

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Dr. Liam O'Conn
University of Edinburgh, HFCS (an
Term 2, 2020

Add WeChat `edu_assist_pro`

Recap: Induction

Assignment Project Exam Help

Suppose we want to show that

$S n$.

Remember that to

Definition of Natural Numbers

- ① 0 is a natural number.
- ② For any natural number n , $n + 1$ is also a natural number.

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Recap: Induction

Therefore, to show $P(n)$ for all n , it suffices to show:

- 1 $P(0)$ (the *base case*), and
- 2 assuming $P(k)$ (the *inductive hypothesis*),
 $\Rightarrow P(k + 1)$

Example

Show that $f(n) = n^2$ for all $n \in \mathbb{N}$, where:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2n - 1 + f(n - 1) & \text{if } n > 0 \end{cases}$$

(done on iPad)

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Induction on Lists

Assignment Project Exam Help

Haskell lists can be defined similarly to natural numbers

Definition of Lists

- 1 `[]` is a list.
- 2 For any list

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Induction on Lists

Haskell lists can be defined similarly to natural numbers

Definition of Lists

- 1 $[]$ is a list.
- 2 For any list

This means, if we want to prove that a property

to show:

holds, it suffices

- 1 $P([])$ (the base case)
- 2 $P(x:xs)$ for all items x , assuming the inductive hypothesis $P(xs)$.

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Induction on Lists: Example

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = x `f` foldr f z xs
```

Example

Prove for all `ls`:

$$\text{sum } ls == \text{foldr } (+) \ 0 \ ls$$

(done on iPad)

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Custom Data Types

So far, we have seen type synonyms using the `type` keyword. For a graphics library, we might define:

```
type Point    = (Float, Float)
type Vector   = (Float, Float)
type Line     = (Point, Point)
type Colour   = (Int, Int, Int, Int) -- RGBA
```

```
movePoint :: Point -> Vector -> Point
movePoint (x,y) (dx,dy) = (x + dx, y + dy)
```

But these definitions allow Points and Vectors to be used interchangeably, increasing the **likelihood of errors**.

Product Types

We can define our own compound types using the data keyword:



```
data Point = Point  
    deriving (Show, Eq)
```

```
data Vector = Vector Float Float  
    deriving (Show, Eq)
```

```
movePoint :: Point -> Vector -> Point  
movePoint (Point x y) (Vector dx dy)  
    = Point (x + dx) (y + dy)
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Records

We could define Colour similarly:

```
data Colour = Colour Int Int Int Int
```

But this has so many

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Assignment Project Exam Help

Records

We could define `Colour` similarly:

```
data Colour = Colour Int Int Int Int
```

But this has so many

Haskell lets us decl

on the previous slide

tion style

```
data Colour = Colour { redC      :: Int
                      , greenC    :: Int
                      , blueC     :: Int
                      , opacityC  :: Int
                      } deriving (Show, Eq)
```

Here, the code `redC (Colour 255 128 0 255)` gives 255.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Enumeration Types

Similar to enums in C and Java, we can define types to have one of a set of predefined values:

```
data LineStyle
```

```
deriving (Show, Eq)
```

```
data FillStyle = SolidFill | NoFill
```

```
deriving (Show, Eq)
```

Types with more than one constructor are called *sum types*.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Algebraic Data Types

Just as the `Point` constructor took two `Float` arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:

```
data Picture0
  = Path    [Point]    Colour L
  | Circle  Point
  | Polygon [Point]    Colour LineStyle FillStyle
  | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
deriving (Show, Eq)
```

```
type Picture = [PictureObject]
```

<https://eduassistpro.github.io/>
Add WeChat [edu_assist_pro](#)

Live Coding: Cool Graphics

Assignment Project Exam Help
Example (Ellipses and Curves)

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Recursive and Parametric Types

Assignment Project Exam Help

Data types can also be defined with **parameters**, such as the well known Maybe type, defined in the stan

```
data Maybe a = Just a | N
```

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Recursive and Parametric Types

Assignment Project Exam Help

Data types can also be defined with **parameters**, such as the well known Maybe type, defined in the stan

```
data Maybe a = Just a | N
```

Types can also be re
could define them ourselves:

```
data List a = Nil | Cons a (List a)
```

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Recursive and Parametric Types

Assignment Project Exam Help

Data types can also be defined with **parameters**, such as the well known Maybe type, defined in the stan

```
data Maybe a = Just a | N
```

Types can also be re
could define them ourselves:

```
data List a = Nil | Cons a (List a)
```

We can even define natural numbers, where 2 is encoded as

```
data Natural = Zero | Succ Natural
```

<https://eduassistpro.github.io/>

Add WeChat [edu_assist_pro](#)

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

*Make illegal states **unrepresentable**.*

Choose types that
scenarios are elim

lure

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

*Make illegal states **unrepresentable**.*

Choose types that
scenarios are eliminated

<https://eduassistpro.github.io/>

Example (Contact Details)

```
data Contact = C Name (Maybe Address) (Maybe Email)
```

is changed to:

```
data ContactDetails = EmailOnly Email | PostOnly Address  
                  | Both Address Email  
data Contact = C Name ContactDetails
```

What failure state is eliminated here? Liam: also talk about other famous screwups

Partial Functions

Failure to follow Yaron's excellent advice leads to *partial functions*.

Definition

A *partial function* is a function not defined for all possible inputs.

Examples: `head`

Partial function
undefined cases a

<https://eduassistpro.github.io/>

Add WeChat `edu_assist_pro`

Partial Functions

Failure to follow Yaron's excellent advice leads to **partial functions**.

Definition

A **partial function** is a function not defined for all possible inputs.

Examples: `head`

Partial function
undefined cases a

To eliminate partiality, we must either:

- **enlarge** the codomain, usually with a `Maybe`

```
safeHead :: [a] -> Maybe a -- Q: How is th  
safeHead (x:xs) = Just x  
safeHead []     = Nothing
```

Add WeChat [edu_assist_pro](https://eduassistpro.github.io/)

<https://eduassistpro.github.io/>

Partial Functions

Failure to follow Yaron's excellent advice leads to **partial functions**.

Definition

A **partial function** is a function not defined for all possible inputs.

Examples: `head`

Partial function
undefined cases a

To eliminate partiality, we must either:

- enlarge the codomain, usually with a `Maybe`

```
safeHead :: [a] -> Maybe a -- Q: How is th  
safeHead (x:xs) = Just x  
safeHead []      = Nothing
```

- Or we must **constrain** the domain to be more specific:

```
safeHead' :: NonEmpty a -> a -- Q: How to define?
```

Type Classes

You have already seen functions such as:

- compare
- (==)
- (+)
- show

that work on **multiple types**, and their corresponding constraints: Ord, Eq, Num and Show

Ord,

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Type Classes

Assignment Project Exam Help

You have already seen functions such as:

- compare
- (==)
- (+)
- show

<https://eduassistpro.github.io/>

that work on **multiple types**, and their corresponding constraints
Eq, Num and Show

Ord,

Add WeChat edu_assist_pro

These constraints are called *type classes*, and can be thought of as
which certain operations are implemented.

Show

The Show type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where -- nothing to do with DP  
  show :: a -> String
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Show

The Show type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where  
  show :: a -> String
```

Types are added to t

```
instance Show Bool where  
  show True  = "True"  
  show False = "False"
```

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Show

The Show type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where  
  show :: a -> String
```

Types are added to t

```
instance Show Bool where  
  show True  = "True"  
  show False = "False"
```

We can also define instances that depend on other instances:

```
instance Show a => Show (Maybe a) where  
  show (Just x) = "Just " ++ show x  
  show Nothing  = "Nothing"
```

Fortunately for us, Haskell supports automatically deriving instances for some classes, including Show.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Read

Assignment Project Exam Help

Type classes can also overload based on the type **returned**, unlike similar features like Java's interfaces

```
class Read a where  
  read :: String
```

Some examples:

- `read "34" :: Int`

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Read

Assignment Project Exam Help

Type classes can also overload based on the type **returned**, unlike similar features like Java's interfaces

```
class Read a where  
  read :: String -> a
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char`

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Read

Assignment Project Exam Help

Type classes can also overload based on the type **returned**, unlike similar features like Java's interfaces

```
class Read a where  
  read :: String -> a
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char` Runtime error!
- `show (read "34") :: String`

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Read

Assignment Project Exam Help

Type classes can also overload based on the type **returned**, unlike similar features like Java's interfaces

```
class Read a where  
  read :: String
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char` Runtime error!
- `show (read "34") :: String` Type error!

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\circ : S \rightarrow S \rightarrow S$ where the operation

- is *associative*.

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\circ : S \rightarrow S \rightarrow S$ where the operation \circ is *associative*.

Associativity is de

<https://eduassistpro.github.io/>

Haskell has a type class for semigroups! The associativity law is a programmer discipline:

```
class Semigroup s where
  (<>) :: s -> s -> s
  -- Law: (<>) must be associative.
```

What instances can you think of?

Semigroup

Assignment Project Exam Help

Lets implement additive colour mixing:

```
instance Semigroup Colour where
```

```
  Colour r1 g1 b1 a
```

```
    = Color
```

```
      (mix b1 b2)
```

```
      (mix a1 a2)
```

```
  where
```

```
    mix x1 x2 = min 255 (x1 + x2)
```

Observe that associativity is satisfied.

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Monoid

Monoids

A *monoid* is a semigroup (S, \bullet) equipped with a special *identity element* $z \in S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y .

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Monoid

Monoids

A *monoid* is a semigroup (S, \bullet) equipped with a special *identity element* $z \in S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y .

```
class (Semigr  
  mempty :: a
```

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Monoid

Monoids

A *monoid* is a semigroup (S, \bullet) equipped with a special *identity element* $z \in S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y .

```
class (Semigr
```

```
  mempty :: a
```

For colours, the ide

```
instance Monoid Colour where
```

```
  mempty = Colour 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Monoid

Monoids

A *monoid* is a semigroup (S, \bullet) equipped with a special *identity element* $z \in S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y .

```
class (Semigr
```

```
  mempty :: a
```

For colours, the ide

```
instance Monoid Colour where
```

```
  mempty = Colour 0 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are *not* monoids?

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Monoid

Monoids

A *monoid* is a semigroup (S, \bullet) equipped with a special *identity element* $z \in S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y .

```
class (Semigr
```

```
  mempty :: a
```

For colours, the ide

```
instance Monoid Colour where
```

```
  mempty = Colour 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are *not* monoids?

Non-empty lists, maximum

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Newtypes

Assignment Project Exam Help

There are multiple possible monoid instances for numeric types like Integer:

- The operation (+) is associative, with identity element 0
- The operati

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Newtypes

Assignment Project Exam Help

There are multiple possible monoid instances for numeric types like Integer:

- The operation (+) is associative, with identity element 0
- The operation (*) is associative, with identity element 1

Haskell doesn't use a single monoid class in the entire program

A common technique is to define a separate type that is representative of the original type, but can have its own, different type class instance

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Newtypes

There are multiple possible monoid instances for numeric types like Integer:

- The operation (+) is associative, with identity element 0
- The operation (*) is associative, with identity element 1

Haskell doesn't use a single monoid class in the entire program.

A common technique is to define a separate type that is representative of the original type, but can have its own, different type class instance.

In Haskell, this is done with the `newtype` keyword.

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Newtypes

A newtype declaration is much like a data declaration except that there can be only one constructor and it must take exactly one argument:

```
newtype Score = S Integer
```

```
instance Semi  
  S x <> S y = S (x + y)
```

```
instance Monoid Score where  
  mempty = S 0
```

Here, `Score` is represented identically to `Integer`. The cost of the conversion is incurred to convert between them.

In general, newtypes are a great way to prevent mistakes. Use them frequently!

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should i

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where  
  (<=) :: a -> a -> Bool
```

What laws should i

For all x, y, and

① *Reflexivity* $x \leq x$

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where
```

```
  (<=) :: a -> a -> Bool
```

What laws should i

For all x , y , and

① *Reflexivity* $x \leq x$

② *Transitivity*: If $x \leq y$ and $y \leq z$, then $x \leq z$

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where  
  (<=) :: a -> a -> Bool
```

What laws should it satisfy?

For all x, y , and z :

- 1 *Reflexivity*: $x \leq x$
- 2 *Transitivity*: If $x \leq y$ and $y \leq z$, then $x \leq z$
- 3 *Antisymmetry*: If $x \leq y$ and $y \leq x$, then $x = y$

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should i

For all x, y, and

- ① *Reflexivity*: $x \leq x$
- ② *Transitivity*: If $x \leq y$ and $y \leq z$, then $x \leq z$
- ③ *Antisymmetry*: If $x \leq y$ and $y \leq x$, then $x = y$
- ④ *Totality*: Either $x \leq y$ or $y \leq x$

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Ord

Ord is a type class for inequality comparison:

```
class Ord a where  
  (<=) :: a -> a -> Bool
```

What laws should it satisfy?

For all x, y , and z :

- 1 *Reflexivity*: $x \leq x$
- 2 *Transitivity*: If $x \leq y$ and $y \leq z$, then $x \leq z$
- 3 *Antisymmetry*: If $x \leq y$ and $y \leq x$, then $x = y$
- 4 *Totality*: Either $x \leq y$ or $y \leq x$

Relations that satisfy these four properties are called *total orders*. Without the fourth (totality), they are called *partial orders*.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where  
  (==) :: a -> a -> Bool
```

What laws should i

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where
  (==) :: a -> a -> Bool
```

What laws should i

For all x, y, and

① Reflexivit

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where  
  (==) :: a -> a -> Bool
```

What laws should i

For all x, y, and

- 1 *Reflexivity*
- 2 *Transitivity*: If $x == y$ and $y == z$ then $x ==$

Add WeChat edu_assist_pro

<https://eduassistpro.github.io/>

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where  
  (==) :: a -> a -> Bool
```

What laws should i

For all x, y, and

- 1 *Reflexivity*
- 2 *Transitivity*: If $x == y$ and $y == z$ then $x == z$
- 3 *Symmetry*: If $x == y$ then $y == x$

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where  
  (==) :: a -> a -> Bool
```

What laws should i

For all x, y, and

- 1 *Reflexivity*
- 2 *Transitivity*: If $x == y$ and $y == z$ then $x == z$
- 3 *Symmetry*: If $x == y$ then $y == x$

Relations that satisfy these are called *equivalence*

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where
  (==) :: a -> a -> Bool
```

What laws should i

For all x, y, and

- ① *Reflexivity*
- ② *Transitivity*: If $x == y$ and $y == z$ then $x == z$
- ③ *Symmetry*: If $x == y$ then $y == x$

Relations that satisfy these are called *equivalence*

Some argue that the Eq class should be only for *equality*, requiring stricter laws like:

If $x == y$ then $f\ x == f\ y$ for all functions f

But this is debated.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Types and Values

Assignment Project Exam Help

Haskell is actually

- The *value* of `3` is `3` etc

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Types and Values

Assignment Project Exam Help

Haskell is actually

- The *value* <https://eduassistpro.github.io/> et 3 etc
- The *type*-like String,
and type *constructors* like Maybe, (->), []

Add WeChat edu_assist_pro

Types and Values

Assignment Project Exam Help

Haskell is actually

- The *value* <https://eduassistpro.github.io/> et 3 etc
- The *type*-like String,
and type *constructors* like Maybe, (->), []

This type level language itself has a type system!

Add WeChat edu_assist_pro

Kinds

Assignment Project Exam Help

Just as terms in the λ language are given

The most basic kind

- Types such as Int
- Seeing as `Maybe` is parameterised by one argument, given a type (e.g. `Int`), it will return a type (e.g. `Maybe Int`)

$* \rightarrow *$:

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Lists

Assignment Project Exam Help

Suppose we have a function.

```
toString :: Int -> String
```

And we also have a fu

```
getNumbers :: [Seed] -> [Int]
```

How can I compose toString with getNumbers to get a function f of type Seed -> [String]?

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Lists

Assignment Project Exam Help

Suppose we have a function.

```
toString :: Int -> String
```

And we also have a fu

```
getNumbers :: [Seed] -> [Int]
```

How can I compose toString with getNumbers to get a function f of type Seed -> [String]?

Answer: we use map:

```
f = map toString . getNumbers
```

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Maybe

Suppose we have a function

Assignment Project Exam Help

```
toString :: Int -> String
```

And we also have a fu

```
tryNumber :: Seed -> Maybe String
```

How can I compose of type Seed ->

Maybe String?

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Maybe

Suppose we have a function

```
toString :: Int -> String
```

And we also have a fu

```
tryNumber :: Seed -> Maybe String
```

How can I compose

Maybe String?

We want a map function but for the Maybe type

```
f = maybeMap toString . tryNumber
```

Let's implement it.

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Functor

All of these functions are in the interface of a single type class, called **Functor**.

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Functor

All of these functions are in the interface of a single type class, called **Functor**.

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous t

ctor is over types

of kind $* \rightarrow *$

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Functor

All of these functions are in the interface of a single type class, called `Functor`.

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous types

`Functor` is over types

of kind `* -> *`

<https://eduassistpro.github.io/>

Instances for:

- Lists
- Maybe
- Tuples (how?)
- Functions (how?)

Add WeChat edu_assist_pro

Demonstrate in live-coding

Functor Laws

Assignment Project Exam Help

The functor type class must obey two laws:

Functor Laws

- ① `fmap id == id`
- ② `fmap f . fmap g == f`

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro

Functor Laws

Assignment Project Exam Help

The functor type class must obey two laws:

Functor Laws

- 1 `fmap id == id`
- 2 `fmap f . fmap g == f`

In Haskell's type system, it's impossible to make a total `fmap` that satisfies the first law but violates the second.

This is due to *parametricity*, a property we will return to in Week 8 or 9

<https://eduassistpro.github.io/>
Add WeChat edu_assist_pro

Homework

Assignment Project Exam Help

- 1 Do the first project, due in exam week
- 2 Last week's quiz is also up
- 3 This week's quiz is also up, due next friday (the friday after the exam)

Add WeChat edu_assist_pro