Iverson Exam 2015

Rob Kremer

Background

- Spearheaded by U of A
- 4 Questions
  - "Continents" (computations on a grid) - a,b,c,d, 10 pts
  - "Divisibility Testing" (FSM) - a,b,c,d, 10 pts
  - "Pebble Game" (permutations) - a,b,c,d, 10 pts
  - "Fractions" (loops, basic math) - 1 part, 3 pts
- VERY hard this year
- Calgary Average: 14.6/33; Median:14.5; High mark:24

Participation/Breakdown (Calgary)

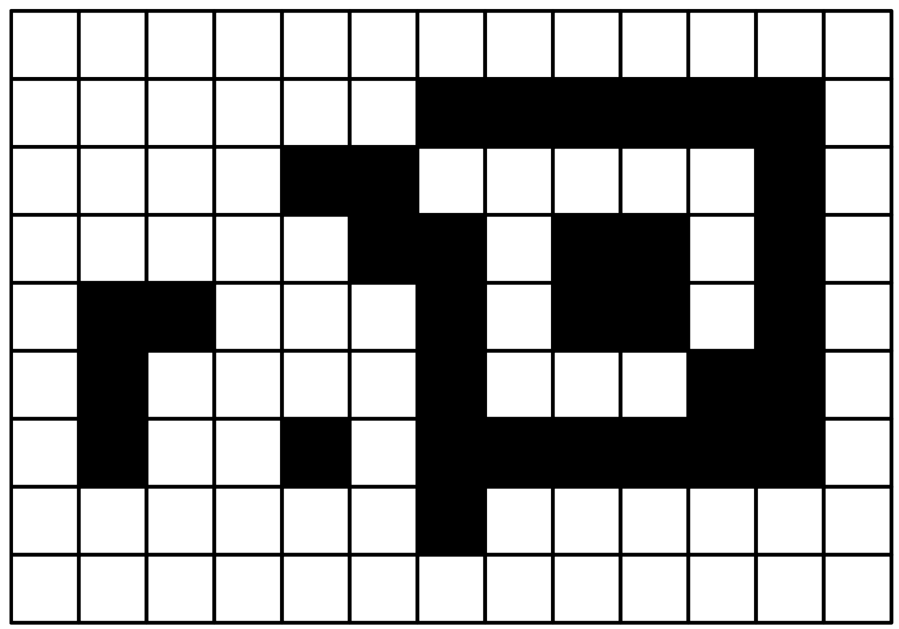
	Avg(33)	Q1(10)	Q2(10)	Q3(10)	Q4(3)	Count
All	14.6	7.0	4.3	2.5	0.8	35
All (Median)	14.5	7.0	5.0	2.0	0.0	35
Lord Beaverbrook High School	11.9	5.9	3.4	2.4	0.2	5
Sir Winston Churchill High School	17.3	8.2	4.7	3.7	1.0	6
Western Canada High School	13.3	6.1	4.4	2.2	0.7	17
William Aberhart High School	16.0	8.8	3.8	1.8	1.8	4
Bishop Grandin High School						1
Ernest Manning High School						1
John G. Diefenbaker High School						1

Awards

Firstname	Lastname	Grade	Name of High School	Total	Q1	Q2	Q3	Q4	Rank Total
David	Ng	grade_12	Sir Winston Churchill High School	24	8.5	6.5	6	3	1
Grant	Spink	grade_11	Ernest Manning High School	22	10	7	2	3	2
Duncan	Lo	grade_12	Sir Winston Churchill High School	21	10	5	6		3
Anthony	Tang	grade_12	Western Canada High School	21	10	7	4	0	3

Question 1: Continents

Pixel-world is a peculiar planet. It is a flat world that is divided into a grid of square cells. Each cells is completely covered by land or completely covered by water. Example: in the grid below, the dark cells represent land and the light cells represent water.



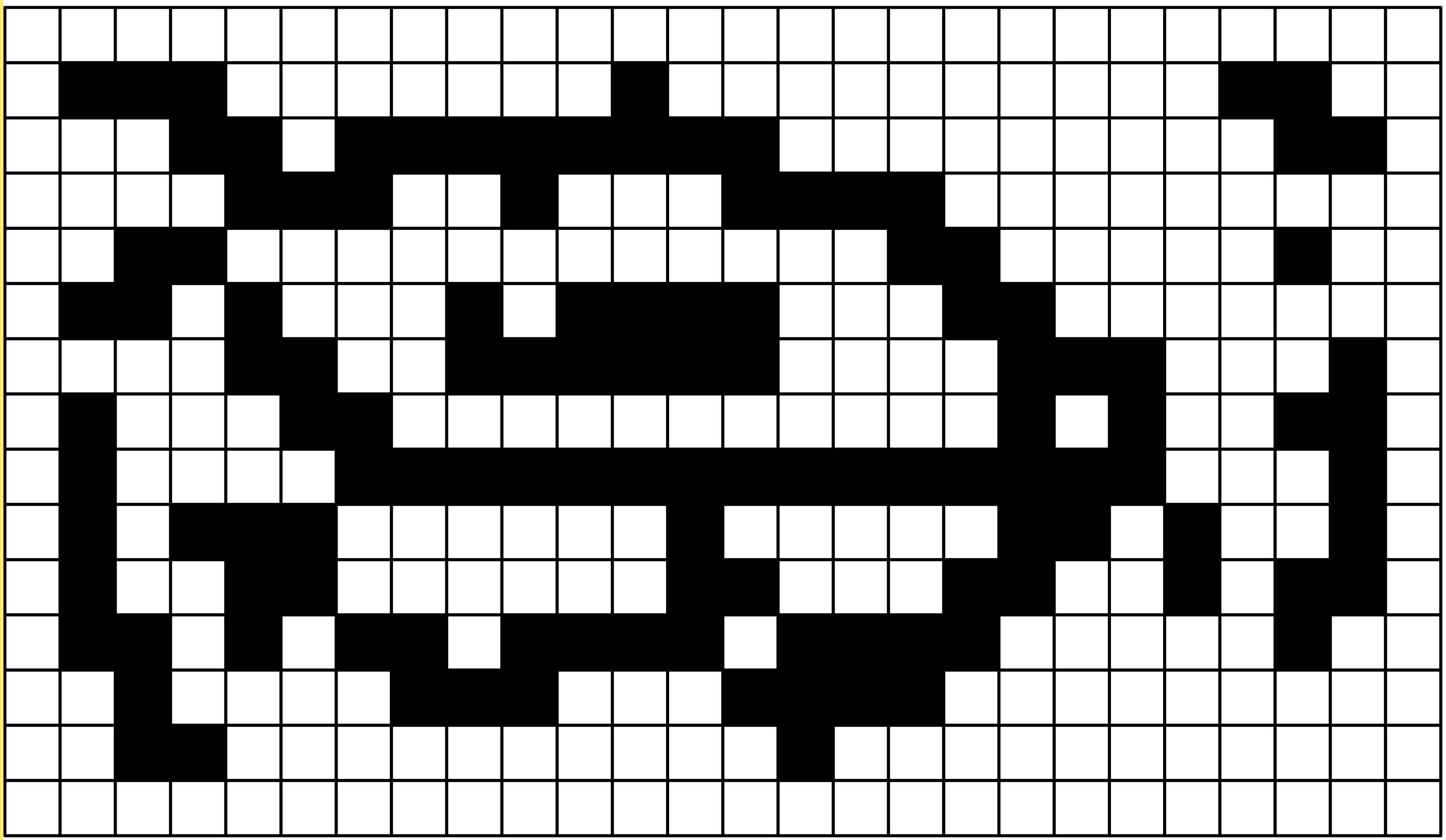
A continent is a collection of land cells, say C, such that

- it is possible to walk between any two cells in C by taking horizontal or vertical steps (no diagonal steps) and never entering a water cell.
- every land cell that can be reached in this way from a land cell in C is also in C.

The size of the continent C is the number of cells in C. Example: the grid shown above has 4 continents, with sizes 1, 4, 4, 24.

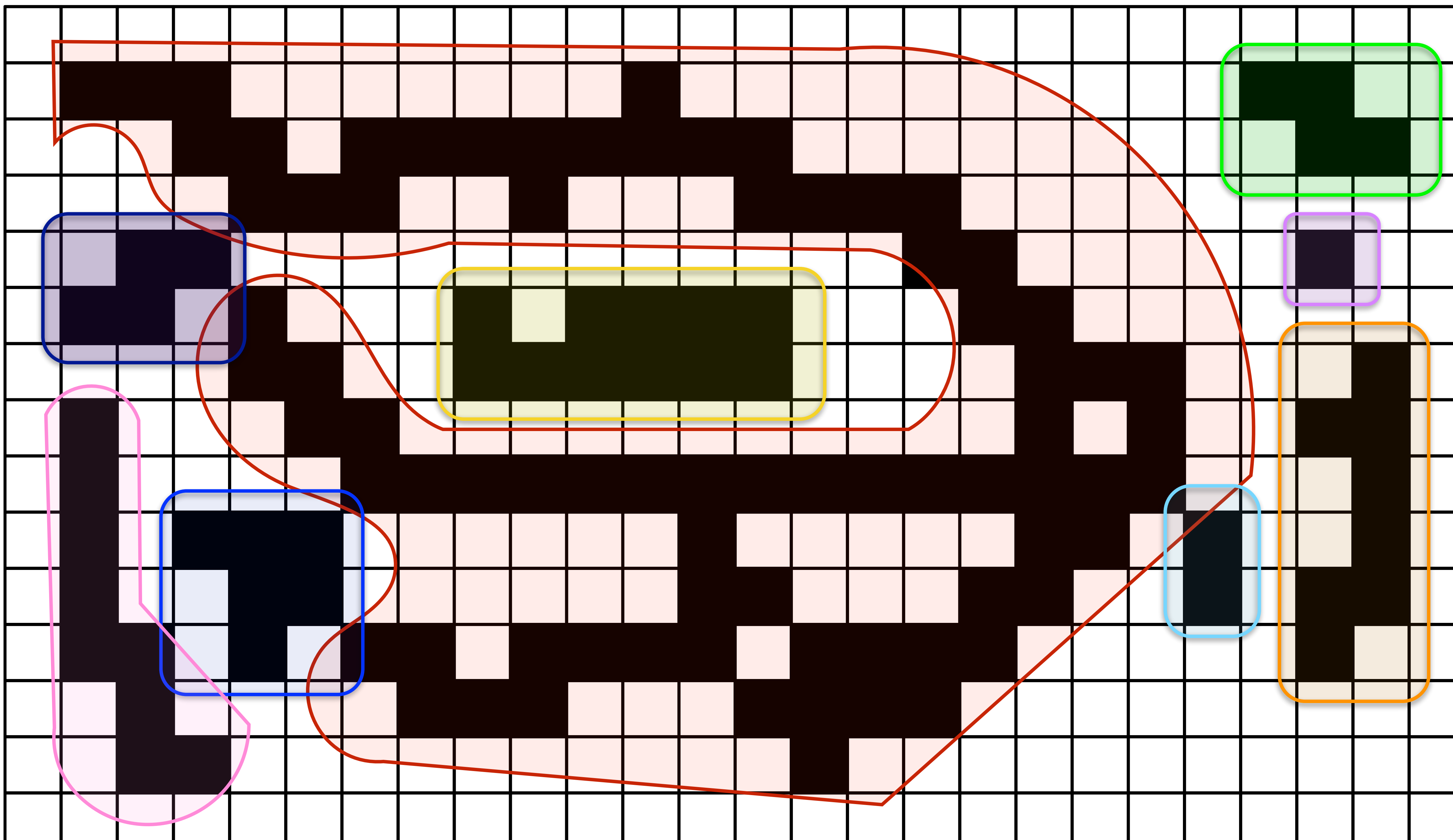
Question 1 a)

[2 marks] Give the number of continents in the following grid.



Question 1 a) (Solution)

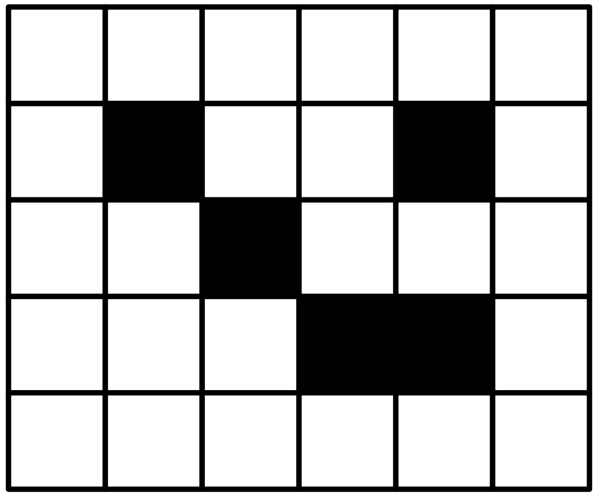
[2 marks] Give the number of continents in the following grid. **9**



Question 1 b)

[2 marks] Suppose, for each land cell, we already know the size of the continent that includes that cell. These sizes are stored in an array and the array is sorted.

Example: 1, 1, 1, 2, 2 is the sorted list of these sizes for the following grid. The size 2 appears twice because it is entered into the array once for each land cell in the continent.



The list below was obtained in this way from a grid with 30 land cells. How many continents does this grid have? Explain briefly.

1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4

Question 1 a) (Solution)

A continent of size `k` will be reported `k` times in the list. For $i \geq 1$, if we let a_i denote the number of times `i` appears in the list then the answer is

```
a1/1 + a2/2 + a3/3 + ...
```

In this case, the number of continents is

```
2/1 + 4/2 + 12/3 + 12/4 = 11
```

1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4

Question 1 c)

[3 marks] Write a function `count_continents(sizes, n)` where `sizes` is a sorted array of `n` integers, obtained from a grid with `n` land cells. The function should return the number of continents in the grid.

Question 1 c) (Solution 1)

Here is a Python implementation that follows the formula from the previous part.

```
def count_continents(sizes, n+1):
    tot = 0
    for i in range(1,n):
        tot += sizes.count(i)//i
return tot
```

Question 1 c) (Solution 1)

Here is a Python implementation that follows the formula from the previous part.

```
def count_continents(sizes, n+1):
    tot = 0
    for i in range(1,n):
        tot += sizes.count(i)//i
return tot
```

Does a complete walk of the array.

Question 1 c) (Solution 2)

The previous solution is a bit slow for large inputs because the `count()` method will scan sizes every iteration of the loop. The following is faster because it only walks through the list once. It takes advantage of the fact that all occurrences of a number will appear consecutively (because the list is sorted).

```
def count_continents(sizes, n):
    prev = 0
    tot = 0

    for j in range(1,n):
        if sizes[j] != sizes[prev]:
            tot += (j-prev)//sizes[prev]
            prev = j
    tot += (n-prev)//sizes[prev]
    return tot
```

Question 1 c) (Solution 2)

An even more efficient version that doesn't even visit every indici:

```
def count_continents(sizes, n):
    i = 0
    tot = 0
    while i<n:
        tot += 1
        i += sizes[i]
    return tot
```

1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3

Question 1 d)

[3 marks] We represent a grid by specifying two positive integers, `rows` and `columns` — indicating the grid size — together with a two-dimensional array `grid`. For $1 \leq r \leq rows$ and $1 \leq c \leq columns$, `grid[r][c]` is `0` if the cell at location `(r, c)` is water, and is `1` if that cell is land.

Write a function `continent_size(r, c, rows, columns, grid)` that returns the size of the continent that includes the cell at location `(r, c)`. If this is a water cell, return `0`. You may assume that every cell on the boundary of the grid is water.

Question 1 d) (Solution 1)

Mark off the cells in the continent one at a time. Initially, mark the cell at coordinate `(r,c)`. While there is marked land cell that is adjacent to an unmarked land cell, then mark that unmarked cell. When there are no more cells to mark, return the number of marked cells.

Question 1 d) (Solution 1)

```
#return the list of the four cells adjacent to cell (r, c)
def neighbours(r, c):
    return [(r-1, c), (r+1, c), (r, c-1), (r, c+1)]
def continent_size(r, c, rows, columns, grid):
    if grid[r][c] == 0:
        return 0
    #create a grid of 0s
    marked = [[0]*columns]*rows
    marked[r-1][c-1] = 1
    count = 1
    while true:
        found = false
        for cr in range(rows):
            for cc in range(columns):
                if marked[cr][cc]:
                    #examine the neighbours of the marked cell
                    for (nr, nc) in neighbours(cr, cc):
                        #mark the neighbour if is is an unmarked land cell
                        if not marked[nr][nc] and grid[nr][nc]:
                            marked[nr][nc] = 1
                            count += 1
                            found = true
        if not found:
            # the entire continent must be marked if we reach here
            return count
```

Question 1 d) (Solution 2)

The previous solution could be made more efficient. Notice that we only have to examine the neighbours of a marked cell once.

The following Python code exploits this fact. It maintains a list to examine that contains the cells that have been marked but have not yet had their neighbouring cells checked. It removes one cell from this list at a time and checks the neighbours of that cell. If any neighbour is a land cell that is not yet marked, it is marked and then added to the list. This way, every cell the continent has its neighbours checked exactly once.

Question 1 d) (Solution 2)

```
#return the list of the four cells adjacent to cell (r, c)
def neighbours(r, c):
    return [(r-1, c), (r+1, c), (r, c-1), (r, c+1)]
def continent_size(r, c, rows, columns, grid):
    if grid[r][c] == 0:
        return 0
    #create a grid filled with 0s
    marked = [[0]*columns]*rows
    marked[r-1][c-1] = 1
    #the list of cells that have been marked
    to_examine = [(r-1, c-1)]
    #the number of cells in the continent we have marked so far
    count = 1
    while len(to_examine) > 0:
        #remove something from the list of unprocessed tiles
        (cr, cc) = to_examine.pop()
        #examine the neighbours of this cell
        for (nr, nc) in neighbours(cr, cc):
            #if the neighbouring tile is an unmarked land tile
            #then mark it and add it to the list of tiles to process
            if marked[nr][nc] == 0 and grid[nr][nc] == 1:
                marked[nr][nc] = 1
                count += 1
                to_examine.append((nr, nc))
return count
```

Question 1 d) (Solution 3)

A very time- and heap- efficient and compact solution. Drawback: destroys the argument grid and is recursive (not so stack-efficient).

```
def cs3(r,c,rows,columns,grid):
  if grid[r][c]==0:
    return 0
  grid[r][c] = 0
  return 1 +\
         cs3(r+1,c,rows,columns,grid) +\
         cs3(r-1,c,rows,columns,grid) +\
         cs3(r,c+1,rows,columns,grid) +\
         cs3(r,c-1,rows,columns,grid)
```

Question 2: Divisibility Testing

In this question, we describe numbers in both decimal and binary form. A decimal number is subscripted with 10; a binary number is subscripted with 2. Example: $5_{10} = 101_2$.

You might know some divisibility tests for decimal numbers. Example: a number is divisible by 3 if and only if the sum of its decimal digits is divisible by 3. This rule does not hold for binary digits: $3_{10} = 11_2$ but the sum of its binary digits is 2_{10}.

We will use finite state machines (FSMs) for our tests. A FSM is a computational device that reads in a string of bits (0 or 1) and decides whether to accept that string. It has these features:

- states, which are depicted as labelled circles (a, b, c in the example below);
- for each state **x**, there are precisely two arcs that start at **x** and point to some other state (perhaps **x** again); one arc is labelled 0 and the other 1;
- one state is the start state and one is the accepting state; the start state has an arrow pointing to it labelled start; the accepting state has a thick border; the start and accepting states can be the same.

A computation with a FSM is simple to describe. A "current state" **v** is maintained which is initialized to be the start state. The input string is read one bit at a time, from left to right. When a bit **b** is read, the current state **v** is updated to be the state that is pointed to from **v** by the arc labelled **b**.

Question 2: Divisibility Testing

In this question, we describe numbers in both decimal and binary form. A decimal number is subscripted with 10; a binary number is subscripted with 2. Example: $5_{10} = 101_2$.

You might know some divisibility tests for decimal numbers. Example: a number is divisible by 3 if and only if the sum of its decimal digits is divisible by 3. This rule does not hold for binary digits: $3_{10} = 11_2$ but the sum of its binary digits is 2_{10}.

We will use finite state machines (FSMs) for our tests. A FSM is a computational device that reads in a string of bits (0 or 1) and decides whether to accept that string. It has these features:

- states, which are depicted as labelled circles (a, b, c in the example below);
- for each state `x`, there are precisely two arcs that start at `x` and point to some other state (perhaps `x` again); one arc is labelled 0 and the other 1;

  “one or more”
- one state is the start state and one is the accepting state; the start state has an arrow pointing to it labelled start; the accepting state has a thick border; the start and accepting states can be the same.

A computation with a FSM is simple to describe. A “current state” `v` is maintained which is initialized to be the start state. The input string is read one bit at a time, from left to right. When a bit `b` is read, the current state `v` is updated to be the state that is pointed to from `v` by the arc labelled `b`.

Question 2: Divisibility Testing

Once the entire input is processed, the FSM accepts the string if **v** is the accepting state, otherwise it rejects the string. We assume that the input string has at least one bit.

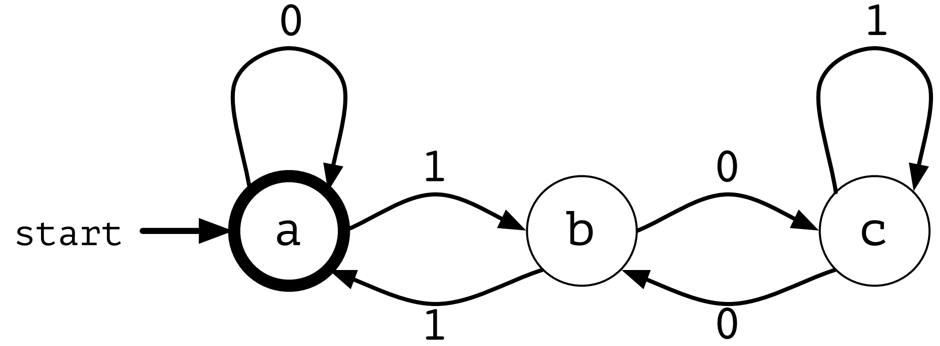


Figure 1: A finite state machine that accepts binary numbers divisible by 3. Here, **a** is both the start state and the accepting state.

Given a FSM, we can illustrate a computation by writing the sequence of states assumed by **v**, connecting consecutive labels with an arrow that labelled by the associated input bit. Example: for the FSM above, here is the computation for input string 110:

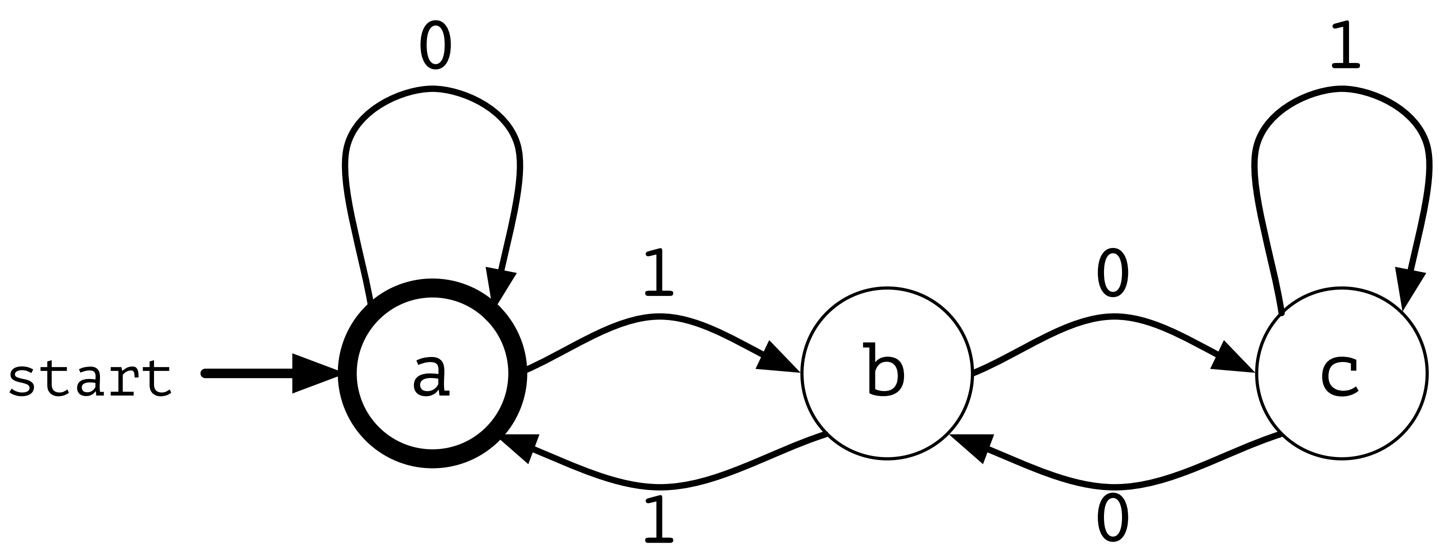
a –1→ **b** –1→ **a** –0→ **a**

and here is the computation for input string 100:

a –1→ **b** –0→ **c** –0→ **b**

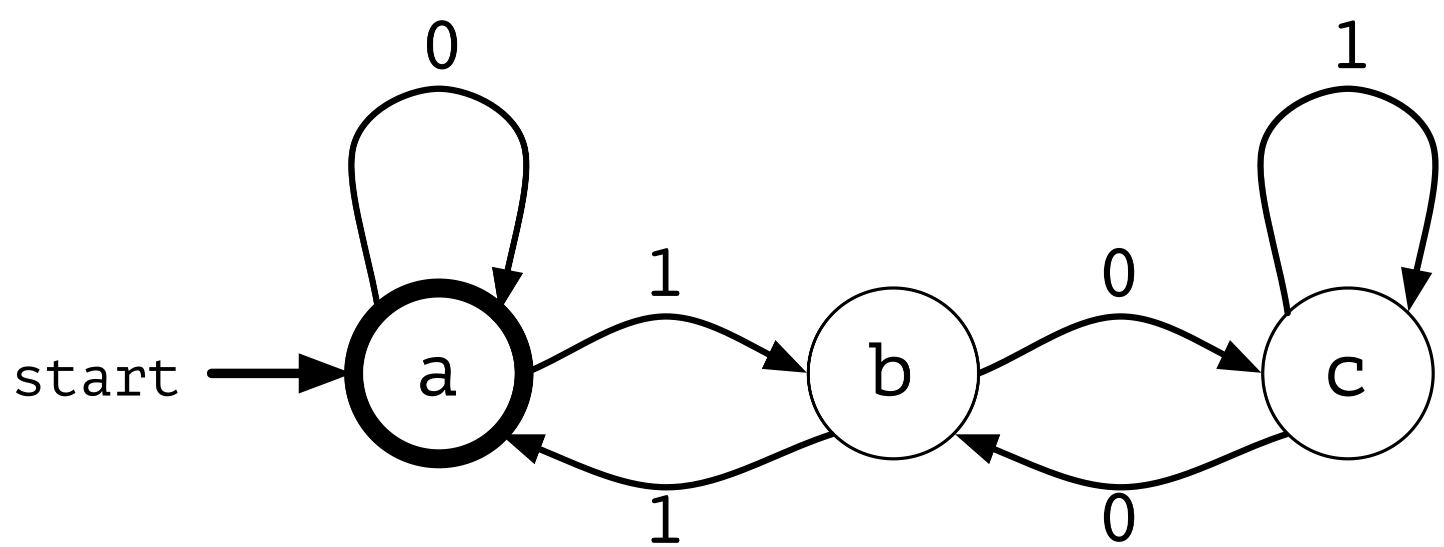
String 110 is accepted because the final state is the accepting state, but string 100 is rejected. In fact, this FSM accepts precisely the binary numbers that are divisible by $3_{10} = 11_2$. Notice that both 011 and 00 are accepted; leading 0s are allowed.

Question 2 a)



[2 marks] Illustrate the computation of the above FSM for (i) input string 1010010 and (ii) input string 1011010. For each string, state whether it is accepted.

Question 2 a) (Solution)



a –1→ **b** –0→ **c** –1→ **c** –0→ **b** –0→ **c** –1→ **c** –0→ **b**

The string 1010010 is rejected. Note: $1010010_2 = 82_{10}$ which is not divisible by 3.

a –1→ **b** –0→ **c** –1→ **c** –1→ **c** –0→ **b** –1→ **a** –0→ **a**

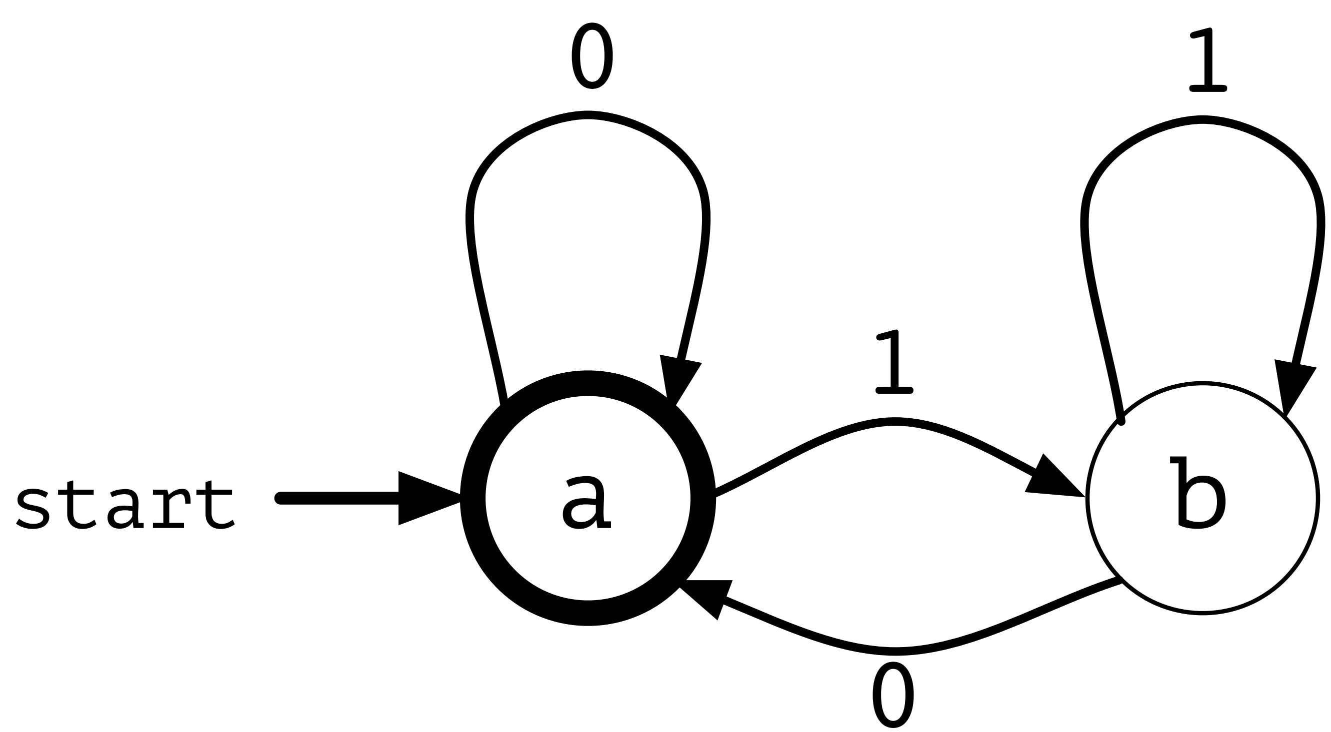
The string 1011010 is accepted. Note: $1011010_2 = 90_{10}$ which is divisible by 3.

Question 2 b)

[2 marks] Draw a FSM that accepts precisely the binary numbers that are divisible by 2_{10} (leading zeros are allowed).

Question 2 b) (Solution)

Even numbers are precisely those whose binary representation ends in a 0.



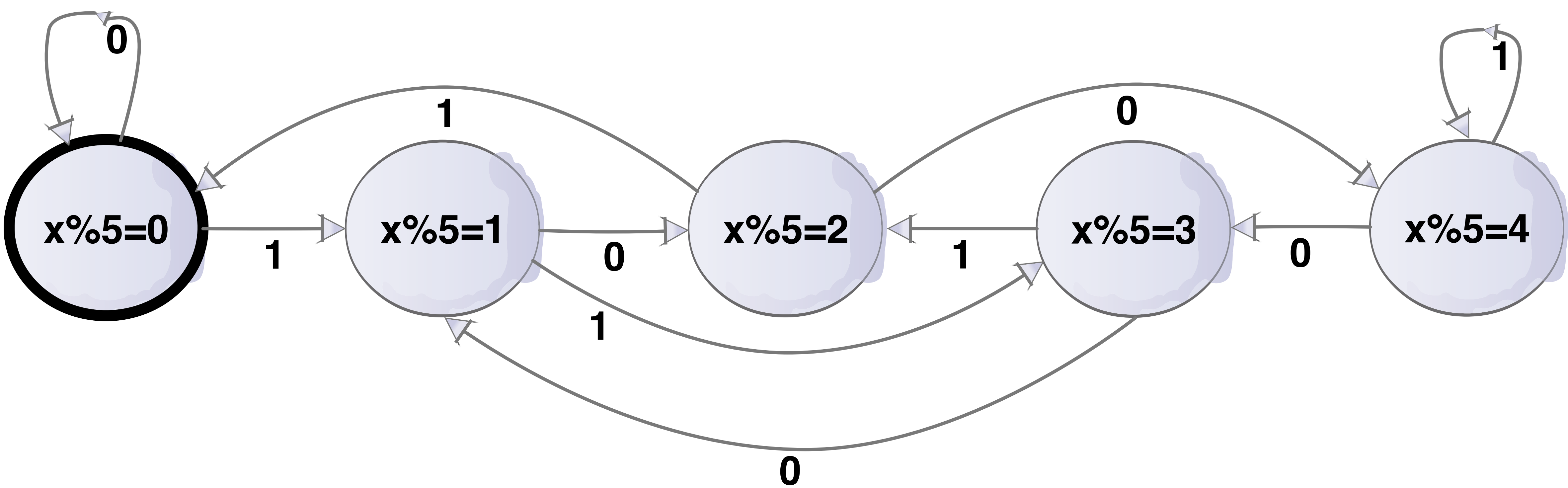
Question 2 c)

[3 marks] Draw a FSM that accepts precisely the binary numbers that are divisible by 5_{10} (leading zeros are allowed).

Question 2 c) (Solution)

Consider how the binary number changes as the bits are read in. If a 0 is read then the number is multiplied by 2. If a 1 is read then the number is multiplied by 2 and increased by 1. Example: 101 represents 5 and 1011 represents $2 \cdot 5 + 1 = 11$.

The states of the FSM will keep track of the value mod 5 of the binary number read so far and the arcs model how this value changes as the bits are read. Example: consider a binary number that is 3 mod 5. Suppose 1 is then appended to the number. The value mod 5 of the new number then becomes $2 \cdot 3 + 1 \equiv 2$.



Question 2 c) (Solution)

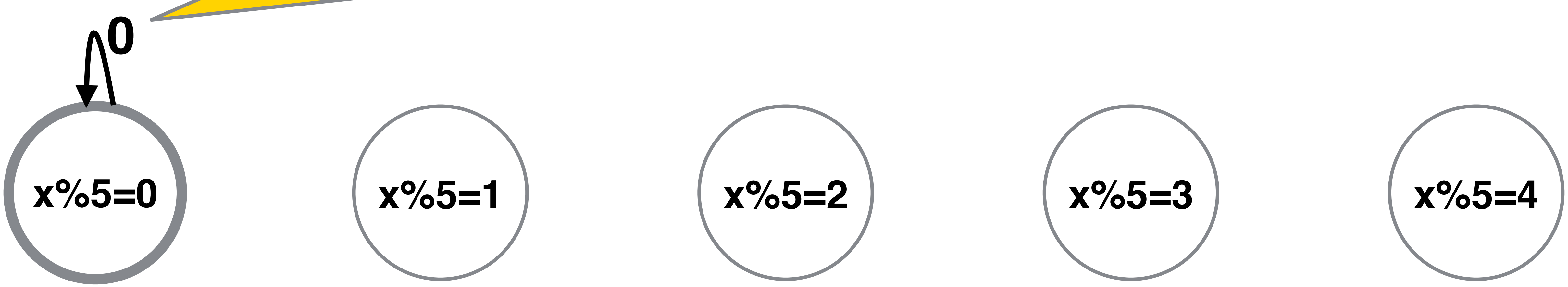
We only need to worry about the "mod 5's" of the number x. Therefore we only need 5 nodes: one for each possible modular number.

x%5=0 | **x%5=0** | **x%5=0** | **x%5=3** | **x%5=4**

The "mod 0" one is our only accept state (0 is divisible by 5).

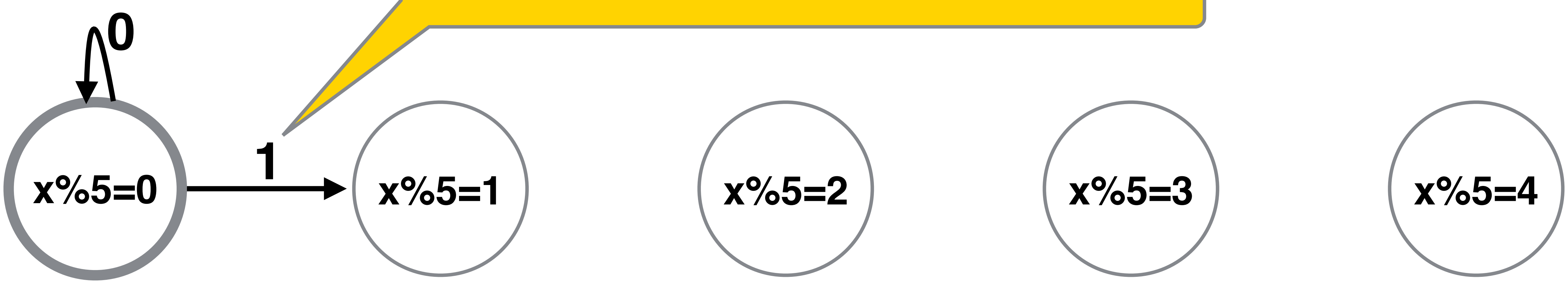
Question 2 c) (Solution)

Appending a zero doubles the number. If x was already divisible by 5 then doubling it will still be divisible by 5.

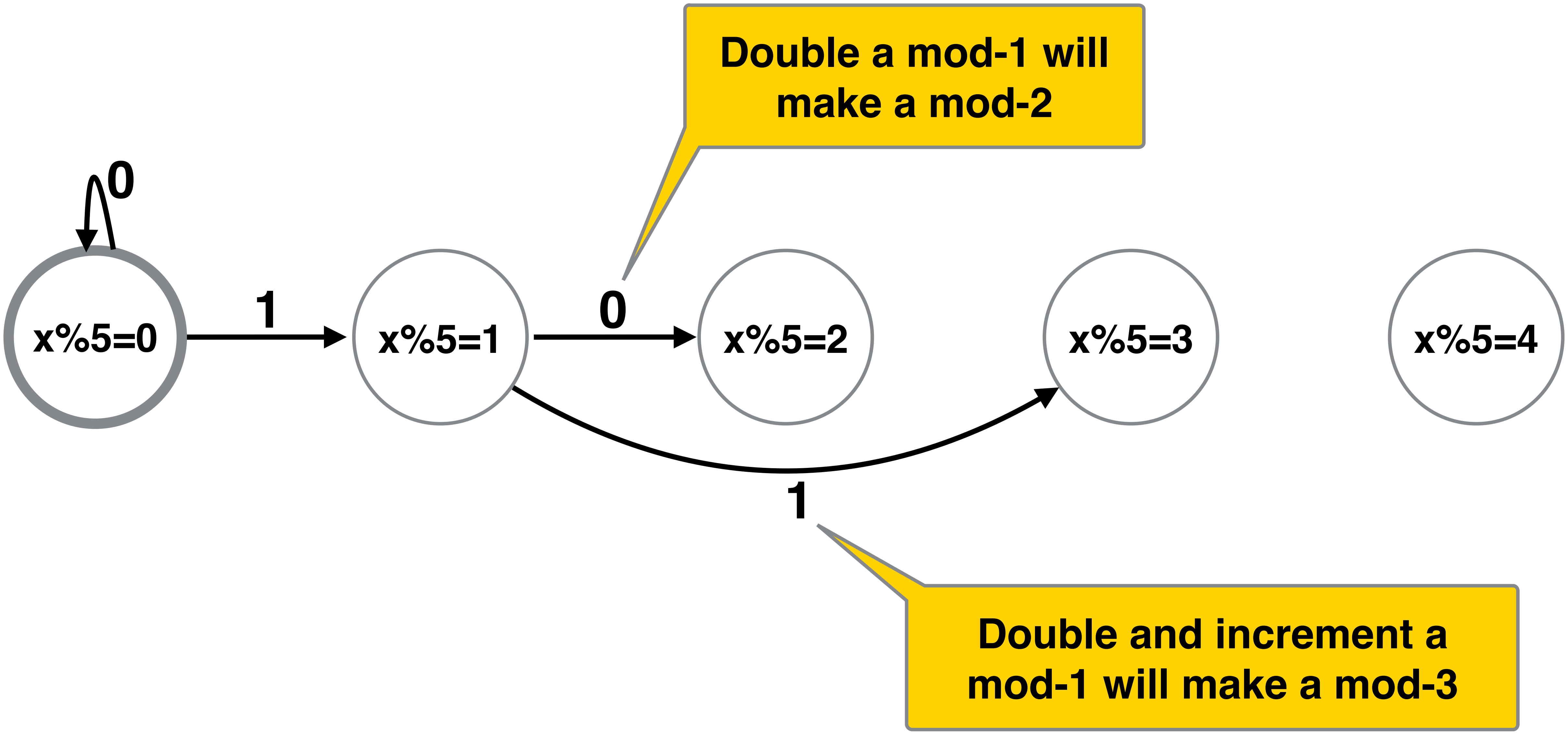


Question 2 c) (Solution)

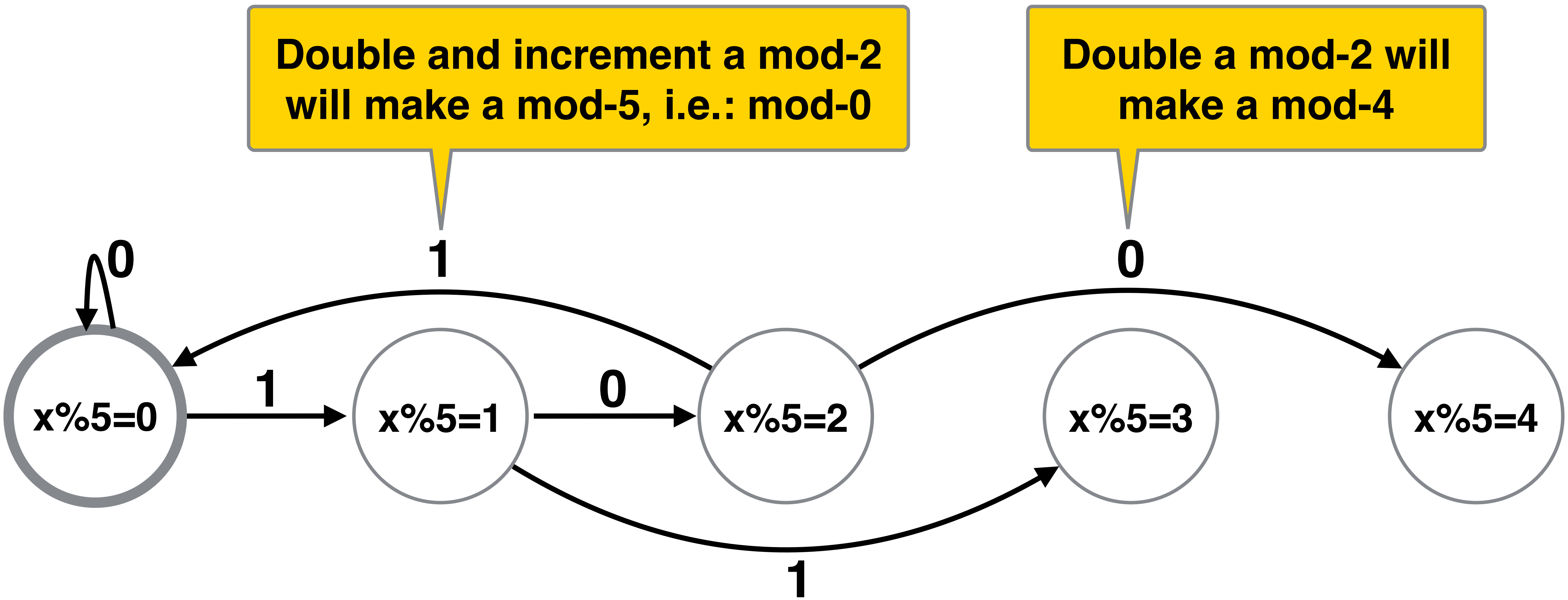
Appending a one doubles the number and adds 1. If x was already divisible by 5 then doubling and adding 1 will make it x%5=1.



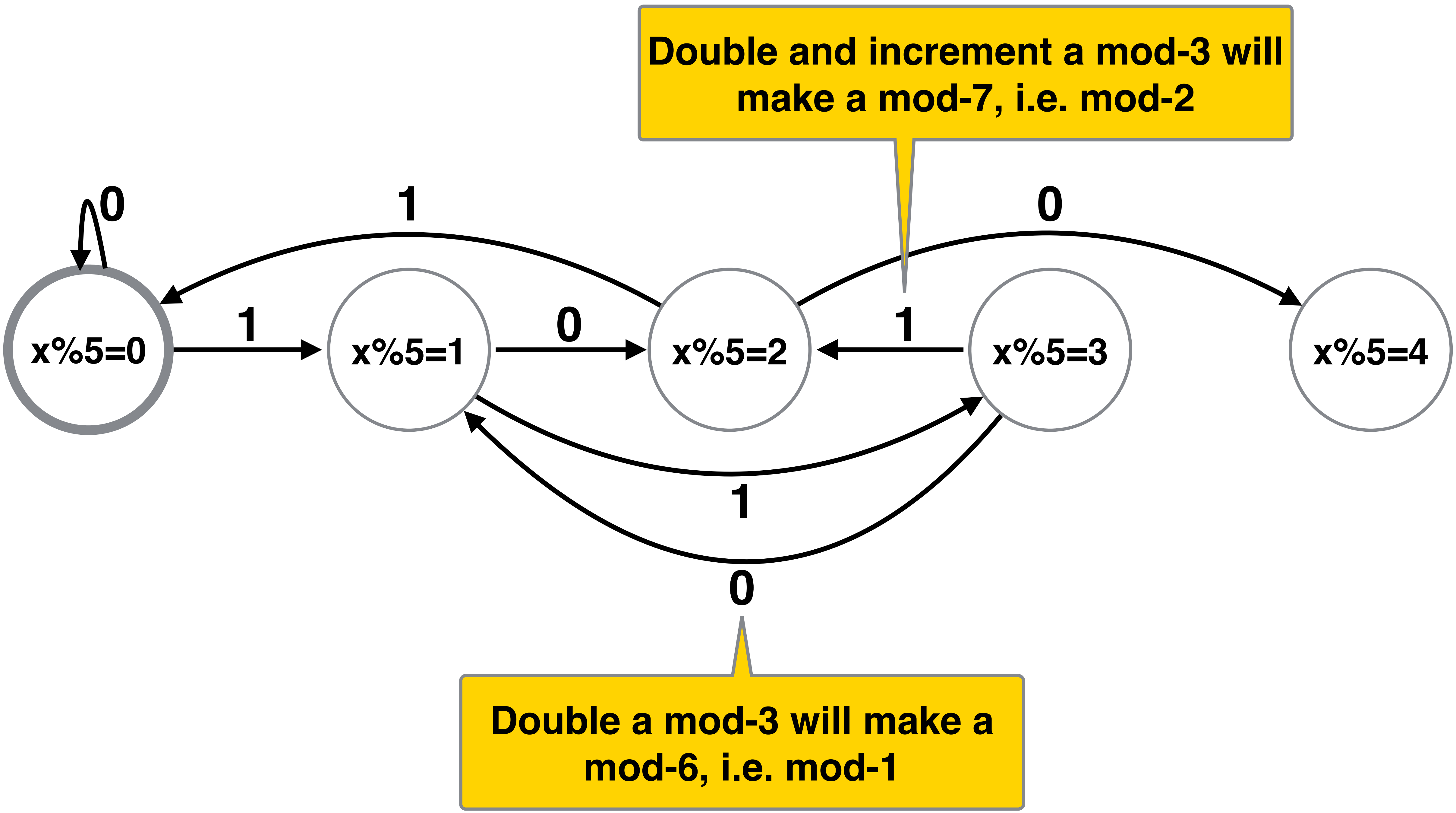
Question 2 c) (Solution)



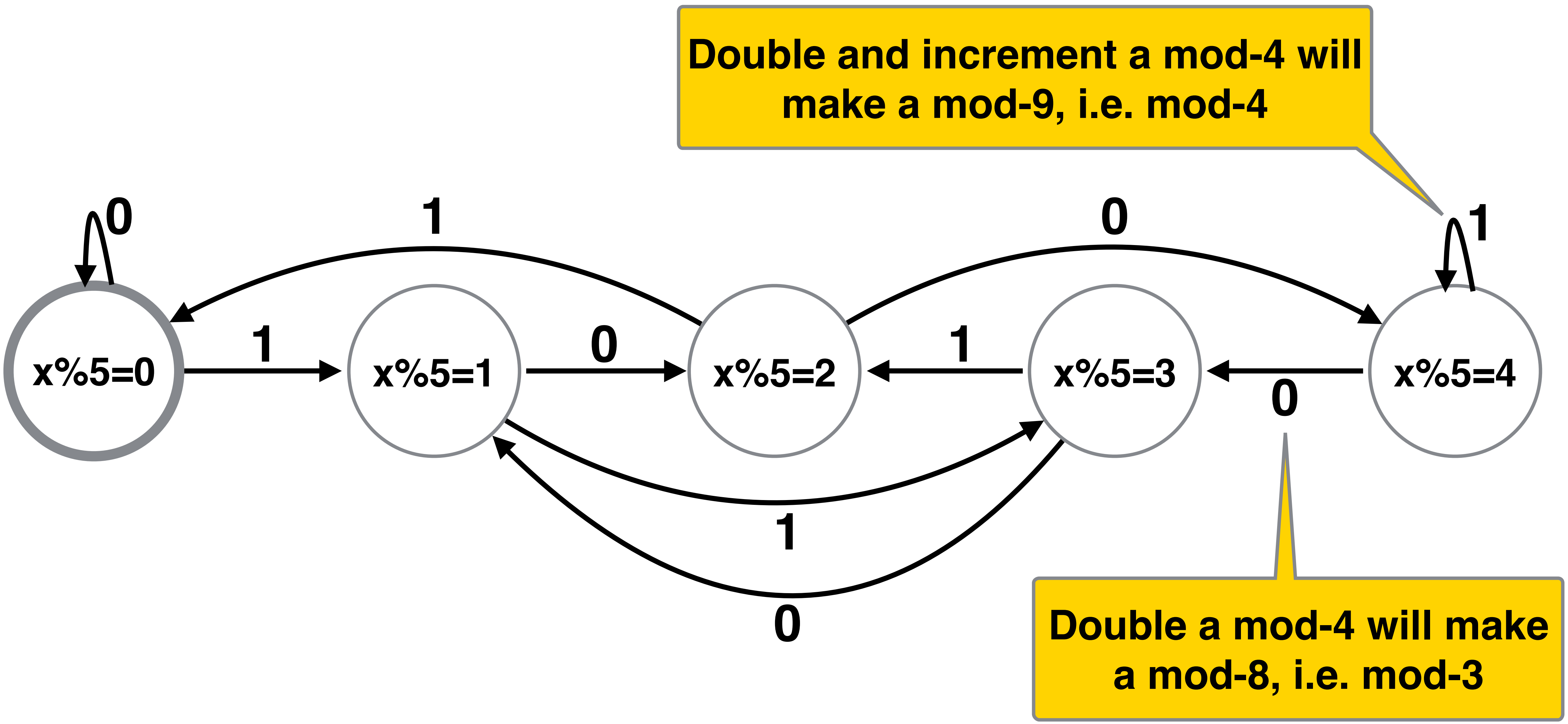
Question 2 c) (Solution)



Question 2 c) (Solution)



Question 2 c) (Solution)

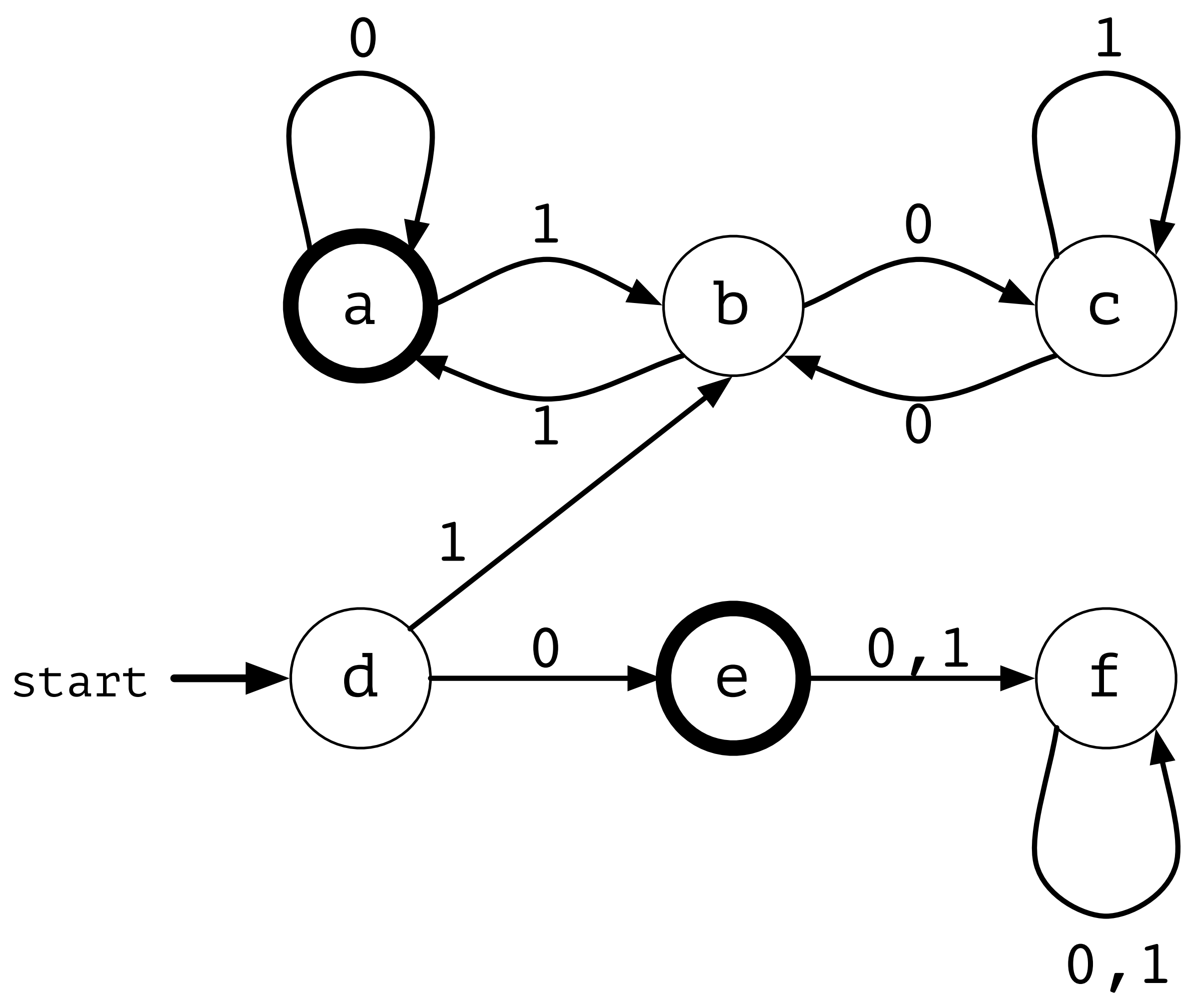


Question 2 d)

[3 marks] Draw a FSM from the description that accepts precisely binary numbers that are divisible by 3_{10} and either are the number zero or have a leading 1. Example: 11 and 0 are accepted; 011, 00, 10 are rejected.

Question 2 d) (Solution)

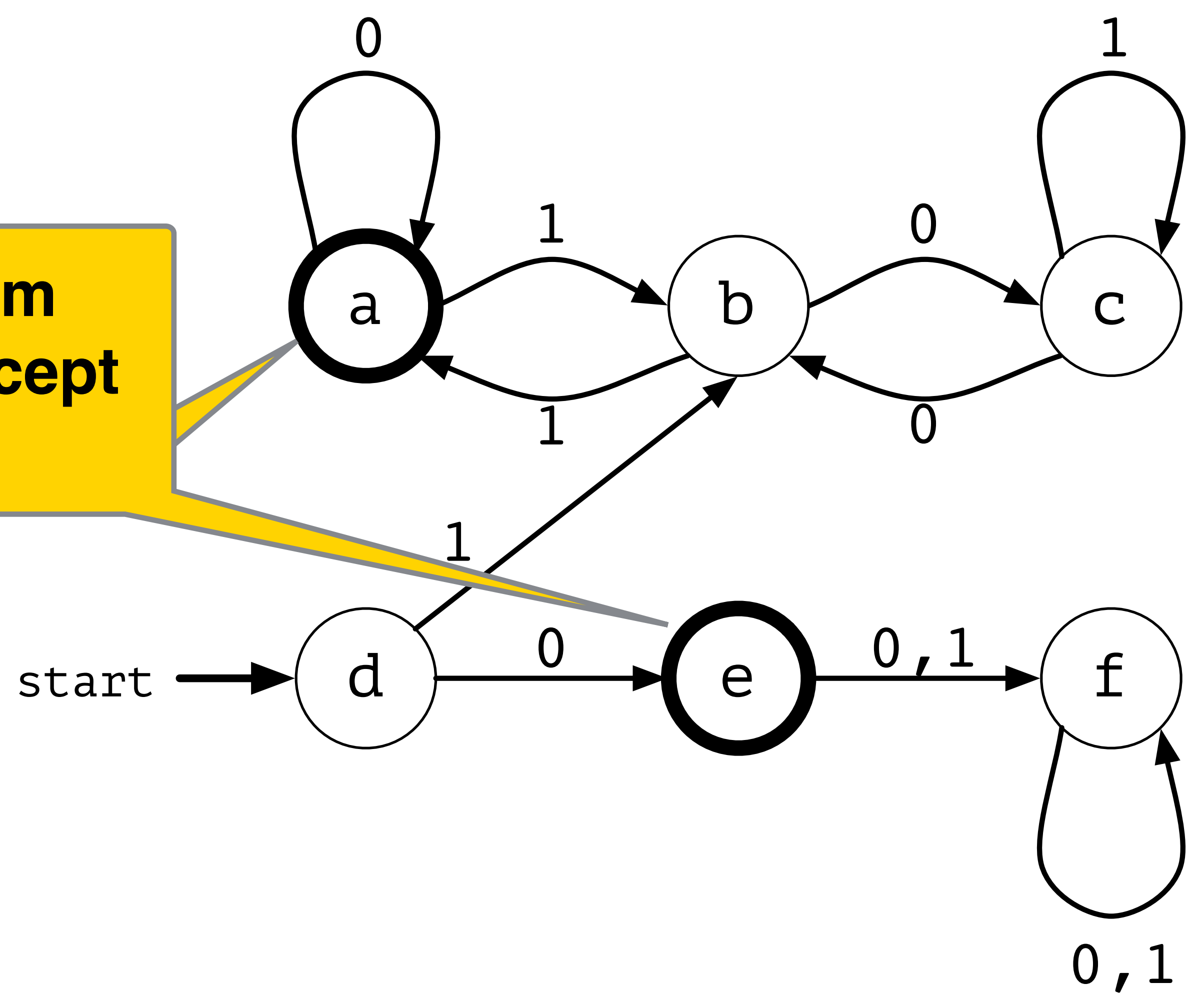
The only allowed string that starts with 0 is 0 itself. The following FSM will move into the "divisibility by 3" FSM if a leading 1 is read. If a leading 0 is read, the rest of the FSM (states e,f below) makes sure nothing else is read.



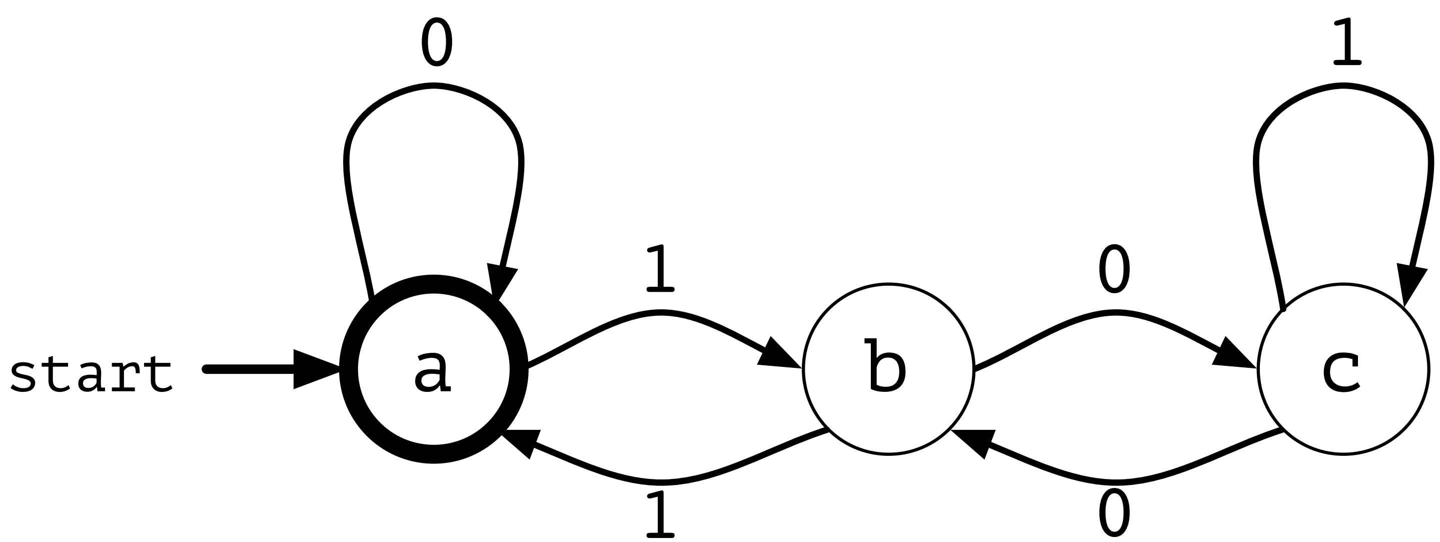
Question 2 d) (Solution)

The only allowed string that starts with 0 is 0 itself. The following FSM will move into the "divisibility by 3" FSM if a leading 1 is read. If a leading 0 is read, the rest of the FSM (states e,f below) makes sure nothing else is read.

This problem requires 2 accept states.

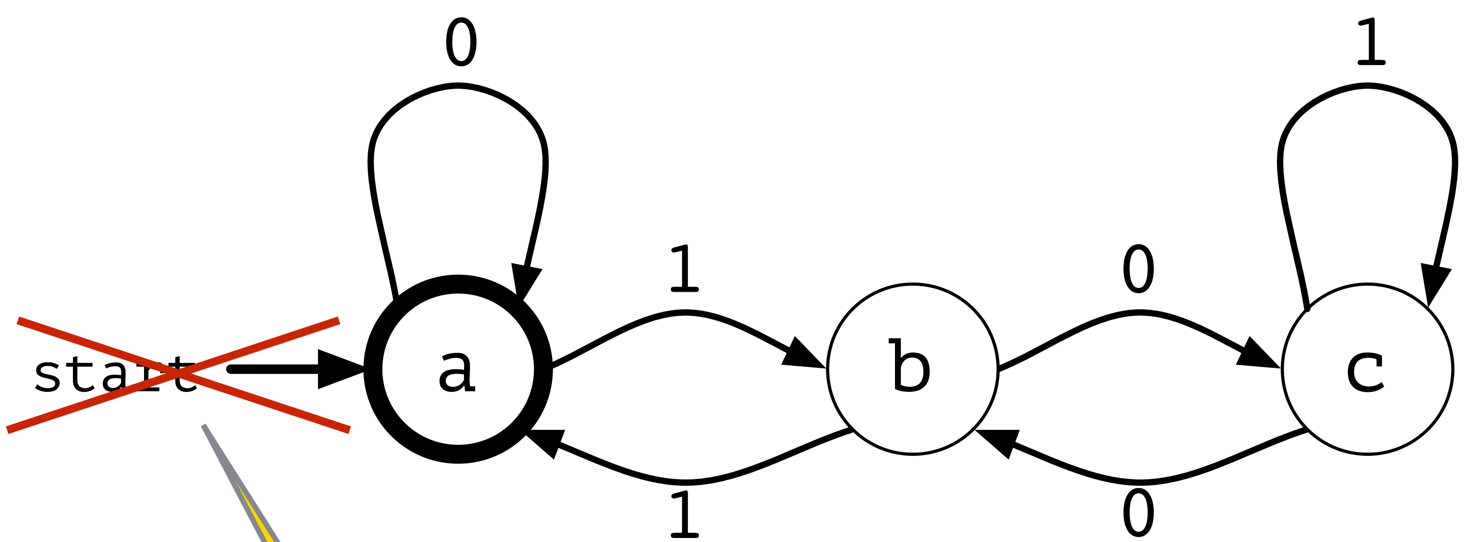


Question 2 d) (Solution)



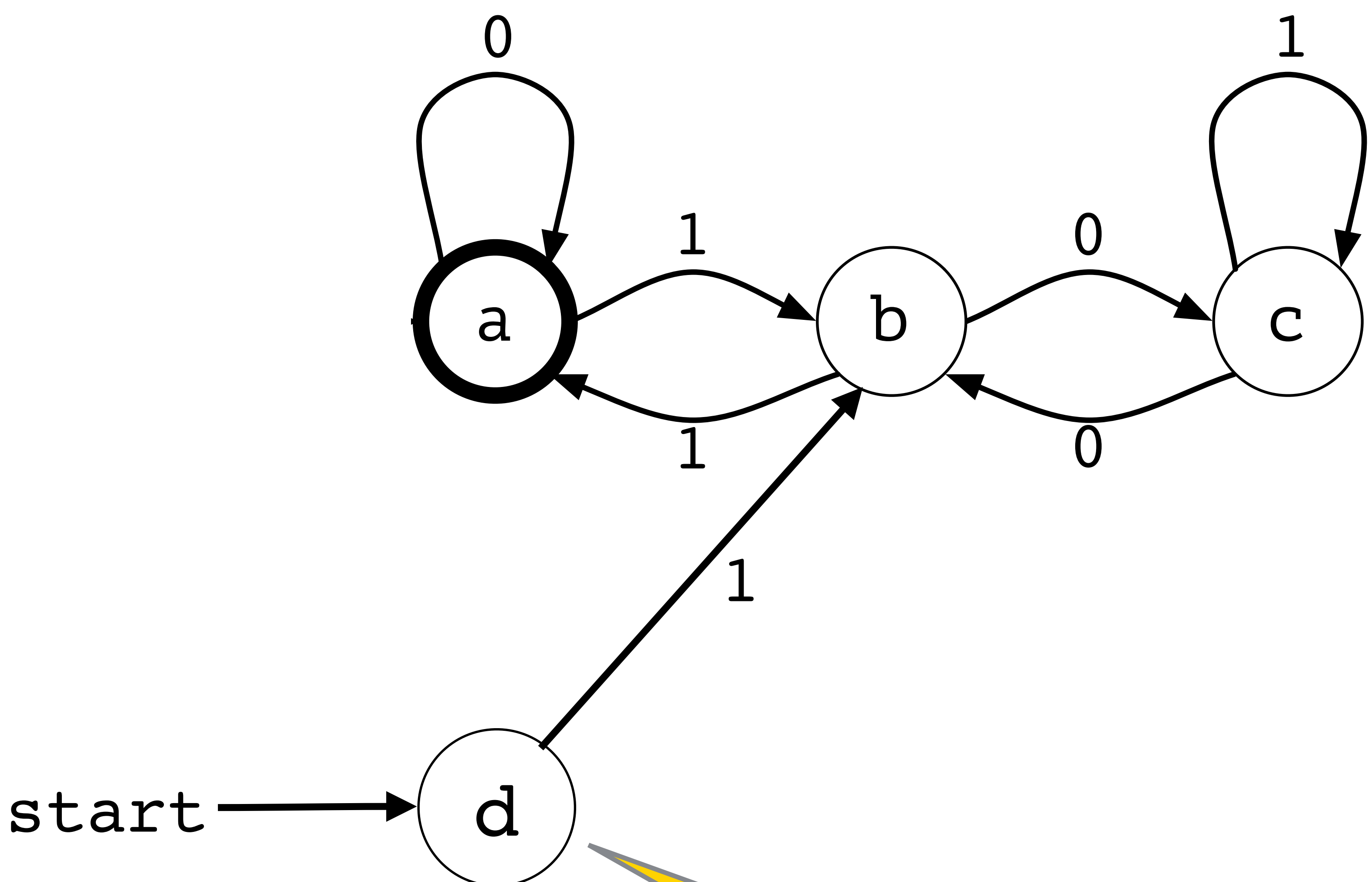
Start with what we know: the given "divisible by 3" FSM.

Question 2 d) (Solution)



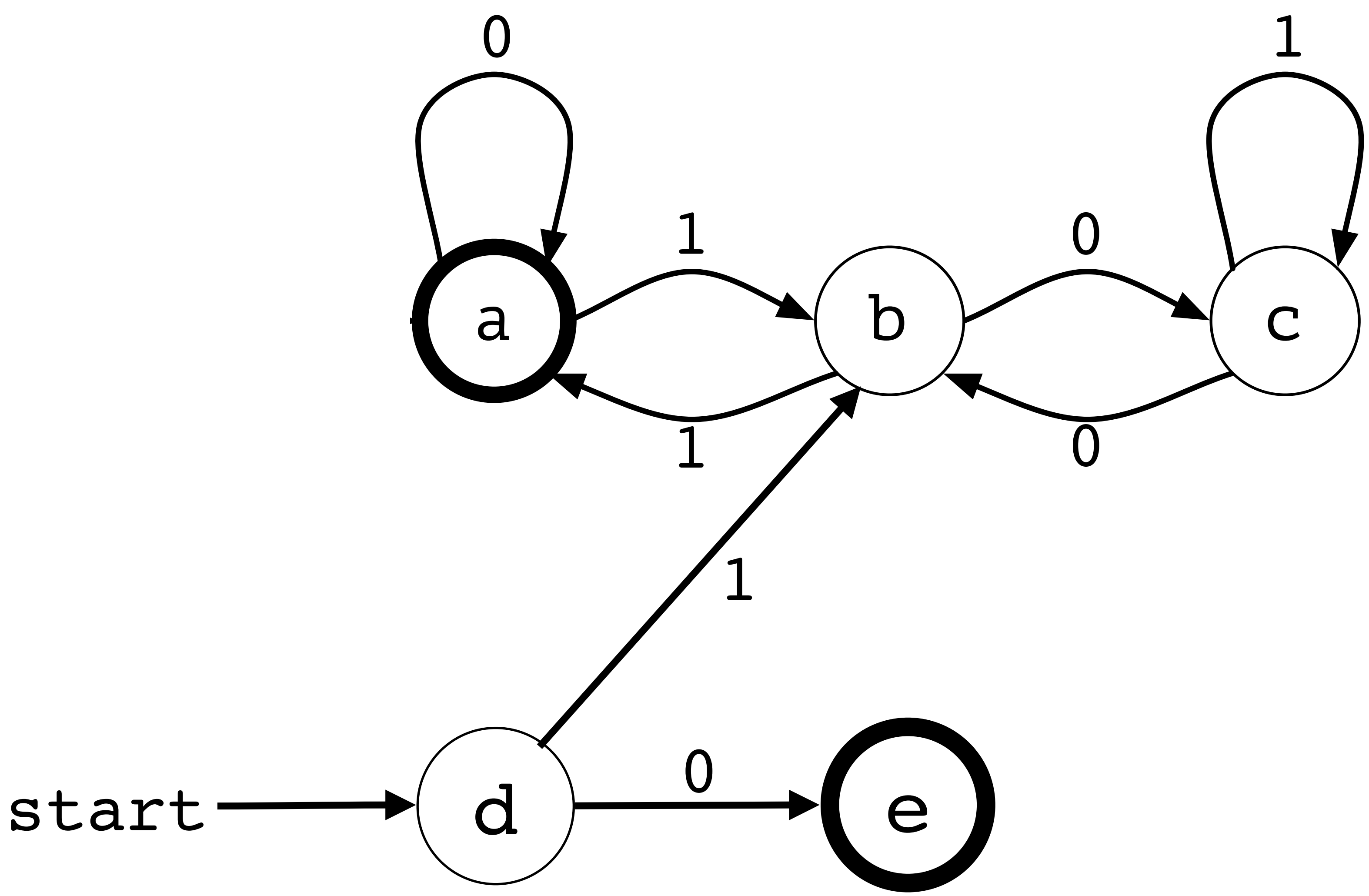
The old start location won't work because it accepts leading 0's. So just drop it

Question 2 d) (Solution)



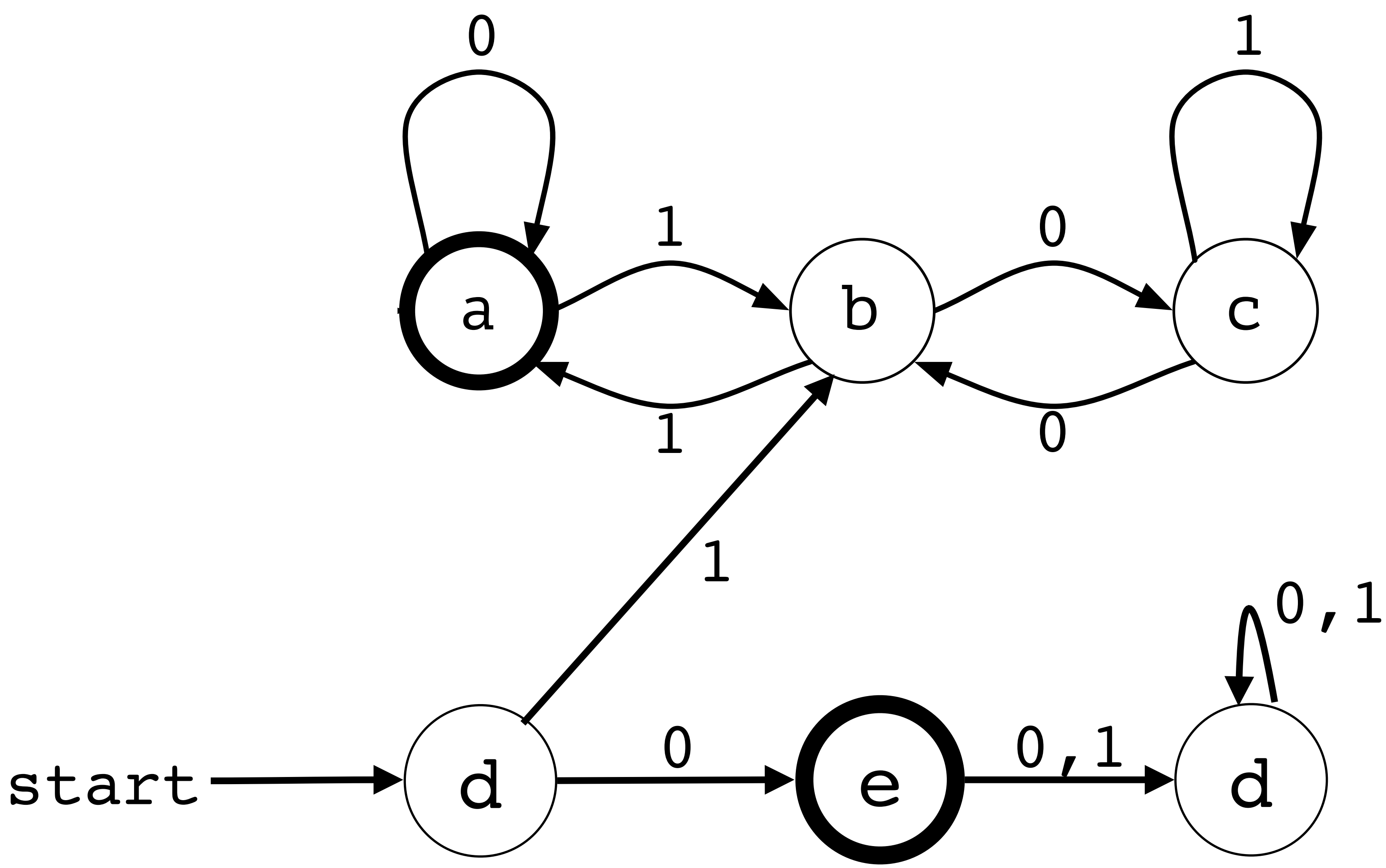
Add it a new start node — the 1 transition goes to the same spot as the old start node.

Question 2 d) (Solution)



But a 0 should be accepted, so we add that as an accept state.

Question 2 d) (Solution)

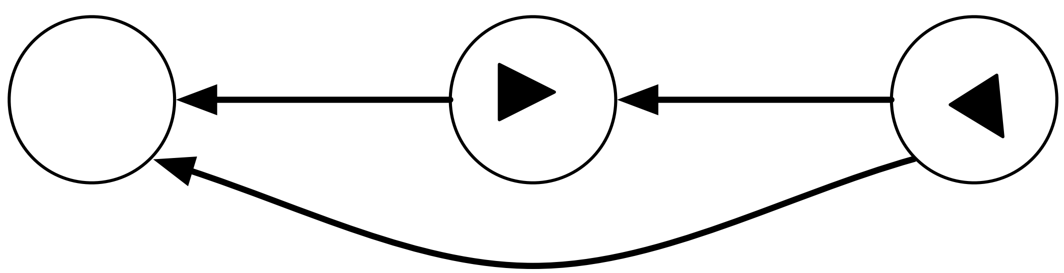


But we can't accept a 0 with anything else after that. So all "out" transitions go to complete sink.

Question 3: Pebble Game

Alice and Bob play a 2-player game. A number of pebbles are placed in various positions that are arranged horizontally. On a turn, a player moves one pebble and to the left. However, not all such moves are valid; valid moves are specified as part of the game. If no pebble can be moved, then the player whose turn it is loses and the other player wins.

Example: here is a game with 3 positions (circles) and 2 pebbles (triangles). The arcs show the valid moves.

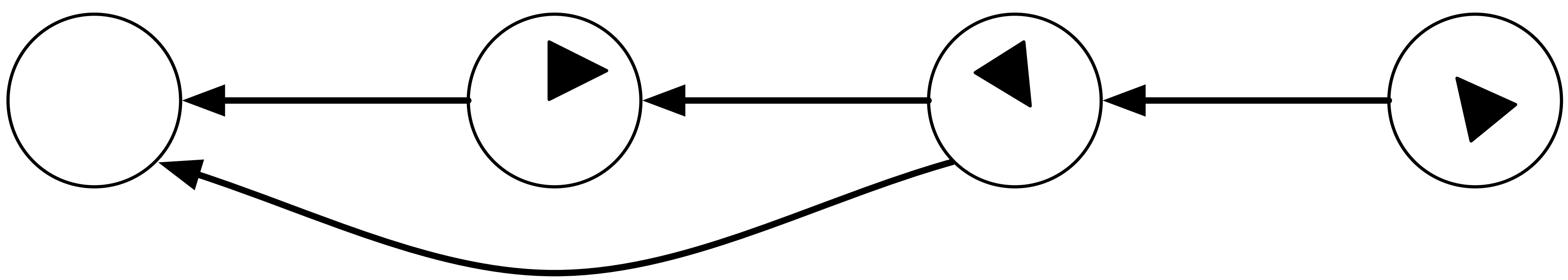


Here, Alice can win the game by moving the rightmost pebble to the middle. Now Bob's only option is to move one of these pebbles to the leftmost point; then Alice moves the other pebble left, and Bob has no moves so Alice wins.

In the following questions, assume both Alice and Bob play perfectly. That is, if the current player can move so that that they can win by continuing to play perfectly, then they make a winning move.

Question 3: a)

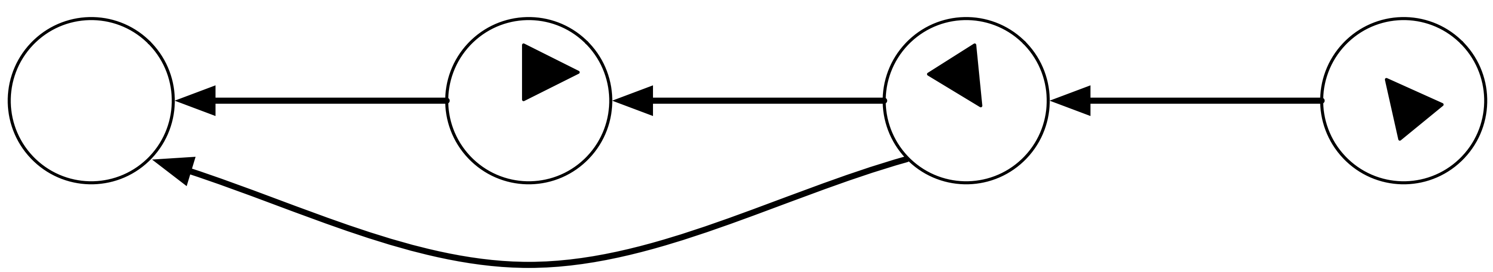
[2 marks] Who wins this game? Remember, Alice plays first. Explain briefly. It might help to label the pebbles.



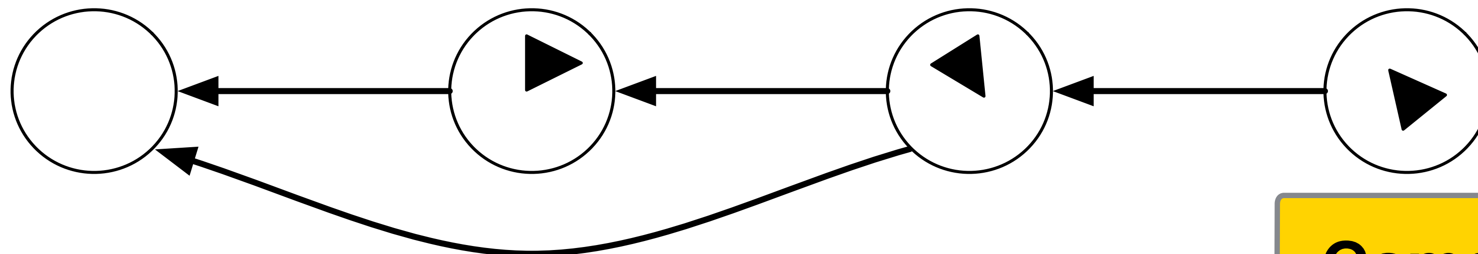
Question 3: a) (Solution)

Call the pebbles a, b, c in order from left to right.
Alice can force a win. She moves pebble b to the position with pebble a. She can now guarantee a win through the given strategy:

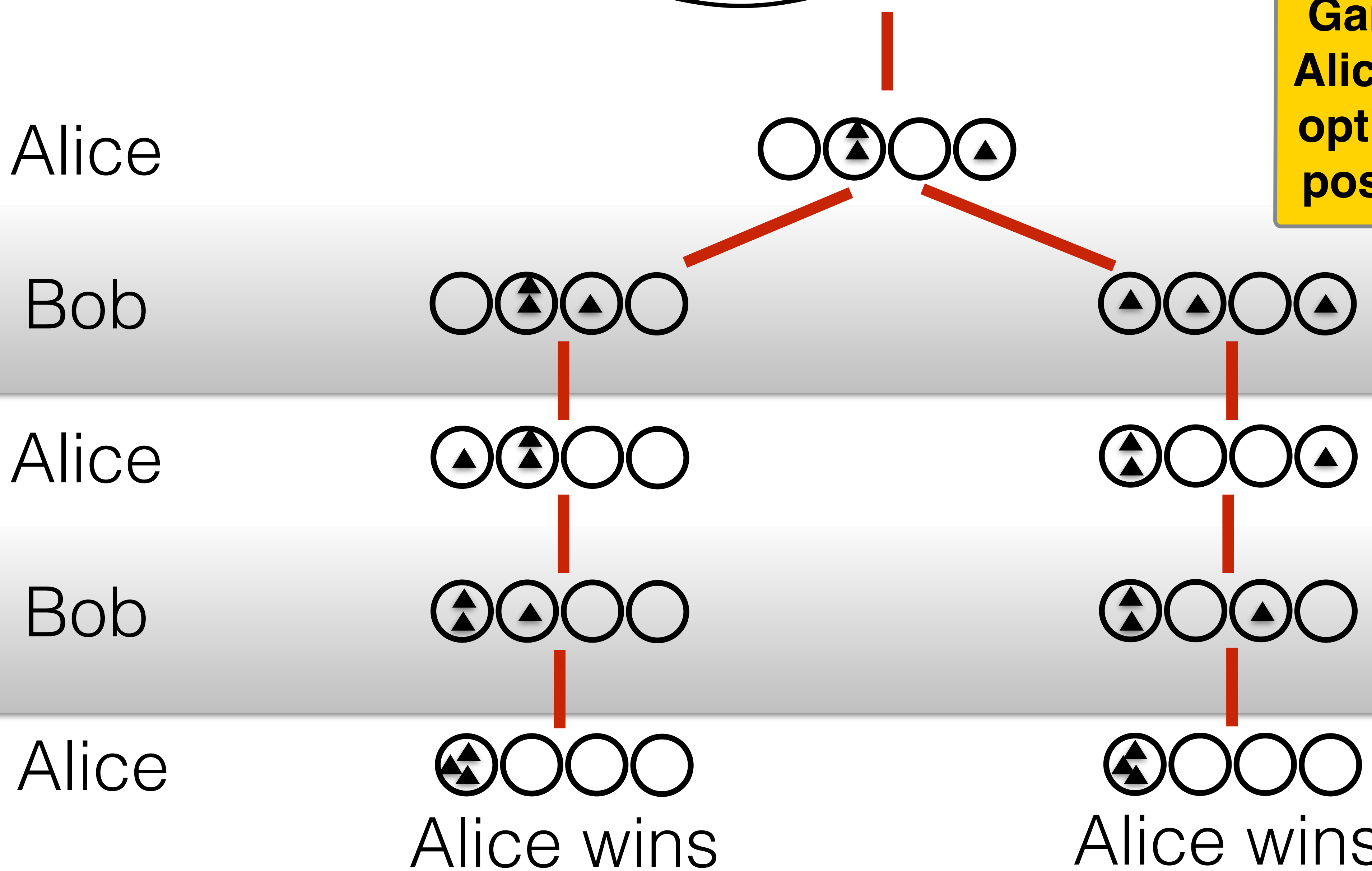
- When Bob moves pebble c, Alice responds by moving c to the leftmost position.
- When Bob moves one of a or b to the leftmost position, Alice responds by moving the other to this position.



Question 3: a) (Proof)

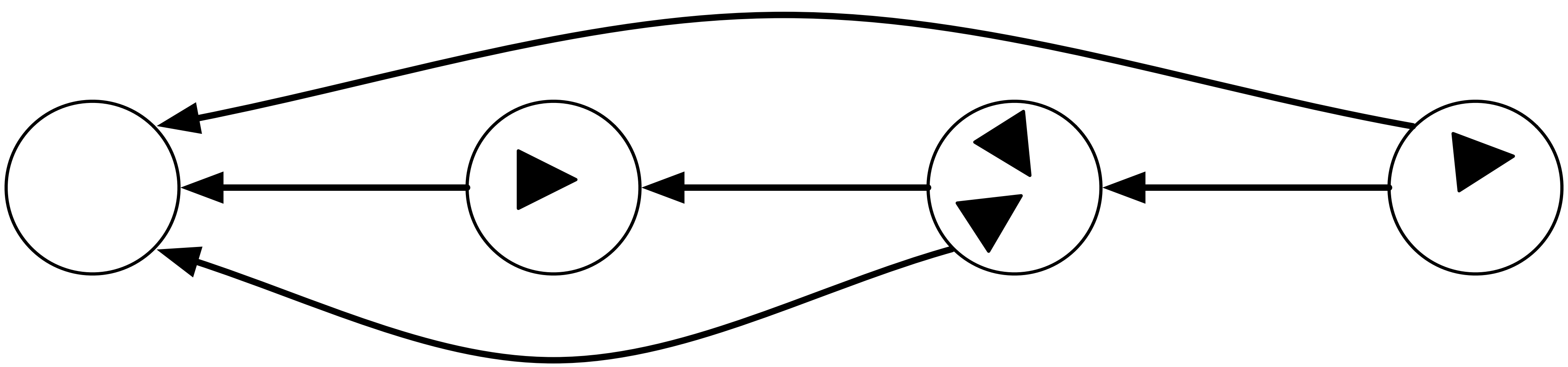


Game Tree: And/Or tree: Alice (or) can choose one option, but we play every possibility for Bob (and).



Question 3: b)

[2 marks] Who wins this game? Explain briefly.

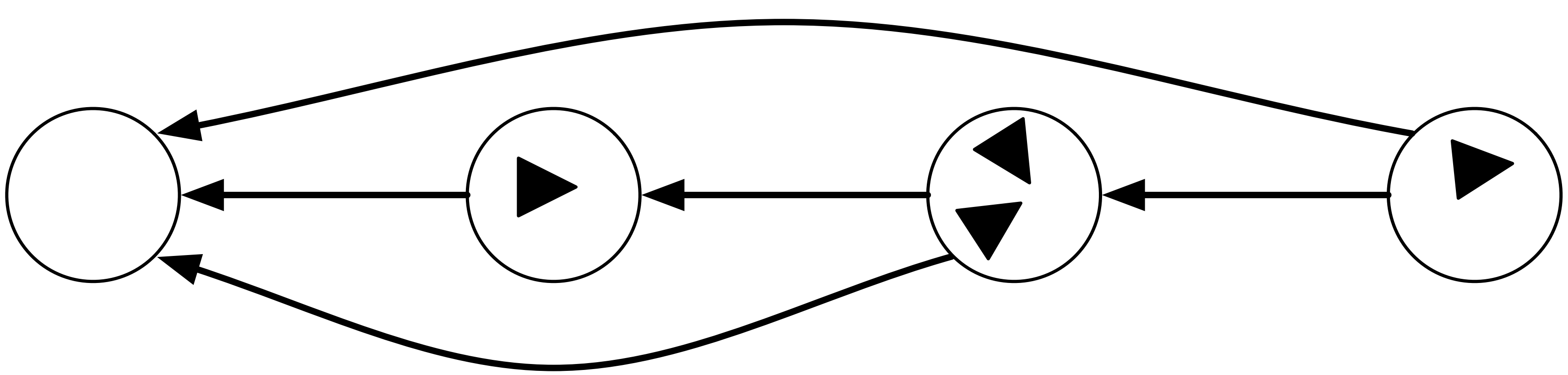


Question 3: b) (Solution)

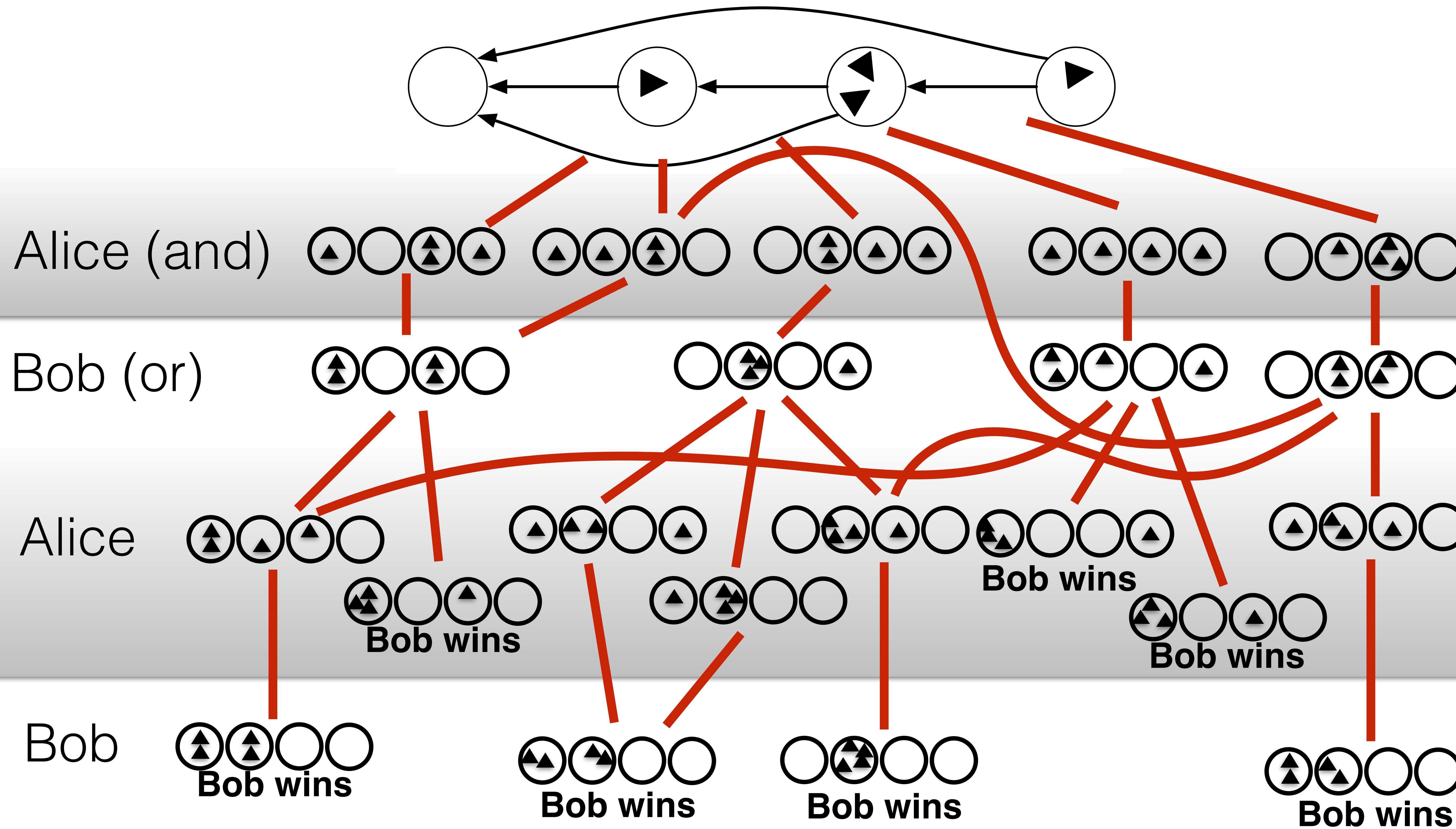
Label the pebbles a, b, c, d in left to right order. Bob will win through the following strategy.

• When Alice moves either a or d to the leftmost position, Bob responds by moving the other to the leftmost position.

• When Alice moves either b or c, Bob responds by moving the other to the same position.

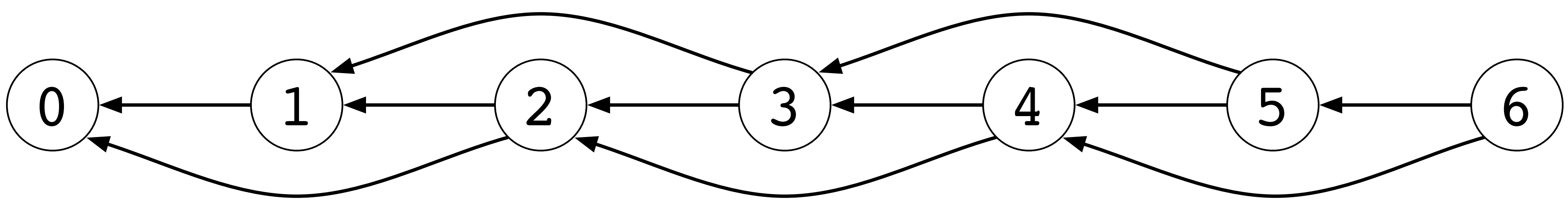


Question 3: b) (Proof)



Question 3: c)

[3 marks] Consider the game with `n` positions numbered from `0` to `n - 1` (left to right) and only these valid moves: for `0 ≤ v ≤ n-1`, a pebble can move from `v` to `v-1`; for `v ≥ 2`, a pebble can move from `v` to `v-2`. The game with `n=7` is shown below.



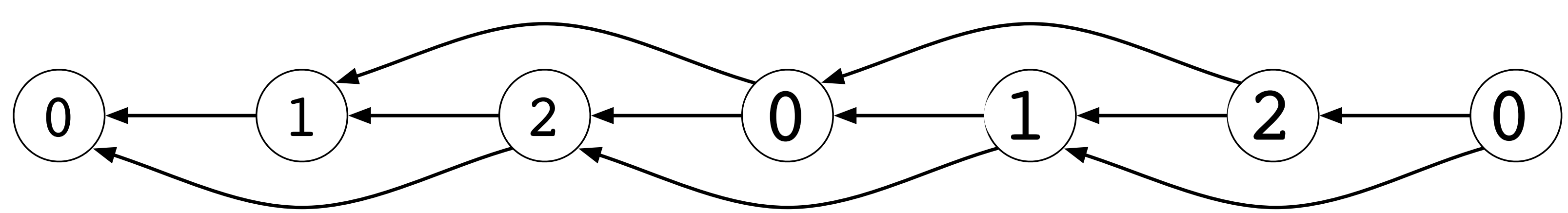
Write a function `winner(i, j)` that determines who wins on such a board with one pebble at location `i > 0` and one pebble at location `j > 0`, where possibly `i = j`. Return a string, either "Alice" or "Bob".

Explain briefly why your code is correct. For full marks, your algorithm should run in a fraction of a second, even for `i` and `j` as large as 109. Hint: there is a nice pattern.

Question 3: c) (Solution)

Alice wins if `i` and `j` are different `mod 3`, Bob wins if they are the same `mod 3`. A short explanation is that if they are different `mod 3` then Alice is able to make them the same by subtracting 1 or 2 from the larger `mod 3` value. If this did not end the game, then Bob's only move results in them being different `mod 3` and Alice repeats. If they are initially the same `mod 3`, then Bob just follow's Alice's strategy described above.

A more precise explanation follows. Change the label of each position `i` to `i mod 3`. In left-to-right order, the labels are 0, 1, 2, 0, 1, 2, 0, 1, 2, ... Eg:



Question 3: c) (Solution)

Let us call the placement of two pebbles *similar* if they lie on positions with the same label. Otherwise they are *dissimilar*. Note that the end placement (both pebbles on 0) is *similar*.

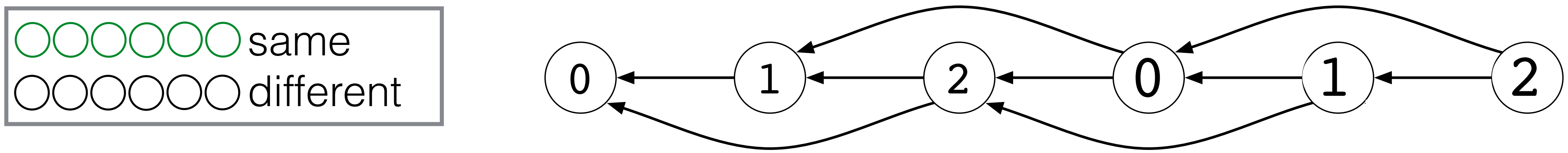
- There is a way to turn any *dissimilar* placement into a *similar* placement by moving one pebble. Move the pebble on the higher `mod 3` label to match the other.
- Any single move from a *similar* placement will result in a *dissimilar* placement because there is no move between positions with the same label.

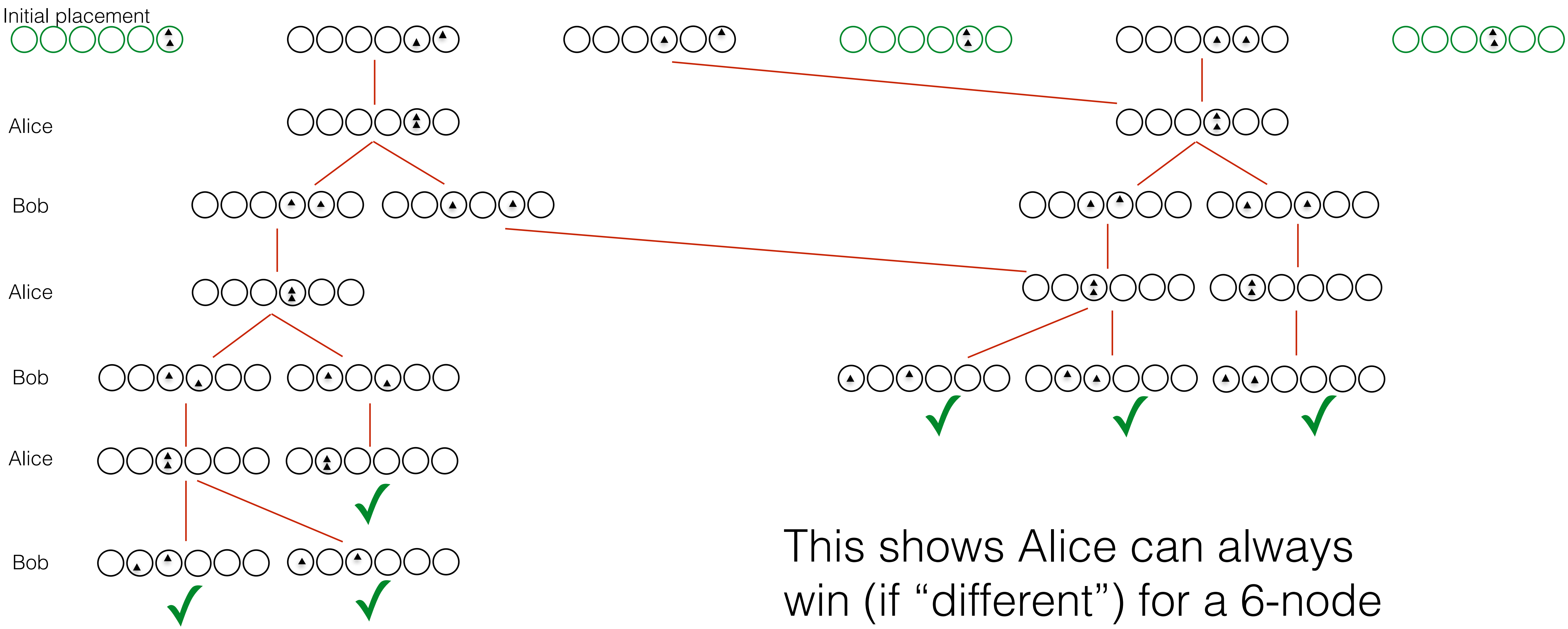
Whenever a player plays from a *dissimilar* placement, then make the move that results in a *similar* placement. If this did not finish the game, then the other player is forced to play from a *similar* position which creates a *dissimilar* position (and cannot be the last move in the game). This strategy is repeated. Thus, if the initial placement is *dissimilar* then Alice will win, otherwise Bob will win.

Question 3: c) (Solution)

```
def winner(i, j):
    if i % 3 == j % 3:
        return "Bob":
    else:
        return "Alice"
```

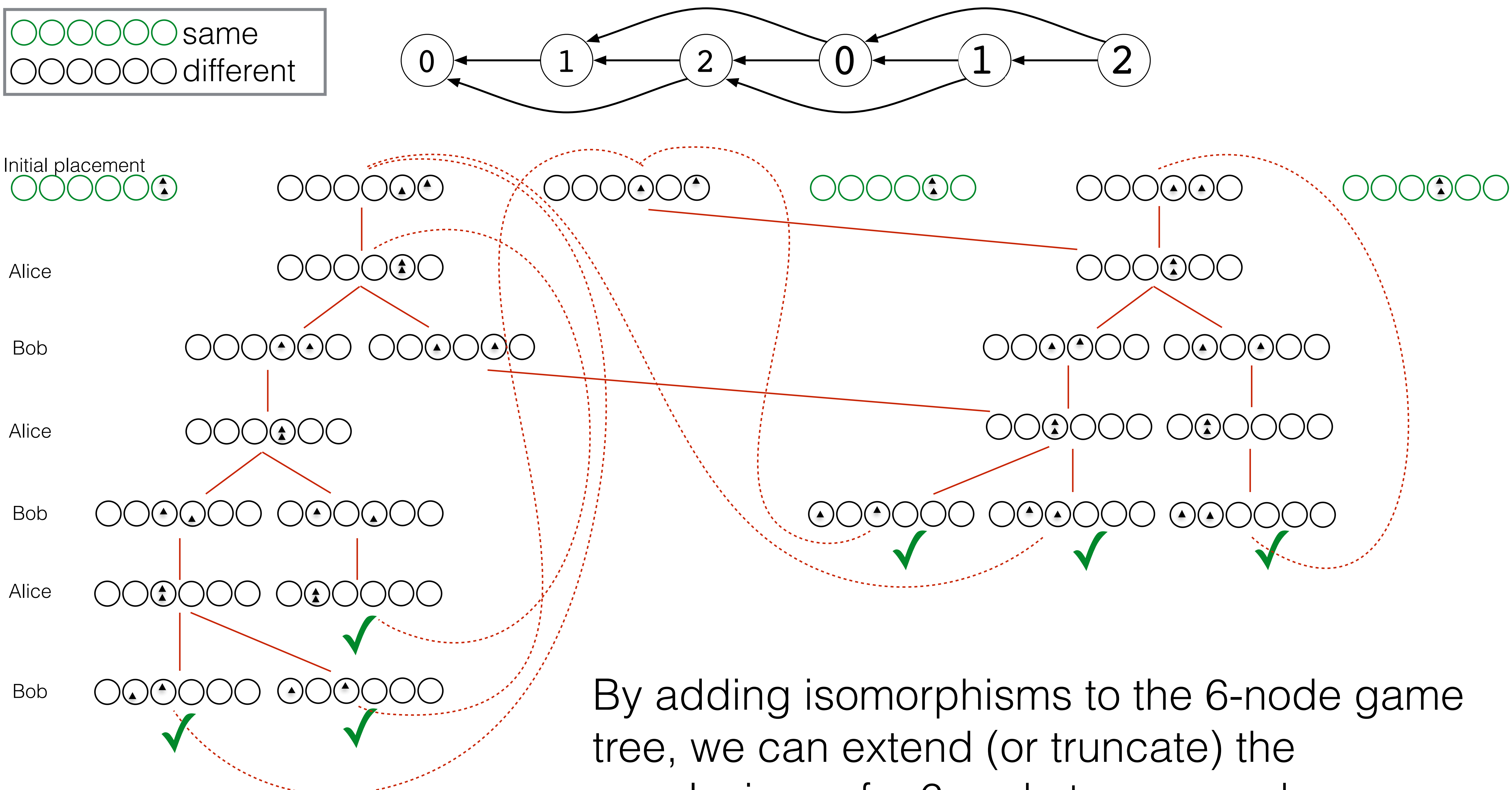
Question 3: c) (Proof for Alice's win)





This shows Alice can always win (if "different") for a 6-node game.

Question 3: c) (Proof for Alice's win)



By adding isomorphisms to the 6-node game tree, we can extend (or truncate) the conclusions of a 6-node to an n-node game.

Question 3: c) (Proof)

For the tree proof, we also need to show the Bob can always will in we start off with a *similar* state, but you get the picture…

Question 3: d)

[3 marks] Now consider the same board as the previous part, except we may have many pebbles on the board. Write pseudocode for a function `win(a, m, n)` where `n` is the number of positions on the board and `a[]` is an array with `m` nonnegative integers, each between `0` and `n-1`, specifying the initial placement of the pebbles. Again, this code should run quickly and you must explain briefly why it is correct.

Question 3: d) (Solution)

Call a placement of pebbles *similar* if both a_1 and a_2 are even, and *dissimilar* otherwise. Note the game ends in a *similar* placement.

- If the placement is *dissimilar*, then there is a single move that makes it *similar*. That is, if exactly one of a_1 or a_2 is odd then move a single pebble of the corresponding label to a position with label 0. If both a_1 and a_2 are odd then move a pebble from a position with label 2 to a position with label 1.
- If the placement is *similar*, then every move makes it *dissimilar*. If the move is from either a label 1 or a label 2 position, then the corresponding value a_1 or a_2 becomes odd. If the move is from a label 0 position, then it ends on a label 1 or label 2 position making the corresponding a_1 or a_2 odd.

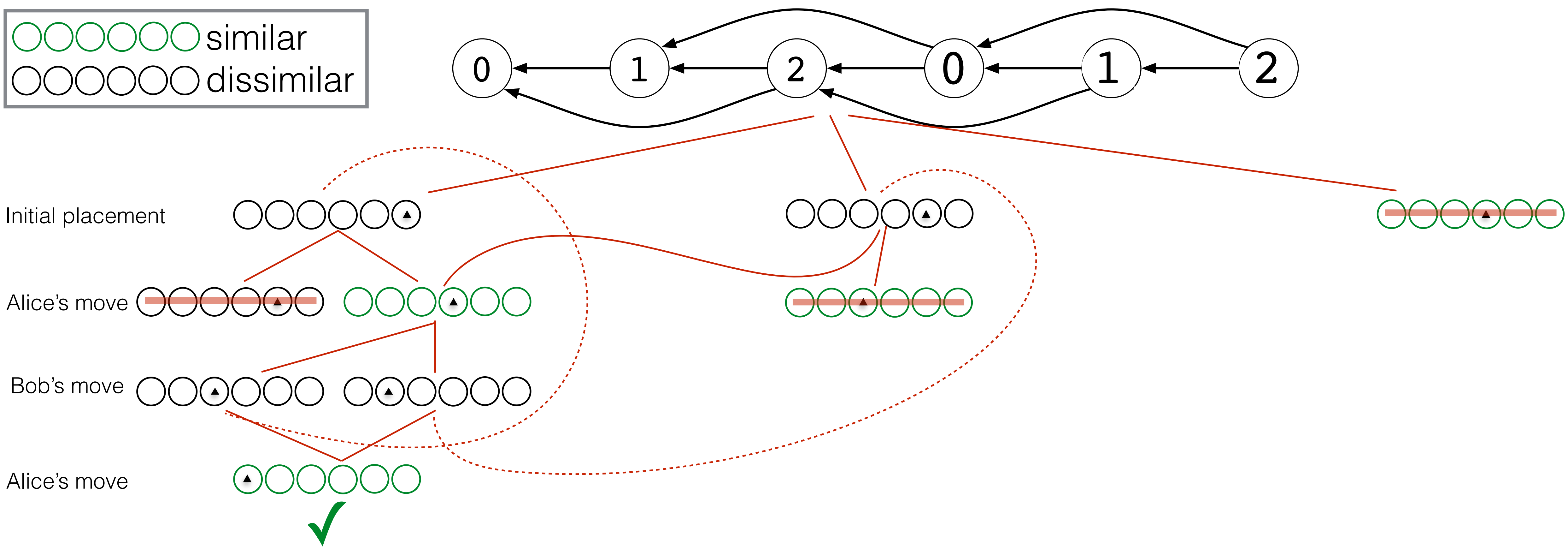
Alice wins if the initial placement is *dissimilar* and Bob wins if the initial placement is *similar* by following essentially the same strategy as in part c, except using this notion of *similar* and *dissimilar*.

Question 3: d) (Solution)

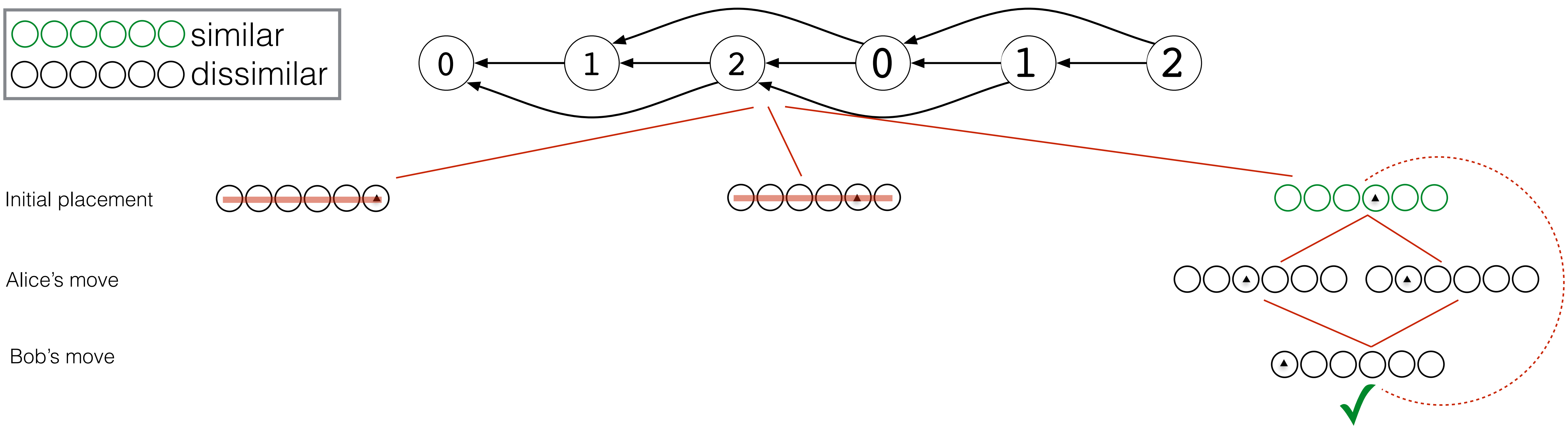
```
def win(a, m, n):
    a1 = 0
    a2 = 0
    for i in a:
        if i%3 == 1:
            a1 += 1
        elif i%3 == 2:
            a2 += 1

    if a1%2 == 0 and a2%2 == 0:
        return "Bob"
    else:
        return "Alice"
```

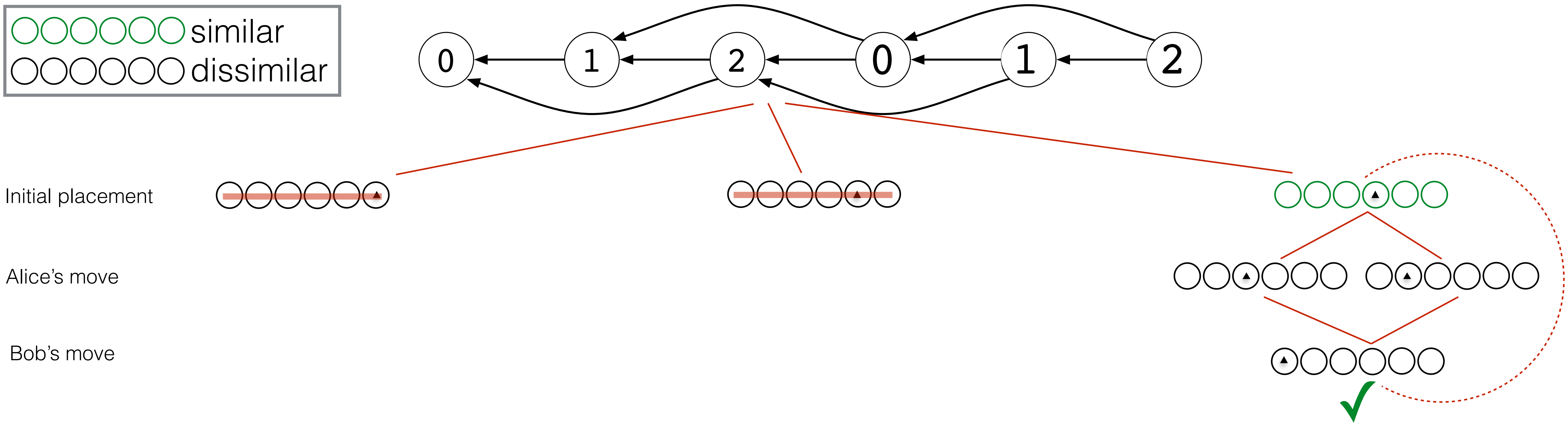
Question 3: d) (Solution Alice, dissimilar)



Question 3: d) (Solution Bob, similar)



Question 3: d) (Solution Bob, similar)



There's a bit of a "leap of faith" here!
Can we go from reasoning about one pebble to n pebbles?
Well, yes: we can think of each pebble in a game as a separate game. None of the games really matter except for the last one, which determines the evenness or oddness of the initial state…

Question 4: Fractions

You are given a fraction `a/b ≥ 0`. However, `a` and `b` might be large. So, given a positive integer `N`, you want to find a "simpler" fraction `c/d` that is as close to `a/b` as possible but with `0 ≤ c,d ≤ N`. That is, you should find `c` and `d` so that `0≤c≤N, 0<d≤N`, and `|a/b-c/d|` is as small as possible. If there are multiple solutions, choose the answer such that `c + d` is as small as possible.

[3 marks] Write a function `simpler(a,b,N)` that prints:

```
closest simpler fraction c / d
```

where `c` and `d` are the numerator and denominator of the answer. Remember, you can write helper functions.
Example: calling `simpler(5, 7, 5)` prints:

```
closest simpler fraction 3 / 4
```

and calling `simpler(11, 17, 7)` prints:

```
closest simpler fraction 2 / 3
```

Question 4 (Solution)

Simply iterate over all pairs `(c,d)` with `0 ≤ c ≤ N` and `1 ≤ d ≤ N`. If one of them forms a fraction that is closer than the previous best, then keep it.

The following code does exactly this. Note, that it never explicitly checks that `c + d` is the smallest among all possible answers. The order we iterate over `c, d` guarantees the first time we encounter a closest fraction that it will have minimum `c + d` value. Can you see why?

Question 4 (Solution)

```
def simpler(a, b, N):
    c=0
    d=1
    for num in range(1,N+1):
        for den in range(1,N+1):
            if closer(num, den, c, d, a, b):
                c, d = num, den
    print("closest simpler fraction", c, "/", d)
```

Requires a double "for" loop.

Question 4

```
# is the fraction num/den closer to a/b than c/d?
def closer(num, den, c, d, a, b):
    #num1/den1 = distance between num/den and a/b
    num1 = abs(num*b - a*den)
    den1 = den*b
    #num2/den2 = distance between a/b and c/d
    num2 = abs(c*b - a*d)
    den2 = d*b
    #cross multiply to check num1/den1 < num2/den2
    return num1*den2 < num2*den1
```

"closer" could be defined in several ways, but if done by division, careful attention has to be paid to explicit conversion.