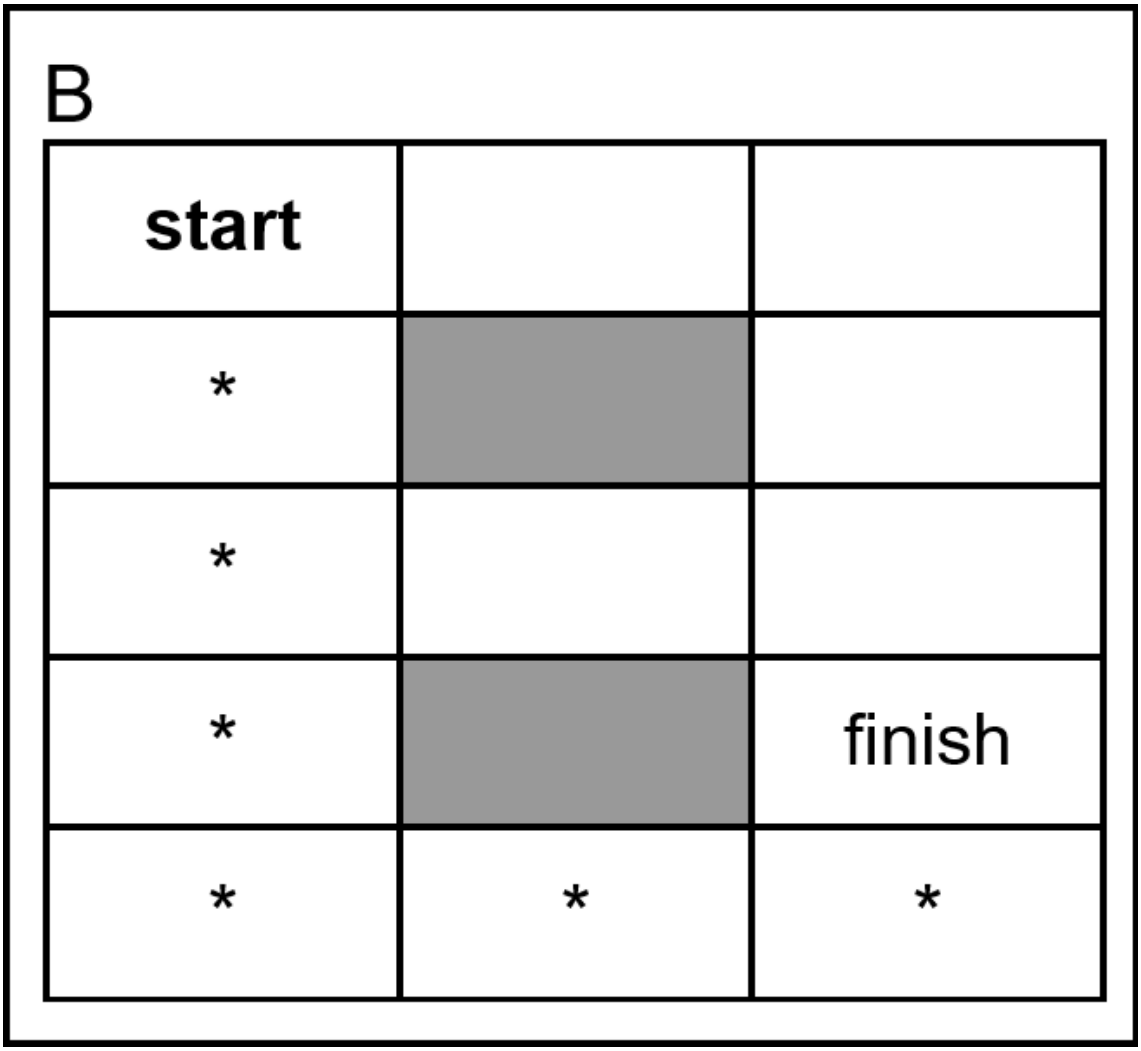
1 Point



Which order matches this maze's solution:

- (•) East then South then North then West
- ( ) North then South then East then West
- ( ) West then South then East then North
- ( ) North then West then South then East

Q7.2 B
1 Point

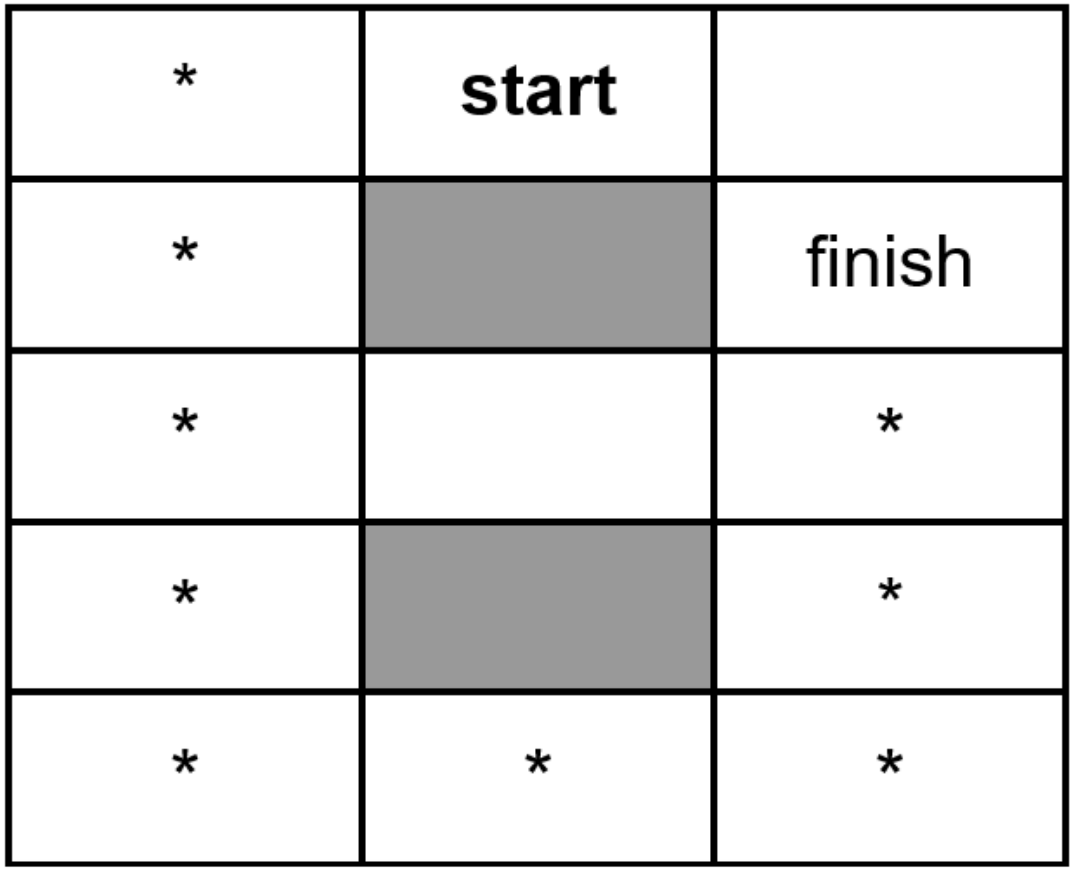


Which order matches this maze's solution:

- (●) East then South then North then West
- ( ) North then South then East then West
- ( ) West then South then East then North
- ( ) North then West then South then East

Q7.3 C
1 Point

C



Which order matches this maze's solution:

- (●) East then South then North then West
- ( ) North then South then East then West
- ( ) West then South then East then North
- ( ) North then West then South then East

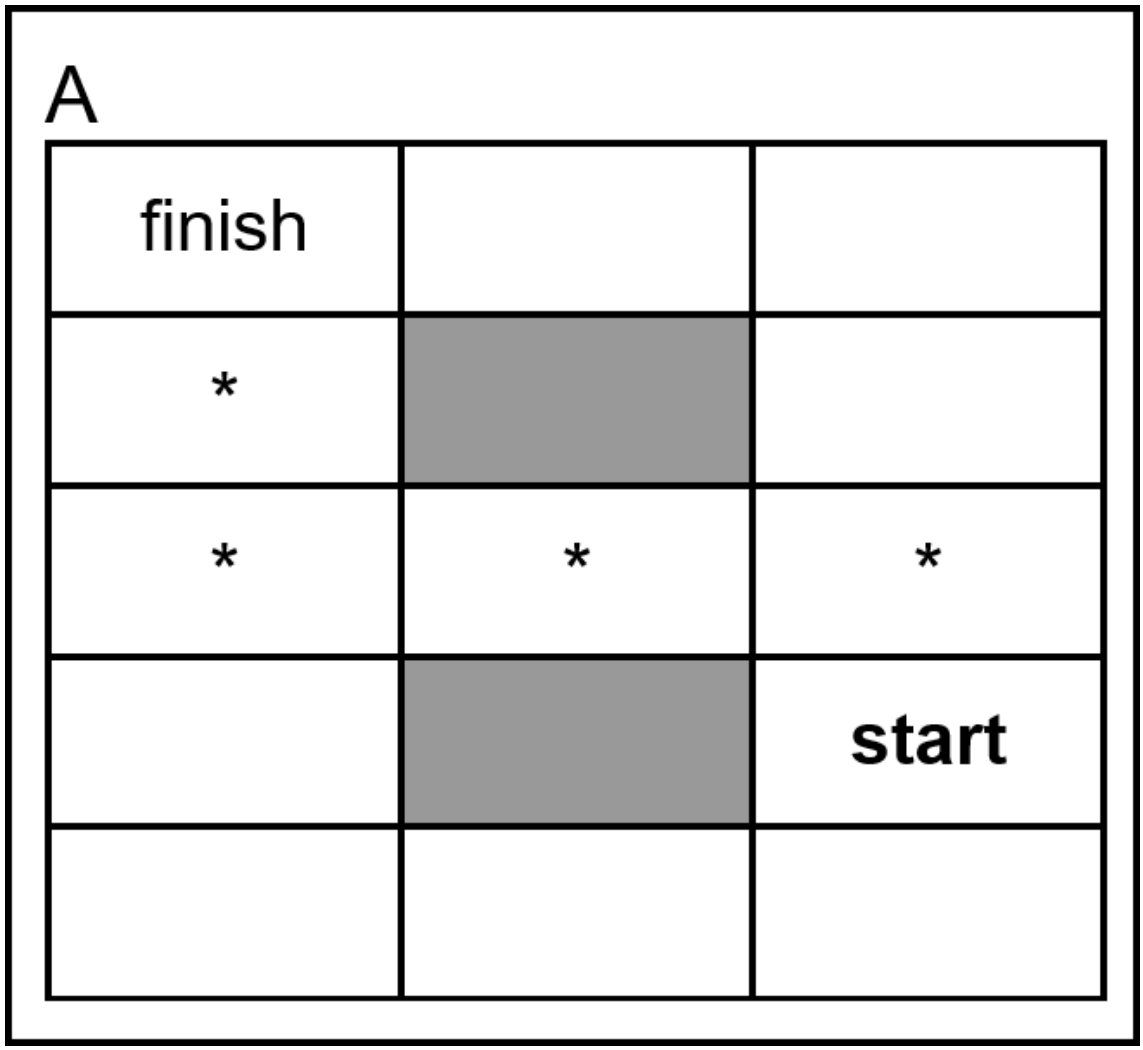
Q8 BFS

3 Points

For each maze A-C in question 7, could it have been the solution from a queue worklist/BFS for any of the given orderings?

Q8.1 A

1 Point

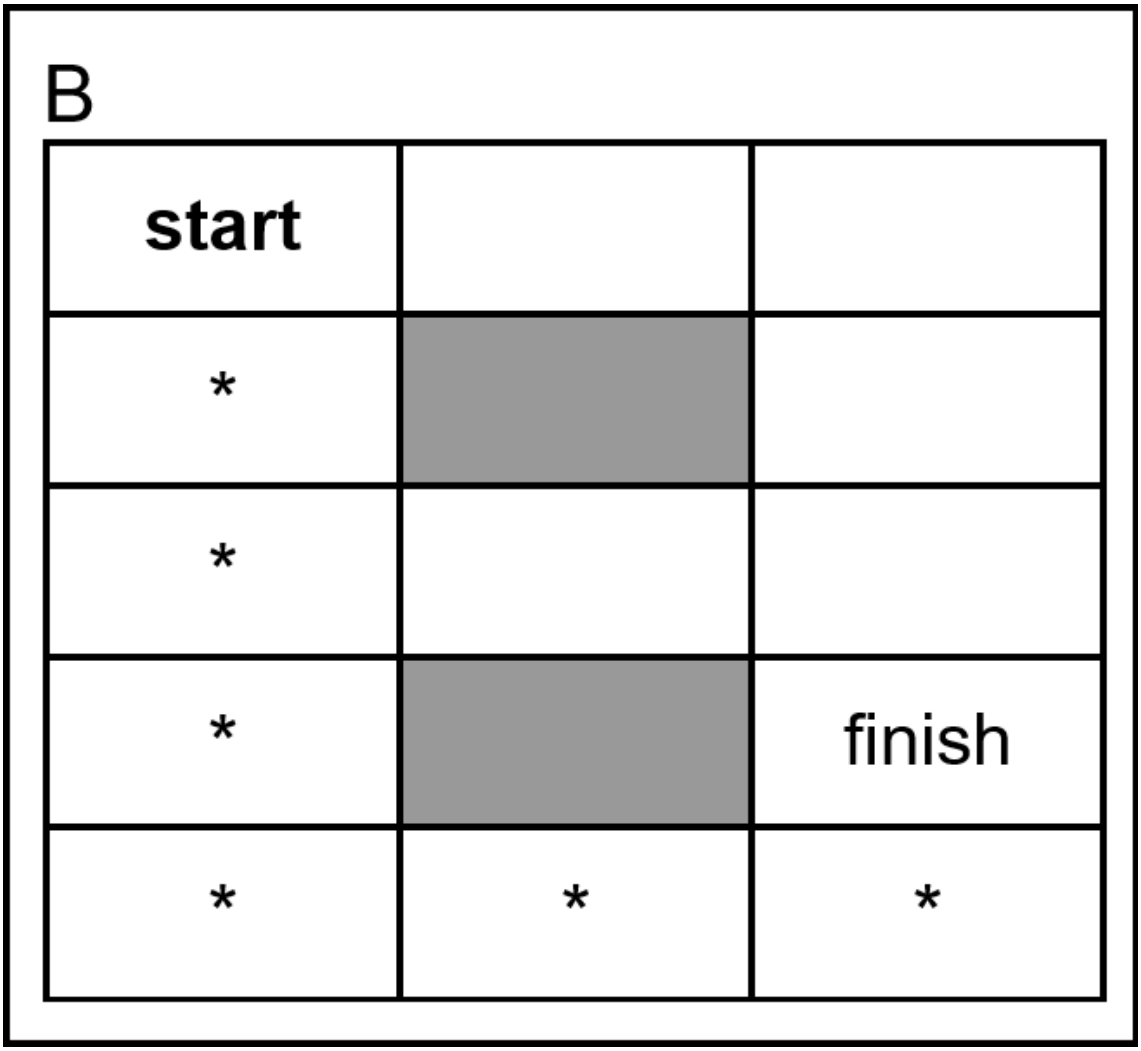


Could it have been the solution from a queue worklist/BFS for any of the given orderings?

- (●) Yes
- ( ) No

Q8.2 B

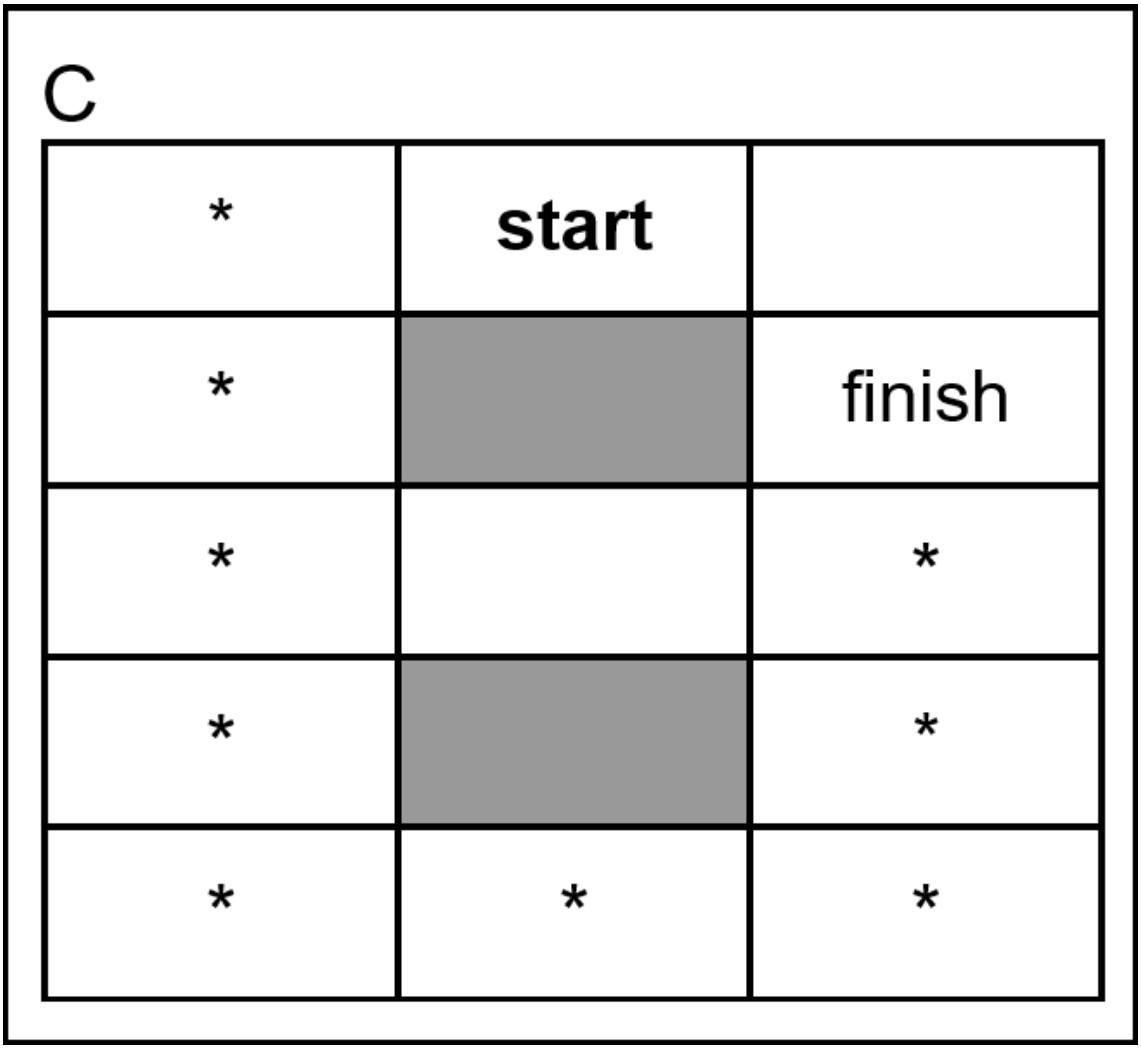
1 Point



Could it have been the solution from a queue worklist/BFS for any of the given orderings?

- ○ Yes
- ◉ No

Q8.3 C
1 Point



Could it have been the solution from a queue worklist/BFS for any of the given orderings?

- ○ Yes
- ◉ No