Induction is a method for proving statements of the form "For all n, $P(n)$", where n ranges over the positive integers. It is particularly useful in computer science when reasoning about the correctness of algorithms. Let's start with an example.

Theorem 1. *For every positive integer n, the sum of the integers from 1 to n is $n(n+1)/2$.*

The kinds of proofs we learned last time don't seem to help. Before we do the proof let's gain some confidence that the theorem is plausible by working out a few small examples:

- When $n = 1$, $(1 + 1)/2 = 1$.
- When $n = 2$, $1 + 2 = 3$, and $2 \cdot (2 + 1)/2 = 3$.
- When $n = 3$, $1 + 2 + 3 = 6$ and $3 \cdot (3 + 1)/2 = 6$.
- When $n = 4$, $1 + 2 + 3 + 4 = 10$ and $4 \cdot (4 + 1)/2 = 10$.

The cases check out, but we cannot go on like this forever. How can we prove the theorem for *all* n? Let's give the predicate "The sum of integers from 1 to n is $n(n + 1)/2$ a name; call it $P(n)$.

Induction models the following reasoning process. First, we prove $P(1)$. Then we prove $P(2)$; in our proof for $P(2)$, we can assume that $P(1)$ is known to be true (use it as an axiom). When we prove $P(3)$, we can assume $P(2)$ to be true, and so on:

$$\frac{P(1) \qquad P(1) \longrightarrow P(2) \qquad P(2) \longrightarrow P(3) \qquad P(3) \longrightarrow P(4)}{P(1) \text{ AND } P(2) \text{ AND } P(3) \text{ AND } P(4)}$$

We can extend this reasoning to a *general* value of n. If we prove $P(1)$ is true, and we prove that for every $n \geq 1$, $P(n + 1)$ is true *assuming* $P(n)$, then $P(n)$ must be true for all n:

Induction proof method:

$$\frac{P(1) \qquad P(n) \longrightarrow P(n+1) \text{ for all positive integers } n}{P(n) \text{ for all positive integers } n}$$

Proposition $P(1)$ is called the *base case*; proposition "$P(n) \longrightarrow P(n + 1)$ for all n" is called the *inductive step*. To prove the inductive step, you can try any of the methods from last lecture.

Proof of Theorem 1. We prove the theorem by induction on n. Let $S(n)$ denote the sum of the first n integers. Then the proposition says that

$$S(n) = \frac{n(n+1)}{2} \qquad \text{for all positive integers } n$$

Base case $n = 1$: $S(1) = 1$ and $1(1 + 2)/2 = 1$, so $S(1) = 1(1 + 1)/2$.

Inductive step: We need to show that for every positive integer n

$$S(n) = \frac{n(n+1)}{2} \quad \longrightarrow \quad S(n+1) = \frac{(n+1)(n+2)}{2}.$$

Let n be any positive integer. We assume $S(n) = n(n+1)/2$. Then

$$S(n+1) = S(n) + (n+1)$$

so, by our assumption that $S(n) = n(n+1)/2$, we get

$$S(n+1) = S(n) + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2}.$$

It follows by induction that $S(n) = n(n+1)/2$ for all positive integers n. $\square$

1 More proofs by induction

Let's do another example, this one involving an inequality. The *factorial* of n, denoted by $n!$, is the number obtained by multiplying all integers from 1 to n:

$$n! = 1 \cdot 2 \cdots n.$$

Theorem 2. *For every integer $n \geq 4, n! > 2^n$.*

We will prove this theorem by induction. The base case here will be $n = 4$.

Proof. We prove the theorem by induction on n.

Base case $n = 4$: $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24 > 16 = 2^4$, so the base case holds.

inductive step: We need to show that for every positive integer $n \geq 4$

$$n! > 2^n \quad \longrightarrow \quad (n+1)! > 2^{n+1}.$$

Let n be any positive integer greater or equal to 4. We assume $n! > 2^n$. Then

$$(n+1)! = n! \cdot (n+1) > 2^n \cdot (n+1) > 2^n \cdot 2 = 2^{n+1}.$$

It follows by induction that $n! > 2^n$ for all integers $n \geq 4$. $\square$

How did one come up with the inequality $2^n \cdot (n+1) > 2^n \times 2$? It was a bit of a lucky guess, but you can reason about it backwards. In the inductive step, we *need* to prove that $2^n \cdot (n+1) > 2^{n+1}$. If we factor out 2^n from both sides, we are left with showing that $n + 1 > 2$. This is the same as saying $n > 1$, which is certainly true under the assumption $n \geq 4$.

This kind of "backwards reasoning" is often helpful in proofs by induction. You are encouraged to use it as part of your scratch work, but not in the written proof.

Theorem 3. *For every positive integer n, $n^3 - n$ is a multiple of 6.*

You can prove this theorem in several ways. Try a proof by cases at home. Here we'll do it using induction.

Proof. We prove the theorem by induction on n.

Base case $n = 1$: $1^3 - 1 = 0$, which is a multiple of 6, so the base case holds.

inductive step: We need to show that for every positive integer n,

$$n^3 - n \text{ is a multiple of } 6 \quad \longrightarrow \quad (n+1)^3 - (n+1) \text{ is a multiple of } 6.$$

Let n be any positive integer. We assume that $n^3 - n$ is a multiple of 6. Then

$$(n+1)^3 - (n+1) = (n^3 + 3n^2 + 3n + 1) - (n+1) = n^3 + 3n^2 + 2n = (n^3 - n) + 3(n^2 + n)$$

By inductive hypothesis, $n^3 - n$ is a multiple of 6. In the last lecture we showed that $n^2 + n$ is even for all n, so $3(n^2 + n)$ is also a multiple of 6. Therefore $(n^3 - n) + 3(n^2 + n)$ is also a multiple of 6.

It follows by induction that $n^3 + n$ is a multiple of 6 for all positive integers n. □

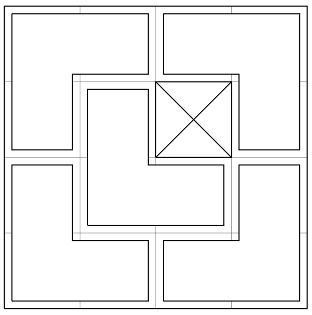
Where did the equality

$$n^3 + 3n^2 + 2n = (n^3 - n) + 3(n^2 + n)$$

come from? Our assumption says that $n^3 - n$ is divisible by 6, but the expression $n^3 + 3n^2 + 2n$ doesn't "contain" $n^3 - n$. To take advantage of the assumption, it makes sense to subtract one n from n^3 and compensate by adding another one to $3n^2 + 2n$.

Strengthening the hypothesis

You are given a 2^n by 2^n square grid with a central square removed. (A central square is one that touches the center of the grid.) You want to tile the remaining squares with L-shaped tiles (each tile occupies 3 squares). Can it always be done? Here is an example with $n = 2$:



Let us prove that that a tiling always exists.

Theorem 4. *For every positive integer n, there exists a tiling of a 2^n by 2^n square grid with a central square removed using L-shaped tiles.*

Let us try to prove this theorem by induction. In the base case $n = 1$, the tile has dimensions 2 by 2 and the proposition is clearly true.

Now let's try the inductive step. Let us fix n and assume the proposition is true for n, namely there exists a tiling of a 2^n by 2^n grid with a central square removed. We want to show that there also exists a tiling of a 2^{n+1} by 2^{n+1} grid with a central square removed. To apply the inductive hypothesis, it makes sense to split this grid into four 2^n by 2^n subgrids. Unfortunately, the subgrids don't satisfy the requirement of having their central square removed; one of them will be missing a corner and the other three will be whole. It looks like we are stuck.

The trick here is to prove a *more general* theorem – one of which Theorem 4 is a special case.

Theorem 5. *For every positive integer n, there exists a tiling of a 2^n by 2^n square grid with* any one *square removed using L-shaped tiles.*

Proof. We prove the theorem by induction on n.

Base case $n = 1$: Given a 2 by 2 grid with any square removed, the other 3 form a 2 by 1 L-shape, so they can be covered by one tile. Therefore a covering of the grid by tiles exists.

Inductive step: Let us assume that a 2^n by 2^n grid with any one square removed can be tiled using L-shaped tiles. We will show that the same is true for a 2^{n+1} by 2^{n+1} grid. Let G be a 2^{n+1} by 2^{n+1} grid with some square removed. Divide G into four 2^n by 2^n quadrants G_1, G_2, G_3, G_4. One of these quadrants will contain the missing tile. Temporarily remove the center tiles of G that do not below to that quadrant. Then each of G_1, G_2, G_3, and G_4 becomes a 2^n by 2^n grid with one square removed. By inductive hypothesis, each one of them can be tiled using 2 by 1 L-shapes. Tile all the subgrids and cover the temporarily removed three center tiles of G by one more L-shape. The resulting tiling covers all of G except for the removed square.

It follows by induction that for every n there exists a tiling of the 2^n by 2^n square grid with any square removed using L-shaped tiles. □

Theorem 5 (which allows for any square in the grid to be removed) is more general than Theorem 4, so we would expect it to be more difficult to prove. However, we proved Theorem 5, while the same proof method failed when we tried it on Theorem 4! The reason is that in proofs by induction, the predicate $P(n)$ plays the role both of assumption and conclusion. Sometimes making a stronger assumption allows us to prove a stronger conclusion.

An interesting feature of this proof is that it not only tells us the desired tiling exists, but also how to find it. You can write a computer program that does it or play with the python code available from the course web page.

A false proof

"Theorem:" In every nonempty set of horses, all horses are of the same colour.

Proof. We prove this theorem by induction on the size n of the set.

Base case $n = 1$: The set has one horse, so the statement is true.

Inductive step: Assume that in every collection of n horses, all of them have the same colour. We will prove that in every set of $n + 1$ horses, all of them have the same colour. Take any set of $n + 1$ horses:

$$h_1, h_2, \ldots, h_{n+1}.$$

By our assumption, the first n horses $h_1, \ldots, h_n$ are of the same colour. By the same assumption, the last n horses $h_2, \ldots, h_{n+1}$ have the same colour. So $h_1, \ldots, h_{n+1}$ all have the same colour.

It follows by induction that all horses in the set are of the same colour. □

Where was the mistake? To "debug" a proof by induction, it is a good idea to try out some values of n and see where the chain of reasoning went wrong. Let $P(n)$ be the predicate "All sets of n horses have the same colour." As we saw, $P(1)$ is true. However, $P(2)$ is already false. So the inductive step fails when $n = 1$, that is when we try to prove $P(2)$ assuming $P(1)$. The proof says that in this case, the first 1 horse(s) have the same colour and the last 1 horse(s) have the same colour. We cannot conclude that both have the same colour! In other words, the deduction

$$\frac{h_1, \ldots, h_n \text{ have the same colour} \qquad h_2, \ldots, h_{n+1} \text{ have the same colour}}{h_1, \ldots, h_{n+1} \text{ have the same colour}}$$

is not valid when $n = 1$.

2 State Machines and Invariants

In computer science you often encounter systems that evolve over discrete time according to some rules. State machines provide a useful abstraction for describing such systems. A *state machine* is specified by a set of *states*, a *transition predicate* $q \to r$ that says if the system is allowed to transition from state q to state r, and a *start state*.

An *invariant* is a predicate of states that remains true over the lifetime of the state machine. Induction allows us to prove invariants: To show the invariant holds, we prove it is satisfied in the initial state, and that assuming it holds at time n, it also holds at time $n+1$ for every n – that is, it is preserved by the transitions.

Here is an example. You have a robot that can walk across diagonals on an infinite 2-dimensional grid. Its coordinates at any given time are described by a pair of integer coordinates (x, y). In each time step, the robot moves by exactly one unit left or right *and* by exactly one unit up or down. At time 0 robot starts at position $(0, 0)$. (Thus, at time 1, the robot will be in one of the four positions $(-1, -1)$, $(-1, 1)$, $(1, -1)$, $(1, 1)$.) Can the robot ever reach position $(1, 0)$?

In this example, the motion of the robot can be described by a state machine whose states are pairs of integers (x, y), whose transitions are described by the predicate

$$\begin{aligned}(x, y) \to (x', y') \quad \text{if} \quad & (x' = x - 1 \text{ AND } y' = y - 1) \text{ OR } (x' = x - 1 \text{ AND } y' = y + 1) \text{ OR} \\ & (x' = x + 1 \text{ AND } y' = y - 1) \text{ OR } (x' = x + 1 \text{ AND } y' = y + 1),\end{aligned}$$

and whose start state is the state $(0, 0)$. The next theorem states that the predicate "$x + y$ is even" is an invariant of this system:

Theorem 6. *For every n, if the robot is at position (x, y) at time n, then $x + y$ is even.*

Therefore the robot can never reach position $(1, 0)$ because $1 + 0$ is odd.

Proof. We prove the theorem by induction on n.

Base case $n = 0$: The start state is $(0, 0)$ and $0 + 0$ is even.

Inductive step: Assume that at time n, the robot is at position (x, y) and $x + y$ is even. Let (x', y') be the position of the robot at time $n+1$. We will prove that $x' + y'$ is even by case analysis:

- The robot moves left and down: Then $x' = x - 1$, $y' = y - 1$, so $x' + y' = (x + y) - 2$ – an even number minus two, therefore even.
- The robot moves left and up: Then $x' = x - 1$, $y' = y + 1$, so $x' + y' = x + y$, which is even.
- The robot moves right and up: Then $x' = x + 1$, $y' = y + 1$, so $x' + y' = (x + y) + 2$ – an even number plus two, therefore even.

- The robot moves right and down: Then $x' = x + 1$, $y' = y - 1$, so $x' + y' = x + y$, which is even.

It follows by induction that the sum of the robot's coordinates is always even. $\square$

Here is another, more challenging example. It is about the following puzzle. The starting configuration is the one on the left. You are supposed to reach the configuration on the right. At each point, you are allowed to move an adjacent tile into the unoccupied square.

1	2	3
4	5	6
8	7	

1	2	3
4	5	6
7	8	

We will show that this is not possible using an invariant. This invariant is trickier than the one in the last problem. To explain it, we need two concepts.

We say tile a appears *before* tile b if tile a is in a higher row than tile b or if they are in the same row, tile a is to the left of tile b. We say the pair of tiles (a, b) form an *inversion* if $a < b$ and tile b appears before tile a.

For example, in the final configuration there are no inversions. In the initial configuration, there is exactly one inversion consisting of the pair $(7, 8)$. In the following configuration, the inversions are $(2, 3)$, $(2, 4)$, $(2, 5)$, $(2, 6)$, $(2, 7)$, $(3, 6)$, $(4, 6)$, $(4, 7)$, $(5, 6)$, and $(5, 7)$.

1	6	3
7	4	
5	2	8

We will show the following invariant: The number of inversions is odd.

Theorem 7. *For every n, after n moves the number of inversions is odd.*

Proof. We prove the theorem by induction on n.

Base case $n = 0$: In the initial configuration, there is exactly one inversion – the pair $(7, 8)$. So initially the number of inversions is odd.

Inductive step: Assume that after n steps, the number of inversions is odd. We will show that the same is true after $n + 1$ steps. The $(n + 1)$st move can be a row move (a tile moves to the left or to the right) or a column move (a tile moves up or down). The proof is by case analysis.

- If the $(n+1)$st move is a row move, then the relative order of any pair of tiles stays the same, so the number of inversions stays the same. By assumption, it is odd.

- If the $(n+1)$st move is a column move, then the only pairs of tiles whose relative order changes are the pairs of type (m, b), where m is the tile that was moved and b is a tile between the tile that was moved and the empty space. There are exactly two tiles of the second type. We conclude that the $(n+1)$st move changed the relative order of exactly two pairs of tiles. Let's call these pairs p_1 and p_2.

  Now we consider three subcases. If p_1 and p_2 were both inversions at time n, then the number of inversions in move $n+1$ decreases by 2, so it remains odd. If one of them was an inversion at time n but not the other, then the number stays the same, so it remains odd. If neither was an inversion at time n, then the number of inversions increases by 2, and it also remains odd. We conclude that the number of inversions after move $n+1$ is odd.

It follows by induction that the number of inversions is always odd. □

In the final configuration, the number of inversions is even. By Theorem 7, the final configuration can never be reached.

3 Strong induction

Recall our intuition for induction: If we want to prove a predicate $P(n)$ holds for all $n \geq 1$, first we prove $P(1)$. Then assuming $P(1)$ we prove $P(2)$. Then assuming $P(2)$ we prove $P(3)$; and so on. In fact, we can make our assumptions stronger as we go along. By the time we are proving $P(3)$, we have proved not only $P(2)$ but also $P(1)$, so we can assume both of them to hold:

$$\frac{P(1) \quad P(1) \longrightarrow P(2) \quad (P(1) \text{ AND } P(2)) \longrightarrow P(3) \quad (P(1) \text{ AND } P(2) \text{ AND } P(3)) \longrightarrow P(4)}{P(1) \text{ AND } P(2) \text{ AND } P(3) \text{ AND } P(4)}$$

The strong induction proof method extends this reasoning to a general value of n: If we prove $P(1)$ and we prove that for every $n \geq 1$, $P(n+1)$ is true assuming $P(1)$ up to $P(n)$ are all true, then $P(n)$ must be true for all n:

Strong induction proof method:

$$\frac{P(1) \qquad (P(1) \text{ AND } \ldots \text{ AND } P(n)) \longrightarrow P(n+1) \text{ for all positive integers } n}{P(n) \text{ for all positive integers } n}$$

Here is a problem where strong induction is useful. Suppose you have an unlimited supply of \$3 and \$5 stamps. Which postage amounts are your stamps good for?

Let's model this as a problem about numbers. The postage amounts you can obtain are numbers of the form $3a + 5b$, where a and b range over all nonnegative integers. You can easily see that the numbers $1, 2, 4, 7$ cannot be written in this way, while $3, 5, 6$ can. Let's now see what happens when $n \geq 8$.

We can write $8 = 3 + 5$, $9 = 3 \times 3$, $10 = 2 \times 5$. Once we have 8, 9, and 10, we can form any other number by adding a sufficient number of 3s: $11 = 8 + 3$, $12 = 9 + 3$, $13 = 10 + 3$, $14 = 11 + 3$, and so on. Here is how we write this argument as a proof by strong induction.

Theorem 8. *Any integer $n \geq 8$ can be written in the form $3a + 5b$ for some integers $a, b \geq 0$.*

Proof. We prove the theorem by strong induction on n.

Base case $n = 8$: We can write $8 = 3 + 5$, so the theorem is true for $a = b = 1$.

Inductive step: Let n be any number greater than or equal to 8. We assume any integer from 8 to n can be written in the form $3a + 5b$. We will show that $n + 1$ can also be written as $3a + 5b$. The proof is by cases:

- **Case $n + 1 = 9$:** We can write $9 = 3 \times 3$.
- **Case $n + 1 = 10$:** We can write $2 = 5 \times 2$.
- **Case $n+1 \geq 11$:** In this case, $(n+1) - 3 \geq 8$, so by inductive hypothesis, $(n+1) - 3 = 3a + 5b$ for some integers $a, b \geq 0$. Then $n + 1 = 3(a + 1) + 5b$.

By strong induction, we can write any $n \geq 8$ in the form $3a + 5b$ for integers $a, b \geq 0$. □

4 Concurrency*

Concurrency is the study of computer programs that run in parallel but have access to a shared resource like memory. In such settings it is usually assumed that each program runs its instructions in sequence, but different programs can interleave their instructions in arbitrary ways. The objective is to design programs that accomplish some cooperative task regardless of the order in which their instructions are interleaved.

To get a sense of the issues that arise in concurrent programming let's do an example. Say Alice and Bob run an e-shipping company. Alice's job is to go around the internet and collect e-boxes. When she has collected at least ten e-boxes, Bob shows up to the scene and takes them away for e-delivery. Alice's program might look something like this:

Alice's code:

```
scavenge the internet for e-boxes
if new e-box found:
    e_boxes = e_boxes + 1
```

Bob's code:

```
if e_boxes ≥ 10
    e_boxes = e_boxes − 10
    e-deliver the ten e-boxes
```

To figure out how a concurrent execution of Alice's and Bob's programs look like we must make some assumptions about which operations constitute a "single step" in these programs. Such operations are called *events*. We will assume that reading a variable (from shared memory into a private register) is an event and writing a variable is also an event, but a consecutive read-write operation like `e_boxes` = `e_boxes` + 1 is not an event.[1]

Then the following undesirable behaviour can happen. Bob spots that the pile has grown to 10 e-boxes, so he decides to take them for e-delivery. While he is in the process of updating an e-box count, a new e-box is brought in by Alice:

[1] The validity of this assumption depends on the workings of the compiler and the computer architecture that these programs are running on. Many compilers do not even guarantee sequential execution of a stand-alone program like Bob's as they may re-order instructions in order to optimize the code.

program	event	`e_boxes`
	initially	10
Bob	read `e_boxes` (line 2)	10
Alice	read `e_boxes` (line 3)	10
Alice	write `e_boxes` (line 3)	11
Bob	write `e_boxes` (line 2)	0

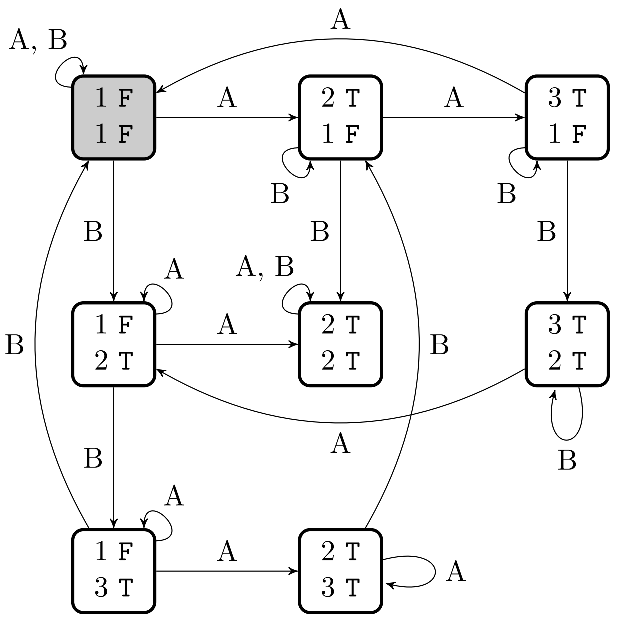
The e-box that Alice brought in the middle of Bob's read/write operation has vanished into thin air! In order for this concurrent program to work as desired, we would like Alice's third line of code and Bob's second line of code to execute as if they were single events. One common mechanism for this is locking: Certain parts of the code are designated as *critical sections* and it is ensured that critical sections satisfy *mutual exclusion*: Alice and Bob cannot be both executing their critical sections concurrently. Is it possible to implement mutual exclusion using only read events and write events?

Mutual exclusion and state machines Let's try the following. Alice has a boolean variable `Awants` that indicates her desire to do critical work. Before entering its critical section, Alice checks that Bob is not interested in critical work himself and proceeds only if this is the case. Bob does the same with respect to Alice:

Alice's code:
```
1   Awants = True
2   while Bwants :
        wait
3   do critical work
    Awants = False
```

Bob's code:
```
1   Bwants = True
2   while Awants :
        wait
3   do critical work
    Bwants = False
```

Initially, `Awants` and `Bwants` are both false. One method for understanding how these two programs interact is to model their concurrent execution as a state machine. The states are specified by the memory contents together with the values of Alice's and Bob's program counters (PCs). In this example, the state machine is small enough so that we can represent the portion induced by the reachable states graphically:



The two rows refer to Alice's and Bob's program state, respectively. Transitions are labeled by the program(s) that trigger them. For example, in the top right state, Alice's program is just about to execute line 3 (and `Awants` is true), while Bob hasn't started running his program (and `Bwants` is false). There are three labeled transitions after this state that correspond to Alice executing her line 3, Bob executing his line 1, and Bob idling just before line 1. In this model each party can execute its program arbitrarily often (states representing PC value 3 transition back to those with PC value 1) and also idle for an arbitrary number of steps before executing, if ever (states with PC value 1 have self-loops).

The condition "both programs are not in their critical section at the same time" means that Alice's and Bob's PCs cannot have value 3 at the same time. In state machine notation, none of the states

$$\boxed{\begin{matrix}3\ \mathtt{T}\\3\ \mathtt{T}\end{matrix}}\qquad\boxed{\begin{matrix}3\ \mathtt{T}\\3\ \mathtt{F}\end{matrix}}\qquad\boxed{\begin{matrix}3\ \mathtt{F}\\3\ \mathtt{T}\end{matrix}}\qquad\boxed{\begin{matrix}3\ \mathtt{F}\\3\ \mathtt{F}\end{matrix}}$$

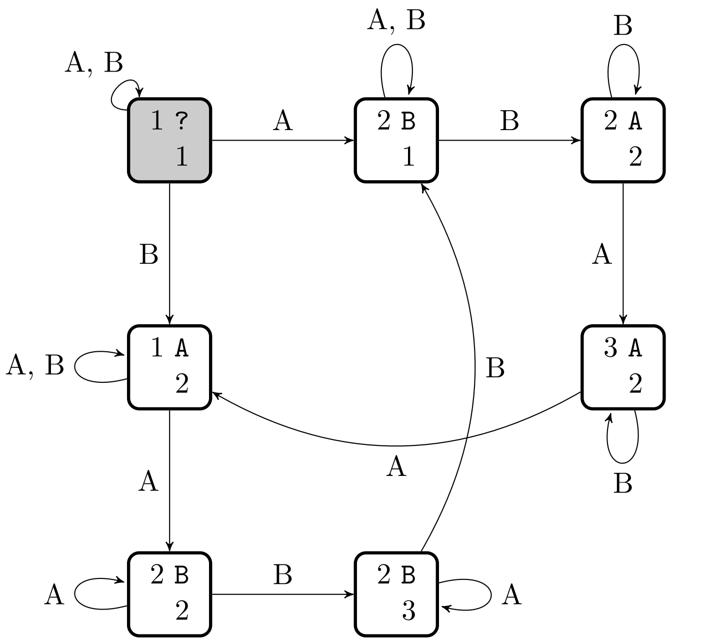
should be reachable from the start state. This is indeed the case. There is, however, another problem: The concurrent execution can *deadlock* in the middle state `2T`, `2T`. If the programs reach this state their execution cannot proceed. This happens when Alice and Bob indicate their desire to go (they both execute their respective line 1) one right after the other.

In the hope of avoiding deadlock let us attempt a different solution: Alice and Bob maintain a shared variable called `turn`. When Alice wants to do critical work, she assigns `turn` to `Bob`. When Alice gets back her turn, she enters her critical section.

Alice's code:
```
1   turn = Bob
2   while turn == Bob :
        wait
3   do critical work
```

Bob's code:
```
1   turn = Alice
2   while turn == Alice :
        wait
3   do critical work
```

The concurrent execution of the two programs is described by the following state machine with transition labels. Here, the diagonal entries of the state indicate Alice's and Bob's PC and the remaining one is the value of the `turn` variable.



This implementation also satisfies the mutual exclusion property of state machines: No state in which both PCs have value 3 is reachable from the start state. However it also deadlocks: If Bob is not interested in doing critical work, Alice cannot ever make it to her critical section. This behavior occurs in the top middle state of the diagram. If Bob keeps revisiting this state indefinitely, Alice can never make progress.

Peterson's algorithm Our two attempted solutions to mutual exclusion are both prone to deadlock. The conditions under which deadlock occurs, however, are complementary in the two algorithms. In the first case, deadlock may happen when both Alice and Bob want to do critical work at the same time. In the second case, deadlock occurs when one of the parties is idle (not interested in doing critical work). The following algorithm combines the ideas behind the two into a single deadlock-free solution:

Peterson's algorithm

```
Alice's code:
1    Awants = True
2    turn = Bob
3    while Bwants AND  4 turn == Bob :
         wait
5    do critical work
     Awants = False
```

```
Bob's code:
1    Bwants = True
2    turn = Alice
3    while Awants AND  4 turn == Alice :
         wait
5    do critical work
     Bwants = False
```

As before, the concurrent execution of the programs can be represented by a state machine with labelled transitions. This state machine is, however, slightly too large to draw by hand with confidence. We could use the computer to convert our code into a (finite) state machine and verify that all states in which mutual exclusion fails are unreachable. Instead, we will reason out the mutual exclusion property by semantic code analysis.

Theorem 9. *There does not exist a concurrent execution of Peterson's algorithm in which Alice and Bob are in the critical section (i.e., both their PCs have value 5) simultaneously.*

Proof. Assume, for contradiction, that both Alice and Bob are in the critical section at some point in time t. We consider two cases depending of which party was the last one to write to the `turn` variable before time t.

If this party was Alice, it means that Alice's condition 4 `turn == Bob` was satisfied before she entered her critical section. It means that Alice must have observed `Bwants` to be false just before she entered her critical section in her line 5. Bob's program must have been waiting in line 1 at this point. So part of the execution must have happened in this order:

1. Alice executes line 3 and reads `Bwants` value `False`.
2. Bob possibly executes part of his program starting in line 1.
3. Alice executes line 5 and enters her critical section.
4. Bob possibly executes more lines of his program.
5. Alice leaves her critical section.

Once Bob reaches line 3 of his program he must find that `Awants` is true (since Alice's program is somewhere between lines 3 and 5) and that `turn` equals `Alice` (since he just set it to that value in his line 2 and Alice did not modify its value in her lines 3-5). It follows that Bob has not entered his critical section while Alice is inside hers, contradicting our assumption.

If, on the other hand, Bob was the last party to write to `turn` before time t, then the situation is completely identical with the roles of Alice and Bob reversed, so we reach a contradiction by the same reasoning. □

As you can see, correctness proofs for concurrent executions can be quite tricky. What about deadlock? If you play with the programs for a bit you will find that there is no obvious ways to make them deadlock. So it is sensible to conjecture that Peterson's algorithm is in fact deadlock-free. But what does deadlock-freeness even mean in the language of propositional logic?

For simplicity let us work with a stronger property called *starvation-freeness.* (The difference is somewhat technical so I won't explain it.) Intuitively, starvation-freeness means that Alice will *eventually* get to do her critical work provided that she wants to. So now we have to explain what "eventually" means.

One condition we must impose is that Bob does not abort his execution, for instance in the middle of his critical work. In such circumstances, mutual exclusion prevents Alice from ever making it to her own critical section. Similarly, we should discount executions in which Alice aborts at some point in time.

Definition 10. An infinite sequence of transitions is *non-aborting* if it contains infinitely many transitions labeled by Alice and infinitely many transitions labeled by Bob.

Non-aborting does not mean that, say, Alice's program has to run infinitely often; her infinite sequence of transitions may all represent idle transitions waiting for her to execute line 1.

Definition 11. Let P be a predicate of states. We say that P *eventually holds* if every non-aborting sequence of transitions visits a state in which the predicate holds (starting from some fixed initial state).

We can now state and prove that Peterson's algorithm is starvation-free for Alice:

Theorem 12. *If Alice's PC has value 2 in any execution of Peterson's algorithm (she has just expressed interest in critical work), Alice's PC will eventually reach value 5 (she will enter her critical section).*

Proof. If Alice's PC has value 2 but does not eventually reach value 5, Alice must be running the loop 3-4 infinitely many times in sequence. Throughout the execution of this loop, `Awants` and `Bwants` must both be true and `turn` must equal `Bob`. Since Bob does not abort, he must execute infinitely many of his own transitions during this period. Since it is Bob's turn, Bob cannot be idling in his loop 3-4. If he is idling in line 1, then `Bwants` must be false and Alice will eventually exit her loop 3-4 and enter her critical section. Otherwise, Bob will eventually reach line 2, yield `turn` to `Alice`, and execute his loop 3-4. At this point Alice's condition 4 fails so she eventually moves on to her critical section, contradicting our assumption. □

Rigorous reasoning about concurrent programs is challenging and error-prone. There is a type of logic called *linear temporal logic* that talks about the evolution of state machines and is particularly suitable for expressing properties such as mutual exclusion and starvation-freeness. There are also algorithms that, given a state machine with transition labels (called a Büchi automaton) and a formula in linear temporal logic, decides if the formula is true for every execution of the automaton. (It is not a priori clear why such an algorithm should exist as it needs to verify a condition for a potentially infinite set of infinite sequences.) Such tools can in principle be used to automatize proofs of propositions like Theorem 9 and Theorem 12, as long as the set of states is finite. The SPIN verification software provides a practical implementation of such an algorithm together with a tool that converts concurrent program specifications into state machines.

References

This lecture is based on Chapters 5 and 6 of the text *Mathematics for Computer Science* by E. Lehman, T. Leighton, and A. Meyer. Material from slides by Prof. Lap Chi Lau were also used in the preparation. The section on Concurrency is partially based on Chapter 2 of the book *The Art of Multiprocessor Programming* by Maurice Herlihy and Nir Shavit.