```haskell
import Control.Monad
import System.Random
import Data.Word
import qualified Data.ByteString as S
import qualified Data.ByteString.Lazy as B
-- INPUT AND OUTPUT --

-- Protip: To run a program you can either compile it and then run the produced executable file by doing
-- ghc --make helloworld and then ./helloworld
-- or you can use the runhaskell command like so: runhaskell helloworld.hs and your program will be executed on the fly.


-- HELLO, WORLD! --

-- Created helloworld.hs in Programs. We define a function "main" to do the function putStrLn with the parameter "Hello, World!"
-- We then compile it like "$ ghc --make [filename minus .hs]" so "$ ghc --make helloworld" and when we run the unix executible file,
-- it tells us "Hello, World!" in the Terminal.

-- The putStrLn function is defined like this:
{-
putStrLn :: String -> IO ()
-}
-- So it takes a String and returns an I/O action, which is something like reading or writing to the Terminal screen. The I/O will
-- also contain some return value in it.

-- We can use the <- operator to get a value from a function and bind it to a varaible. So name <- getLine would bind the result of you entering
-- something to the variable "name". Think of it like a box. The <- allows you to access what's inside the box that is allowed to go outside the
-- "clean" place that is the code.

-- Created fortune.hs.
-- We include a function in the putStrLn because it just returns a string, so we can print it back out no problem.
-- You can't say something like "putStrLn "hello there" ++ getLine" because getLine returns type IO String and not String, so you can't combine.
-- Every IO action has a result that is associated with it. We could have written foo <- putStrLn "What's your name?" but it wouldn't have done
-- anything. But we could have done it.

-- In a 'do' block, the last action can't be bound to anything.
-- You can use IO actions in GHCI just fine without having to compile a program.

-- Created toupper.hs
-- We can use let bindings just like we did in list comprehentions in that we don't need a 'where' as well. So we can just define a new
-- expression and then use it later on in the 'do' block.
-- It's good practice to line things up correctly indentation-wise becaue indentation is important in Haskell.

-- Created reverse.hs
-- Now we're going to create a program that will reverse the words that we put in and then die when we enter a blank line.

-- In the main, we have an if/else. Every if must have an else because every expression must have a result.
-- If our case (that no line was typed) is true, it returns an IO action, and it also returns another IO action if the line is filled with
-- something. That's why your if/else expressions need to have an IO action on both sides. if CONDITION then IO ACTION else IO ACTION.

-- In our "else" we need to have exactly 1 IO Action, so we use a 'do' block to encapsulate that.

-- We also have a "return" when there is no line to do anything with. This is nothing like languages like Java, C, or PHP. In Haskell, return
-- makes an IO action out of a pure value. So calling 'return "hahahahaha"' would give us type IO String. It doesn't do anything or cause a do
-- block to end. The program is more than happy to continue on running after even 1000 returns.
-- Return is like the oposite of <-. return puts a value into a box, and <- takes it out of the box.
main = do
        a <- return "heck"
        b <- return "yeah"
        putStr' $ a ++ " " ++ b
-- For example, this would output "heck yeah"

-- When dealing with IO do blocks, we mostly use return either to create an IO action that doesnt' do anything or because we don't want the IO
-- action that's made up from a do block to have the result value of its last action, but we want it to have a different value, so we make a
-- return that always has our desired type as a result and we put it at the end.

-- putStr is like putStrLn but it doesn't put an extra new line at the end.
-- putStr :: String -> IO ()

-- putStr is defined recursively with the help of putChar.
-- putChar just prints out a char.

-- Let's make putStr!
putStr' :: String -> IO ()
putStr' [] = return ()
putStr' (x:xs) = do
        putChar x
        putStr' xs

-- print takes a value that derives show and prints it to the terminal. When we are using GHCI, it's using print to print things out to the
-- screen. print is basically putStrLn . show

-- Created getchartest.hs
-- getChar is like getLine, but it only gets a character.
-- Due to buffering, it doesn't actually run the program until we press the enter key. But once we press it, it's like we're just entering
-- things as we did.

-- We can use Control.Monad with this as well to use the 'when' function. Imported Control.Monad at the top. This is interesting because in a
-- 'do' block, it looks like a control flow statement, but it's actually a normal function. It takes a boolean value and an IO action. If
-- True, then it does the IO action, otherwise it return ()

-- Created getchartestmonad.hs
-- So using 'when' is good for encapsulating the 'if this then IO action else return ()'

-- Created sequence.hs
-- sequence takes a list of IO actions and then calls those IO actions one after another.
-- sequence :: [IO a] -> IO [a]
{-
main = do
        a <- getLine
        b <- getLine
```

```
        c <- getLine
        print [a,b,c]

is exactly the same as doing

main = do
        rs <- sequence [getLine, getLine, getLine]
        print rs

-}

-- mapM and mapM_ are two functions that do mapping over lists. Because mapping an IO action over a list of so common, these two funcitons were
-- created. mapM takes a function and a list, maps the function over the list, and then sequences it.
-- mapM_ takes a function and a list and maps the function over the list, but gets rid of the results when we sequence it. We usually use mapM_
-- when we don't care about the result our sequence has.

-- Created capslocked.hs
-- The forever function takes an IO action and returns that IO action over and over again forever.

-- Created a forM.hs
-- forM if a function that takes a list of stuff and a function. It then maps the function over the list and sequences it. So it's like mapM but
-- the parameters are switched.
-- The lambda in the file is a function that takes a number and returns an IO action.
-- We can think of forM as "make an IO action for every element in a list." What each IO action is will depend on the element that was used to
-- make the action. It then preforms those actions and binds the result to something or throws it away.


-- FILES AND STREAMS --

-- Let's meet getContents. This is an IO action that reads everything from the standard input until it encounters an end-of-file character.
-- What's cool about getContents is that it does lazy IO. So when you say foo <- getContents it doens't just read it and store it in memory.
-- It instead says "okay whatever. I'll read the contents later when you actually need to use it."

-- This is useful for when we're piping output form one program into our program.

-- Created getcontentshaiku.hs
-- We can run this and give it some input by using 'cat' and a file name then piping it with | into the program. We can also run it normally
-- with runhaskell or ./ if we've compiled it.

-- When the result of getContents is bound to a value, it isn't represented in memory as a real string. It's more of a promise that it will
-- produce a string eventually. When we map toUpper over 'contents' in our getcontentshaiku.hs, it's also a promise that it will be mapped
-- over the eventual 'contents'. This is also going on with putStrLn. putStrLn is asking for some capslocked line, and so it tells that to
-- contents, which tells it to getContents, which then gets stuff from the Terminal.

-- Created lessthan10.hs
-- This is a program to take in lines but only print them if they are less than 10 characters.
-- This pattern of getting contents from a file and piping it into a program and then calling a function to intereact with that stuff is so
-- common that there is actually a function for it.

-- The interact function has type String -> String as a parameter and returns an IO action that will take some input, run that function on the
-- input, and then print the function's results. Let's modify the lessthan10.hs to do that.
-- There.
-- We could even write the function like
{-
main = interact $ unlines . (filter (<10) . length)) . lines
-}
-- There, now it's only one line.
-- Interact is mainly used when you're wanting to give a program some input and then get an output from it.

-- Created palindrome.hs
-- Let's make a program that continuously reads a line and then tells us if it's a palindrome or not. We could just use getLine and then
-- go back to main every time, but we can use interact.

-- Even though we made a program that just transforms one big string input into another, it acts like we did it line by line. This is because
-- Haskell is lazy and it wants to do the first line of the first string, but since it can't it waits until it has the first line and then
-- gives us an output right away.
-- We can also run the program by piping a text file into it.
-- $ cat words.txt | runhaskell palindromes.hs

-- Nothing is eaten when we type in an input to the lazy IO until it absolutely has to be because what we want to print right then depends on
-- the input.

-- Created girlfriend.hs
-- Now we can learn about reading from and writing to files.
-- Our program is several IO actions that are glued together with a do block.
-- The first function we encouter is the openFile function, which is new to us. It has a type of FilePath -> IOMode -> IO Handle
-- FilePath is just a type synonym for String
-- IOMode is defined like this
-- data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
-- Finally, openFile returns an IO Action that will open the specified file in the specified mode. If we bind that action to something, we get
-- what is called a handle. A value of type Handle represents where our file is.

-- In the next line, we see a function called hGetContents, which takes a Handle so it knows where to get the contents from and returns an IO
-- String
-- The function is pretty much like getContents, only getContents reads from the standard input while hGetContents gets from an actual file.
-- It works the same as getContents as in if we had a 3GB file, it wouldn't put it into memory and clog us up. Instead, it would read from the
-- file as needed.

-- Note the difference between a Handle and the contents. A handle is just something we refer to our file with, while the contents represents
-- what is actually in the file. If it were a book, the handle would be your bookmark showing where you are, while the contents is the actual
-- chapter.

-- We then print the contents of the file to the screen and then close it with hClose, which takes a handle and returns an IO action that closes
-- the file

-- Another way of doing what we just did is with withFile. The type of witFile is FilePath -> IOMode -> (Handle -> IO a) -> IO a
-- It takes the path of a file an, IOMode, and then a function which takes a handle and returns an IO action. What it returns is an IO action
-- that will open the file, do something with it, and then close it.
-- girlfriend2.hs is our previous program written with withFile.
-- I've also included our own rendition of withFile as an added bonus!

-- Just like we have hGetContents, we also have hGetLine, hGetChar, hPurStrLn, and hPutStr, etc.
```

```
-- For example, hPutStrLn will take a handle and a string. With this handle, it will write to a file the String that it was given, and then add
-- a new line after it. In the same vein, hGetLine will take a handle and return an IO action that reads a line from a file.

-- Loading files and then treating their contents as string is so common, that we have three nice functions to help us out.
-- readFile has type FilePath -> IO String. So it will take a file name and read that file lazily and bind its contents to something as a String

-- writeFile has a type of FilePath -> String -> IO (). IF A FILE EXISTS WITH THE NAME, IT WILL BE ERASED BEFORE WRITING HAPPENS
-- Created girlfriendtocaps.hs
-- In there is a main that will take the girlfriend.txt and make it into caps.

-- appendFile has the type of FilePath -> String -> IO ()
-- appendFile is like writeFile, but it doens't clear the file and make it 0 before adding to it.
-- Created appendtodo.hs in order to append to our todo.txt

-- We can change the size of the buffer ourselves instead of letting the computer decide either 1 line or whatever the OS thinks is cool.
-- hSetBuffering takes a Handle and a BufferMode and returns an IO action that sets the buffering. The values of the BufferMode are either
-- NoBuffering, LineBuffering, and LineBuffering (Maybe Int). The Maybe Int is just how big the chunk should be in bytes. If it is Nothing, the
-- OS determines the chunk size. NoBuffering sucks because it needs to access the disk one character at a time.

-- Created bufferingread.hs
-- This program is the same as the reading program we had before, but this time it's reading in chunks of 2048 bytes.

-- Reading files in bigger chunks can help us if we want to minimize disk access or if we are reading over a slow network resource.

-- We can also use hFlush, which is a function that takes a handle and returns an IO action that will flush the buffer of the file associated
-- with the handle. With line-buffering, the buffer is flushed after every line, and with chunk-buffeirng, it is flushed after every chunk.
-- The buffer is also flushed after closing a handle. We can use hFlush to force the reporting of data so far.
-- We can think of it as a toilet. Your toilet is set to flsh after 1 gallon of water is added to it. When it is flushed, the water (data) in
-- it is read, and the bowl begins filling again. But you can manually flush it by pressing the lever and forcing the water to be read even
-- if it isn't 1 gallon.

-- Created deletetodo.hs
-- We've already created a program to add to our list, but what about a program to take away from our list of things to do?
-- In here, we first get the handle of our file and bind it to 'handle'
-- Second, we use a function we haven't used before. That function is openTempFile. It takes a directory (in this case ".") and a file name.
-- It will create a file called 'temp' along with some random characters. It returns an IO action that creates a temporary file and also a
-- pair of values that is the temporary file name and also the handle.

-- Next, we bind the contents of todo.txt to 'contents', then split it into a list of strings with 'lines'. Then, we zip the numbers 0.. with the
-- list item that is a task to do. We then make the list of new strings into one and print it to the terminal before asking the user which item
-- they would like to remove.

-- Then, we read in a number and turn it into an Int with 'read'
-- Once we have done that, we can use delete to remove the n-th value from the lined list of todo.txt. After we remove it, we write that contents
-- to the temp file that was created earlier. We do that by using hPutStr [HANDLE] [STRING]

-- Once it's written to the temp file, we delete the todo.txt and then rename the temp file to 'todo.txt'
-- BE CAREFUL THOUGH, SINCE removeFile AND renameFile BOTH TAKE DIRECTORIES. NOT HANDLES.


-- COMMAND LINE ARGUMENTS --

-- Created todo.hs
-- We have a file to append to our todo.txt, we ahve one to remove something from it, but all those imply that we only have one todo list and
-- that the user has access to calling the specific .hs files and stuff. What we want to do is create a program that can take arguments for
-- adding, removing, and view.

-- Created arg-test.hs
-- First we need to learn about taking arguments.
-- The System.Environment module has two interesting functions. One of them is getArgs and one is getProgName. getArgs has type IO [String]
-- getProgName has the type of IO String and just has the program name.
-- First, we bind the arguments to args and the program name to progName

-- Now back to todo.hs
-- We first define the dispatch function, which just returns an association list that has both the argument you can give and a function to go
-- with it.


-- RANDOMNESS --

-- The System.Random module has all the functions we need in otder to generate semingly random data. Because Haskell is a functional language,
-- that means that if we give a certain function a parameter twice, it needs to return the same thing every time. This is a problem beucase if
-- we call a random function twice, it'll have to return the same random data twice. But what we can do is have it take random stuff as a
-- parameter. Stuff like the time, the date, the temperature of teh CPU, etc. so that we can generate random data.

-- The random function has type random :: (RandomGen g, Random a) => g -> (a, g)
-- RandomGen is a typeclass for types that can act as SOURCES of random, and the Random typeclass is for types that can take on random values.
-- A Boolean type can take on random values, namely True and False.

-- To use the random function, we need to get a random generator somehow. The System.Random module exports a type called StdGen, which is an
-- instance of the RandomGen typeclass. We can make one manually or tell the computer to make one.

-- mkStdGen function has the type mkStdGen :: Int -> StdGen. It takes an Int and gives us a random generator.
giveRandom :: Int -> (Int, StdGen)
giveRandom a = random $ mkStdGen a :: (Int, StdGen)
-- We need to tell Haskell which types we want to return because it doesn't know what instance of Random we want to reuturn!
-- If we call giveRandom on the same number twice, we get the same result each end every time, but if we use a differnet number, we get
-- different results.

-- The reason we get back a new StdGen is so that we can give it to another random function if we want to.
-- Let's make a coin toss function.
coinToss :: StdGen -> (Bool, Bool, Bool)
coinToss gen =
        let
                (firstCoin, newGen) = random gen
                (secondCoin, newGen') = random newGen
                (thirdCoin, _) = random newGen'
        in
                (firstCoin, secondCoin, thirdCoin)
-- The reason we don't need to do (Bool, StdGen) is because Haskell can infer that we want to return Bool from the type declaration.

-- What if we want to return 5 or 6 or 7 coin flips?
```

```haskell
multiCoinToss :: StdGen -> Int -> [Bool]
multiCoinToss gen num = take num $ randoms gen :: [Bool]
-- 'randoms' takes a StdGen and returns an infinite list of random values. We use 'take' to take X values from that list.

-- Implimentation of 'randoms'
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
-- Because it's an infinite list, we can't give a new random generator back.

-- We can make a function to give a number of values and a new generator though.
finiteRandoms :: (RandomGen g, Random a, Num n, Eq n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms num gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (num-1) newGen
        in  (value:restOfList, finalGen)

-- How about if we want a value within some sort of range? We can use the randomR function to generate a value within some range.
randomR' :: (Random a, RandomGen g) => a -> a -> g -> (a, g)
randomR' a b gen = randomR (a,b) gen

-- There's also randomRs to produce an infinite stream of stuff.
randomRs' :: String
randomRs' = take 10 $ randomRs ('a','z') (mkStdGen 10) :: String

-- System.Random offers the getStdGen function, which returns IO StdGen
-- Created randomFun.hs
-- Because it gets a new StdGen each time the program runs, it works out and gives us a different resiults each time.
-- Preforming getStdGen twice in the same program will just return the same StdGen, so you need to be mindful of that.

-- One way to get 2 different strings that are 20 characters long is to take 20 from the stream, and then take 20 more from the same stream.
-- Created gettwostrings.hs
-- We use the splitAt function from Data.List to split the stream at 20 two times and then just print out the values. Easy.

-- Created gettwostringsnew.hs
-- Another way to do the above is to use the newStdGen function. This splits our current generator into two new generators. It updates the
-- global generator with the new one, and it encapsulates the new generator as a result.
-- One we run newStdGen, it updates the global StdGen, so if we called getStdGen again, it would give us a new StdGen!

-- Created guessthenumber.hs
-- We could do it like we have it, or we could put it all in the main function. I think it's better to have it the way we currently have it
-- so that we can have a function that can be use somewhere else at a later time.


-- BYTESTRINGS --
-- Bytestrings are like lists, but they are only 1 byte (8 bits) long. The way they handle laziness is also different.
-- The two types of byetstrings are the strict ones and the lazy ones.
-- Strict Bytestrings are in Data.Bytestring and there are no promises involved. You can't have an infinite list of strict sytestrings because
-- once you evaluate the first element in the list, you have to evaluate the whole thing. The upside is that it can be faster, but it's also
-- likely to fill up your memory faster due to it evaluating the text all at once.

-- The other type is in Data.Bytestring.Lazy and it's more lazy. These are lazy, but they aren't as lazy as lists are. Each element is stored in
-- chunks. Each chunk is 64K, which will fit into a computer's CPU cache.

-- These two modules contain functions that are also found in Data.List, but they use Bystestring instead of [a] and Word8 instead of a.
-- Imported Lazy as B and Strict as S. Also Data.Word for Word8

-- The pack function has type pack :: [Word8] -> ByteString. We can think of it as taking a list, which is lazy, and making it less lazy.
pack' :: [Word8] -> B.ByteString
pack' a = B.pack a
-- The Word8 type is like Int, but it has a much smaller range. That range is anywhere from 0 to 255, and it represents an 8-bit number. It's
-- just like Int in that it's also a part of the Num typeclass.

-- Unpack is the opposite of pack. You give it a bytestring and it returns a list of Word8.

-- fromChunks takes a list of strict ByteStrings and converts it into a lazy ByteStrings. toChunks does the opposite.
fromChunks' :: B.ByteString
fromChunks' = B.fromChunks [S.pack [40..42], S.pack [43..45], S.pack [46..48]]
-- this is good if you have a lot of strict bytestrings, but you don't want to join them together in memory all at once.

-- The bytestring version of : is 'cons'. It takes a byte and a bytestring and puts the byte at the beginning of the bytestring. It's lazy, so it
-- will make a new bytestring even if the one you provide is not full. That's why it's good to use the strict version (con')
cons'' :: B.ByteString
cons'' = B.cons' 85 $ B.pack [80..84]

-- empty makes an empty bytestring.

-- It also has some functions that are the same and behave the same as the ones in System.IO. The only difference between the two is that String
-- is just replaced with ByteString.
-- Created bytestringcopy.hs
```