Learning objectives:

- ☐ identify errors in inductive proofs
- ☐ prove correctness of recursive programs with induction

Last time, we introduced induction. Let's warm up by trying to identify errors in the following proof.

Example 1:

The following sentences are used to prove the following proposition. Put them in order, and correct any errors.

Prove that $7^n - 1$ is a multiple of 6 for all $n \geq 0$.

- Then there exists an integer b such that $7^k - 1 = 6b$.
- Because b is an integer, $7b + 1$ is an integer, so $p(k+1)$ is true.
- **Inductive step:** Let $k \geq 1$ and assume that $p(k)$ is true.
- Let the induction hypothesis be the predicate: $p(n) = 7^n - 1$ *is a multiple of* 6 *for all* $n \geq 0$.
- **Base case:** $p(1)$ is true because $7^1 - 1 = 6$, which is a multiple of 6 since $6 \times 1 = 6$.
- We use a proof by induction.
- Let the induction hypothesis $p(n)$ is true.
- Therefore, by induction on n, $p(n)$ is true for all $n \geq 0$.
- Multiplying both sides by 7, we get $7^{k+1} - 1 = 6(7b + 1)$.

Solution:

Proof. We use a proof by induction. Let the induction hypothesis be the predicate: $p(n) = 7^n - 1$ *is a multiple of* 6. We will prove that $p(n)$ is true for all $n \geq 0$.

- **Base case:** $p(0)$ is true because $7^0 - 1 = 0$, which is a multiple of 6 since $6 \times 0 = 0$.
- **Inductive step:** Let $n \geq 0$ and assume that $p(n)$ is true. Then there exists an integer b such that $7^n - 1 = 6b$. Multiplying both sides by 7 and adding 6 to both sides, we get $7^{n+1} - 1 = 6(7b + 1)$. Since b is an integer, then $7b + 1$ is also an integer, so $p(n+1)$ is true.

Therefore, by induction on n, $p(n)$ is true for all $n \geq 0$. ☐

Note that in the second sentence, it is incorrect to keep the *for all* quantifier, because $p(n)$ would no longer be a predicate in that case (it still needs to depend on the input variable n).

Be careful!

Be careful with your implications! It is incorrect to show that $p(k+1) \implies p(k)$.

1 Induction with sets

We've done a bunch of number-y examples, so let's do one with sets. This is good practice for the types of proofs we will do later with graphs.

Define the *power set* as the set of all possible subsets of a set. For example, for a set $A = \{a, b, c\}$, the power set is

$$\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Example 2:
Prove that the cardinality of the power set with n elements is $|\mathcal{P}(A)| = 2^n$.

Solution:
Proof. We use a proof by induction. Let the induction hypothesis be: $p(n)$ = *the cardinality of the power set with n elements is* $|\mathcal{P}(A)| = 2^n$.

Base case: Our base case is for $n = 0$, in which we have the single emptyset. Therefore, $|\mathcal{P}(A)| = 2^0 = 1$.

Inductive case: Assume $p(n)$ is true. Then the cardinality of the power set of a set with n elements is 2^n. Now, consider the set A_{n+1} with $n+1$ elements: $\{a_1, a_2, \ldots, a_n, a_{n+1}\}$. We want to show that $\mathcal{P}(A_{n+1}) = 2^{n+1}$. Remove the last element of A_{n+1}, to create a set with n elements: $A_n = \{a_1, a_2, \ldots, a_n\}$. By the definition of the power set, $\mathcal{P}(A_{n+1})$ includes every element in $\mathcal{P}(A_n)$ paired with a_{n+1}, along with every element in $\mathcal{P}(A_n)$:

$$\mathcal{P}(A_{n+1}) = \mathcal{P}(A_n) \cup \{x \cup a_{n+1} \mid x \in \mathcal{P}(A_n)\}.$$

The cardinality of $|\mathcal{P}(A_{n+1})|$ is the sum of the cardinalities of both sets, minus the cardinality of their intersection. Therefore,

$$\begin{aligned}
|\mathcal{P}(A_{n+1})| &= |\mathcal{P}(A_n)| + |\{x \cup a_{n+1} \mid x \in \mathcal{P}(A_n)\}| \\
&= 2^n + |\{x \cup a_{n+1} \mid x \in \mathcal{P}(A_n)\}| && \text{by } p(n) \\
&= 2^n + 2^n && 2^n \text{ elements of } \mathcal{P}(A_n) \text{ are paired with } a_{n+1} \\
&= 2 \cdot 2^n \\
&= 2^{n+1}
\end{aligned}$$

Therefore, by induction the cardinality of the power set of a set with n elements is 2^n. □

Mathematical induction has a lot of similarities with *recursion*. Remember, that when writing recursive programs, it is very important to make sure you have a **base case** and **recursive case**, similar to the base case and inductive steps used in a proof by induction. It is important to make sure your recursive programs work correctly, so we will now practice proving the correctness of a few recursive functions.

Consider the following pseudocode which describes a recursive solution for reversing a string.

```
reverseString(s)

    input: s (string)
    output: reversed string
1   if length(s) == 1     # base case
2       return s
3   else   # recursive case
4       return reverseString( s[2:length(s)] ) + s[1]
```

Algorithm 1: Recursive function for reversing a string. In the pseudocode, the string indexing starts at 1 (not 0, like Python or C-like languages). Elements (characters) of the string can be accessed with square brackets ([]), and a substring can be extracted with a colon (*start*: *end*) which includes *end*.

Pseudocode?

This is often useful when you want to describe a sequence of steps as you would in a programming language without restricting yourself to specific language. You can use basic keywords like **if**, **else**, **for**, **return** and also highlight when you might be calling a separate function. The focus of pseudocode is truly on the *algorithm* along with the corresponding inputs and outputs, not on the semantics of your code.

Example 3:
Prove that the **reverseString** function listed in Algorithm 1 is correct.

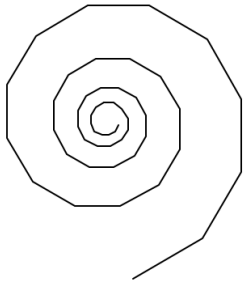
Solution:
Proof. We use a proof by induction. Let $p(n)$ be the predicate that **reverseString** correctly reverses an input string of length n. We will prove that **reverseString** correctly reverses strings for $n \geq 1$.

Base case: Consider strings of length $n = 1$. The reverse of this string is just the string itself, which Line 2 correctly returns.

Inductive case: Let $n > 1$ and assume that $p(n)$ is true. That is, **reverseString** correctly reverses strings of length n. Now consider a string of length $n + 1$. Since $n \geq 1$, the algorithm jumps to the recursive step on Line 4. Remove the first character from this string to create a string of length n and pass this into **reverseString**. By $p(n)$, then this string of length n is correctly reversed and we need only move the first character (which we removed to create a string of length n) to the end. This is what Line 4 does, so $p(n + 1)$ is true.

Therefore, by induction on the length of the input strings n, **reverseString** works correctly. □

In the last example, we proved the correctness of the **stringReverse** function. Sometimes, we want to prove our recursive function achieves some property.



Sample output of Algorithm 2 for $n = 50$, $L = 50$ and $\alpha = 0.95$.

Example 4:

Prove that the total length drawn by Algorithm 2 is $L\frac{1-\alpha^n}{1-\alpha}$, when called with n generations $n > 0$ and a factor $0 < \alpha < 1$.

Solution:

Proof. We use a proof by induction on the number of generations n. Let the induction hypothesis be $p(n) =$ *Algorithm 2 draws a total length of* $L\frac{1-\alpha^n}{1-\alpha}$.

Base case: Our base case is at $n = 0$, in which case nothing is drawn. Line 2 correctly draws nothing at $n = 0$, which agrees with $p(0) = L\frac{1-\alpha^0}{1-\alpha} = \frac{0}{1-\alpha} = 0$ since $\alpha \neq 0$.

Inductive step: Assume $p(n)$ is true. That is, the total length drawn by **spiral**$(n, L, \alpha) = L\frac{1-\alpha^n}{1-\alpha}$. Now consider the call **spiral**$(n+1, L, \alpha)$ for $n > 0$. We are now in the recursive case. Since $p(n)$ is true, then Line 6 draws a spiral of length $\alpha L\frac{1-\alpha^n}{1-\alpha}$. Don't forget the α in front because the L that is passed to **spiral** is αL. The only additional drawing happens on Line 5, which incurs an additional length of L. Therefore, the total length drawn at $n+1$ is

$$\alpha L\frac{1-\alpha^n}{1-\alpha} + L = L\frac{\alpha - \alpha^{n+1} + 1 - \alpha}{1-\alpha} L\frac{1-\alpha^{n+1}}{1-\alpha}.$$

Thus, by induction on the number of generations n, $p(n)$ is true. □

spiral(n, L, α)

```
   input: n: number of generations, L: length to draw,
   α: factor to decrease length in next generation
   output: none
1  if n == 0    # base case
2      return
3  else   # recursive case
4      turn_left(30 degrees)    # turns left by 30 degrees
5      draw_line(L)    # draws a straight line of length L
6      spiral(n − 1, αL, α)
```

Algorithm 2: Recursive function for drawing a spiral. Assume that the function **draw_line** draws a straight line of some input length L and **turn_left** turns the heading by some input angle (in degrees). This is very similar to the **forward** and **left** functions in Python's Turtle graphics module.