

Week 6

## Ch 9: More Input and More Output Ch 10: Functionally Solving Problems

---

University of the Fraser Valley

Dr. Russell Campbell

[Russell.Campbell@ufv.ca](mailto:Russell.Campbell@ufv.ca)

COMP 481: Functional and Logic Programming

# Assignment Project Exam Help

1

<https://powcoder.com>

## Add WeChat powcoder

### Chapter 9

- Reading and Writing Files
- The Bracket Function
- Writing with Handles
- To-Do Function
- Command-Line Arguments
- Handling Bad Input
- Randomness
- Toss a Coin
- Sequences of Random Numbers
- Randomness and I/O
- ByteStrings
- Strict and Lazy ByteStrings
- Copying Files with ByteStrings

### Chapter 10

- Reverse Polish Notation Calculator
- Writing an RPN Function
- Adding More Operations
- Heathrow to London
- Using Input for the Road System

2

## Functor Operations Example (David Semke)

```
import Data.List

-- Get a list of unique sums from the sums of all possible pairs
of items from list1 and list2

list1 = [5, 6, 7, 8]
list2 = [1, 2, 3, 4]

method1 = nub [a+b | a <- list1, b <- list2]

method2 = nub $ fmap (+) list1 <*> list2

method3 = nub $ (+) <$> list1 <*> list2

method4 = nub $ pure (+) <*> list1 <*> list2

method5 = nub $ [(+)] <*> list1 <*> list2
```

# Assignment Project Exam Help

3

<https://powcoder.com>

Add WeChat powcoder

## Files and Streams

Streams are sequences of data supplied over time.

- similar to characters supplied by keyboard as the user types them
- not much random access, only the next character in the sequence

4

## Reading File Input

Save the following text data as `haiku.txt` in a subfolder named `data`:

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Then save the following script as `capslocker.hs`:

```
{- ### capslocker.hs v1 ### -}
import Control.Monad
import Data.Char

main = forever $ do
  l <- getLine
  putStrLn $ map toUpper l
```

## Assignment Project Exam Help

5

<https://powcoder.com>

## Add WeChat powcoder

## Command-Line Compile and Execute

Compile the program and execute with a redirect:

```
stack ghc capslocker.hs
capslocker < data\haiku.txt           (Windows)
./capslocker < data/haiku.txt         (Unix-like)
```

Choose the above execute based on your platform.

- in Windows, there is no need for the `./` prefix
- the redirect `<` takes the contents of the `haiku.txt` file one line at a time and pipes it into standard input
- this is the same as if we had typed it on the keyboard while `capslocker` executed

6

## getContents (1)

We can simplify the ``capslocker`` program by making use of ``getContents`` function.

- ``getContents`` does lazy I/O in Haskell
  - it gets the input when required later, instead of right away
  - it reads standard input until it gets an end-of-file character
  - its type is ``getContents` :: IO String``
  - it takes care of how much input to get for us, unlike ``forever``

```
{- ### capslocker.hs v2 ### -}
import Data.Char

main = do
    contents <- getContents
    putStr $ map toUpper contents
```

# Assignment Project Exam Help

7

<https://powcoder.com>

## Add WeChat powcoder

## getContents (2)

To run ``capslocker`` without redirected input, user input will continue to get echoed as capitalized messages to stdout:

- to stop the program, send it an end-of-file signal
- in Windows, press ``CTRL-Z`` and then press ``ENTER``
- in other platforms, press ``CTRL-D``

Some notes about execution involving ``getContents``:

- ``getContents`` makes a promise to keep processing more lines of input, one line at a time
- it does not process a line in the future until it is typed
- because ``contents`` is bound to ``getContents``, it acts similarly
- because ``map toUpper`` takes ``contents`` as input, it also promises to execute per line as entered
- they all repeat execution for each line until an end-of-file character is entered

8

## \*lines and unlines

We can practice lazy input like this more with a program that restricts which input should be displayed depending on length:

```
{- ### shortLinesOnly.hs v1 ### -}
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly =
    unlines . filter (\line -> length line < 10) . lines
```

- `lines` and `unlines` functions behave like `words` and `unwords`
- but for input separated by end-of-line characters split into elements of a list and back

## Assignment Project Exam Help

9

<https://powcoder.com>

## Add WeChat powcoder

## \*Execute shortLinesOnly

Save the following text in the `data` folder as `short.txt`:

```
{- ### data\short.txt ### -}
i'm short
so am i
i am a loooooooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooooooooooooooong
short
```

\* go ahead and redirect the above text file into `shortLinesOnly`

10

## \*shortLinesOnly version 2

The previous implementation has a useful function to do the same kind of operations on each line of input:

```
{- ### shortLinesOnly.hs v2 ### -}
main = interact shortLinesOnly
```

```
shortLinesOnly :: String -> String
shortLinesOnly =
    unlines . filter (\line -> length line < 10) . lines
```

- ``interact`` takes a function of type ``String -> String`` as a parameter and gives back an I/O action
  - the I/O action performs the same execution as the previous version of our ``shortLinesOnly`` program
  - i.e.: it processes one line at a time of streamed input
- \* we can either redirect, or type lines of input ourselves

# Assignment Project Exam Help

11

<https://powcoder.com>

## Add WeChat powcoder

## \*Palindromes (1)

We can practice ``interact`` more by writing ``palindromes.hs`` to test whether a line of input is a palindrome.

- a **palindrome** is a string with characters in forward sequence that also have the same sequence backward
- let us write a function to pass to ``interact`` so that it replaces palindromes with
  - "palindrome", or
  - "not palindrome"

12

\*Palindromes  
(2)

```
{- ### palindromes.hs ### -}
main = interact respondPalindromes

respondPalindromes :: String -> String
respondPalindromes =
  unlines
    . map (\line ->
      if isPal line
      then "palindrome"
      else "not palindrome"
    )
    . lines

isPal :: String -> Bool
isPal xs = xs == (reverse xs)
```

Assignment Project Exam Help

13

<https://powcoder.com>

Add WeChat powcoder

\*Palindromes  
(3)

Try out palindromes.hs with some input:

```
{- ### data\words.txt ### -}
dogaroo
radar
rotor
madam
```

14

— Reading and Writing Files —

## Assignment Project Exam Help

15

<https://powcoder.com>

## Add WeChat powcoder

`openFile`  
and  
`hGetContents`

Save the following in a file called ``data/girlfriend.txt`` (from Avril Lavigne's hit song "Girlfriend"):

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

First, read file contents and print it to standard output:

```
{- ### girlfriend.hs v1 ### -}
import System.IO

main = do
  handle <- openFile "data/girlFriend.txt" ReadMode
  contents <- hGetContents handle
  putStr contents
  hClose handle
```

16



## openFile (1)

- `openFile :: FilePath -> IOMode -> IO Handle`
  - parameters take:
    - (1) a **specified file**,
    - (2) an `IOMode` (`'ReadMode'`), and
    - (3) an **I/O action** to open a file and **yield** the file's **associated handle**
- `'FilePath'` is just a type synonym for `'String'`,  
i.e.: `'type FilePath = String'`
- `'IOMode'` is defined like so:
  - `'data IOMode =`  
`ReadMode | WriteMode | AppendMode | ReadWriteMode'`
  - (note that it is not `'IO Mode'` with a space, which would be an I/O action with yield of `'Mode'`)
  - `'IOMode'` is just an enumeration

## Assignment Project Exam Help

17

<https://powcoder.com>

## Add WeChat powcoder

## openFile (2)

- programs in general (not just Haskell) are allowed by the OS to work with a file by its file descriptor
  - the file descriptor in Haskell is typed as a `'Handle'` that implements a bit more functionality with the mode
  - `'getContents'` works with standard input
  - `'hGetContents'` works with specified files
  - these functions are altogether otherwise the same
- `'hGetContents'` will only load parts of the file as needed, which helps when we are loading a large file to avoid hogging RAM
  - `'handle'` points to current location in the file to keep track of reading it
- `'putStr'` is familiar
- `'hClose'` takes a handle and returns an I/O action that closes the file
  - always close a resource obtained from outside of your program
  - otherwise, the program could terminate trying to open a file with a handle that has not been closed

18

## withFile (1)

Function `withFile` compacts the execution of our previous program, and it has the following signature:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

- `FilePath` takes a specified file
- `(Handle -> IO a)` function
- lastly, `withFile` returns an I/O action
  - it will open the file
  - perform some action with the function passed in
  - close the file
  - if anything goes wrong, `withFile` makes sure to close the file

# Assignment Project Exam Help

19

<https://powcoder.com>

Add WeChat powcoder

## withFile (2)

```
{- ### girlfriend.hs v2 ### -}
import System.IO

main = do
    withFile "data/girlfriend.txt" ReadMode
        (\handle -> do
            contents <- hGetContents handle
            putStr contents
        )
```

- `(\handle -> do ...)` function ( :: handle -> IO a )
  - it is typical to pass it in as a lambda this way
  - `withFile` makes sure to close the file handle if anything goes wrong

20

## Exceptions

Errors during execution that happen (e.g.: calling `'head'` on an empty list) typically print an error message:

- these are called **exceptions**
- `'withFile'` ensured a file handle is closed when an exception is raised
- for any resource (such as a file), we need to ensure it is released given any situation or error

Then the common design for working with a resource and dealing with input and output using it is as follows:

- get the resource as an I/O action
- ensure it can be closed regardless of exception
- process the data of the resource as an I/O action
- (altogether, these describe an I/O action of the resource type)

# Assignment Project Exam Help

21

<https://powcoder.com>

Add WeChat powcoder

— The Bracket Function —

22

## The `bracket` Function

Then the following type for the `bracket` function...

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO a
```

...matches its parameters (and return type) for the common design described.

- the first `IO a` is the resource obtained, but as an I/O action
- `(a -> IO b)` is a function to execute regardless of exception
- `(a -> IO c)` is a function to process the resource
- the last `IO a` needs to match context of input I/O action `IO a`

# Assignment Project Exam Help

23

<https://powcoder.com>

Add WeChat powcoder

## Using the `bracket` Function

Then implementing a function such as `withFile` becomes very concise:

```
{- example of exception handling -}
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' name mode f =
    bracket (openFile name mode)
      (\handle -> hClose handle)
      (\handle -> f handle)
```

24

— Working with Handles —

Assignment Project Exam Help

25

<https://powcoder.com>

Add WeChat powcoder

Functions  
for  
Handles

There are counterpart functions for the same actions with handles that we have been doing with stdin and stdout:

- ``hGetLine``
- ``hPutStr``
- ``hPutStrLn``
- ``hGetChar``

26

## Files as Large Strings

Using files as if they were large strings is very common, so we have three more functions to help with this:

- ``readFile :: FilePath -> IO String`` take a specified file and performs I/O read action (lazily) and can bind to some variable
  - a bit more concise than calling ``openFile`` and then ``hGetContents``
- ``writeFile :: FilePath -> String -> IO ()`` takes a specified file, a string, and performs I/O write action
  - if the file already exists, its contents are first erased before writing
- ``appendFile :: FilePath -> String -> IO ()``
  - same as ``writeFile``, but does not erase contents first, and instead appends the given string after the contents of the file

# Assignment Project Exam Help

27

<https://powcoder.com>

Add WeChat powcoder

## File Reading and Writing (Simranjit Singh)

```
import System.IO
import Data.Char

main :: IO ()
main = do
  line <- getLine
  let num = read line :: Int
  contents <- readFile "msg.txt"
  writeFile "secretmsg.txt" (encode num contents)

-- enCode
encode :: Int -> String -> String
encode offset msg = map (\c -> chr $ ord c + offset) msg
```

28

## appendFile (Hunter Klassen)

```
import System.IO

main = do
  student <- getLine
  appendFile "classList.txt" (student ++ ", ")

File classList.txt:
Klassen,
```

# Assignment Project Exam Help

29

<https://powcoder.com>

Add WeChat powcoder

## Directories and Files (Amy Campbell)

```
import System.IO
import System.Directory
import Data.List
import System.Process

main = do
  createDirectoryIfMissing False "planner"
  contents <- getDirectoryContents "planner"
  let cats = unlines contents
  putStrLn "Choose Subject:"
  putStrLn cats
  catName <- getLine
  putStrLn "Enter assignment name:"
  newAssign <- getLine
  appendFile ("planner/" ++ catName ++ ".txt") (newAssign ++ "\n")
```

30

## Directories and Files Functions

(Amy Campbell)

- `getDirectoryContents` will show ``.`` and ``.``
- `listDirectory` is the same, but does not show ``.`` and ``.``
- (we have seen `appendFile` in a previous example)

Assignment Project Exam Help

31

<https://powcoder.com>

Add WeChat powcoder

— To-Do List —

32



## To-Do List (1)

We have enough to write an application to keep a list of things we would like to do and store them in a file for us.

Creating files involves POSIX libraries, so for now, create the `data/todo.txt` file, and write the following program:

```
{- ### todo.hs v1 ### -}
import System.IO

main = do
  todoItem <- getLine
  appendFile "data/todo.txt" (todoItem ++ "\n")
```

# Assignment Project Exam Help

33

<https://powcoder.com>

## Add WeChat powcoder

## To-Do List (2)

Type in some chores, but as it is, you have to run the program for each to do item you want to add:

```
{- ### data\todo.txt ### -}
iron the dishes
dust the dog
take the salad out of the oven
```

- you can observe the contents of the to do list file with
  - `cat data/todo.txt` in Bash,
  - and with `type data/todo.txt` in Windows CMD

34

## Deleting To-Do Items (1)

```
{- ### deletetodo.hs v1 ### -}
import System.IO
import System.Directory
import Data.List

main = do
  contents <- readFile "data/todo.txt"
  let
    todoTasks = lines contents
    numberedTasks =
      zipWith (\n line -> show n ++ " - " ++ line)
        [0..] todoTasks
  putStrLn "These are your TO-DO items:"
  mapM_ putStrLn numberedTasks
  putStrLn "Which one do you want to delete?"
  numberString <- getLine
  (cont'd on next slide...)
```

Assignment Project Exam Help

35

<https://powcoder.com>

Add WeChat powcoder

## Deleting To-Do Items (2)

```
(cont'd...)

let
  number = read numberString
  newTodoItems =
    unlines $ delete (todoTasks !! number) todoTasks
(tempName, tempHandle) <- openTempFile "." "temp"
hPutStr tempHandle newTodoItems
hClose tempHandle
removeFile "data/todo.txt"
renameFile tempName "data/todo.txt"
```

36

## Deleting To-Do Items (3)

- `todoTasks` stores a list of lines from the `todo.txt` file
  - it looks something like
 

```
`["iron...", "dust...", "take the salad..."]`
```
- `zipWith` prefixes a number to each line in the `todoTasks` list
  - `numberedTasks` looks something like
 

```
`["0 - iron...", "1 - dust...", "2 - take the salad..."]`
```
- `mapM_` prints out the `numberedTasks`, each element on a separate line
- `number` gets assigned to a choice from the user of which numbered line to delete

## Assignment Project Exam Help

37

<https://powcoder.com>

## Add WeChat powcoder

## Deleting To-Do Items (4)

- `!!` accesses the element we want, and `delete` removes the first occurrence of it from the list
  - `unlines` combines the edited list back into one string (with end-of-line characters between elements)
- the function `openTempFile` is a round-about way to later replace the old todo list file
  - `openTempFile` takes two parameters:
    - a temporary directory, which we give as the current directory `"."`
    - the prefix to a temporary file name, which we call `"temp"`, and a few random digits are added
    - the new temporary file name and handle are returned in a pair
- we use the temporary file to write the new edited list and replace the old file

38

## More Exceptions

There are some issues with the ``deletetodo.hs``:

- if the program terminates from some kind of error
  - the temporary file is created,
  - but not renamed
- we need functionality similar to ``bracket`` where
  - a resource gets cleaned up automatically when we are done
  - but only if there is some kind of exception

# Assignment Project Exam Help

39

<https://powcoder.com>

Add WeChat powcoder

## \*Deleting To-Do Items: Version 2

We use ``bracketOnError`` from the ``Control.Exception`` module.

```
{- ### deletetodo.hs v2 ### -}
import System.IO
import System.Directory
import Data.List
import Control.Exception

main = do
  contents <- readFile "data/todo.txt"
  let
    todoTasks = lines contents
    numberedTasks =
      zipWith (\n line -> show n ++ " - " ++ line)
        [0..] todoTasks
  putStrLn "These are your TO-DO items:"
  mapM_ putStrLn numberedTasks
  putStrLn "Which one do you want to delete?"
  numberString <- getLine
```

40

## \*Deleting To-Do Items: Version 2

The parameters of  
'bracketOnError':

- the resource first
- what to do on an exception
- what to do as expected normally

```
let
  number = read numberString
  newTodoItems =
    unlines $ delete (todoTasks !! number) todoTasks
  bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
      hClose tempHandle
      removeFile tempName
    )
    (\(tempName, tempHandle) -> do
      hPutStr tempHandle newTodoItems
      hClose tempHandle
      removeFile "data/todo.txt"
      renameFile tempName "data/todo.txt"
    )
```

# Assignment Project Exam Help

41

<https://powcoder.com>

Add WeChat powcoder

— Command-Line Arguments —

42

## Command-Line Arguments

The To-Do App limitations only work with the one text file, and only add or delete one item from a list one at a time.

Instead, we can specify what we would like our program to do exactly when we execute it. We use `System.Environment`:

- it lets us read command-line arguments:

```
{- ### argTest.hs ### -}
import System.Environment
import Data.List

main = do
  args <- getArgs
  progName <- getProgName
  putStrLn "The arguments are:"
  mapM putStrLn args
  putStrLn "The program name is:"
  putStrLn progName
```

# Assignment Project Exam Help

43

<https://powcoder.com>

## Add WeChat powcoder

## Combine To-Do App Functions

We combine the smaller example programs to insert and delete items for our to-do list.

Altogether, we want the user to choose command-line options for controlling its functionality:

- view tasks
- add tasks
- delete tasks

To add a task to the `todo.txt` file, we will want to enter into the terminal, and similarly for the other two options:

```
(this is the desired behaviour from terminal execution)
./todo add todo.txt "Find the magic sword of power"
./todo view todo.txt
./todo remove todo.txt 2
```

44

User  
Enters  
Command

The program start will decide between which function to run which matches the first command argument input:

```
{- ### todo.hs part 1 ### -}
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove

main = do
    (command:argList) <- getArgs
    dispatch command argList
```

Assignment Project Exam Help

45

<https://powcoder.com>

Add WeChat powcoder

Example  
User  
Execution

- in `main` we first get the arguments matched with pattern `(command:argList)`
- then `command` should pattern match with one of the defined versions of the `dispatch` function
  - the result returns the function to apply to the `argList`

So, suppose we type in the terminal:

```
./todo add todo.txt "Find the magic sword of power"
```

Then `main` should parse `add` as the command and `["todo.txt", "Find the magic sword of power"]`

46

## add

Next, we implement the three functions for our app, starting with `add`:

```
{- ### todo.hs part 2, add v1 ### -}
```

```
add :: [String] -> IO ()
```

```
add (filename:todoItems) =  
    appendFile filename (unwords (todoItems ++ ["\n"]))
```

- (no errors dealt with for mismatched file names until later)
- the rest of the list is dealt with as one long task to be appended on a new line at the end of the user-specified file
  - `unwords` will concatenate all string elements in the list together as one string separated by a space

# Assignment Project Exam Help

47

<https://powcoder.com>

Add WeChat powcoder

## Foldable and Exception Modules

Before we move on to combine with the `view` and `remove` functions, add the following imports:

```
{- ### todo.hs part 3 ### -}
```

```
import Data.Foldable
```

```
import Control.Exception
```

48



## View To-Do List

Foldable module is for `mapM\_` and Exception module is for use of `bracketOnError` to handle potential file issues.

```
{- ### todo.hs part 4 ### -}
view :: [String] -> IO ()
view (filename:remainder) = do
  contents <- readFile filename
  let
    todoTasks = lines contents
    numberedTasks =
      zipWith (\n line -> show n ++ " - " ++ line)
        [0..] todoTasks
  putStrLn "These are your To-Do items:"
  mapM_ putStrLn numberedTasks
```

# Assignment Project Exam Help

49

<https://powcoder.com>

## Add WeChat powcoder

## Remove from To-Do List (1)

And basically the same function as before with `remove`:

```
{- ### todo.hs part 4 ### -}
remove :: [String] -> IO ()
remove [filename, numString] = do
  contents <- readFile filename
  let
    todoTasks = lines contents
    numberedTasks =
      zipWith (\n line -> show n ++ " - " ++ line)
        [0..] todoTasks
  putStrLn "These are your To-Do items:"
  mapM_ putStrLn numberedTasks
```

50

## Remove from To-Do List (2)

```
let
  number = read numString
  newTodoItems =
    unlines $ delete (todoTasks !! number) todoTasks
  bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
      hClose tempHandle
      removeFile tempName
    )
    (\(tempName, tempHandle) -> do
      hPutStr tempHandle newTodoItems
      hClose tempHandle
      removeFile "data/todo.txt"
      renameFile tempName "data/todo.txt"
    )
```

# Assignment Project Exam Help

51

<https://powcoder.com>

## Add WeChat powcoder

## Example To-Do List

You could try out the application with something similar to  
`view` output that might look like the following:

```
{- ### example output you might design ### -}
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from dry cleaners
```

The `dispatch` function we defined is easily extensible.

- add your own functions with another pattern match
- try adding a `bump` function that takes a number argument and moves that numbered task to the top of the to-do list

52

— Handling Bad Input —

## Assignment Project Exam Help

53

<https://powcoder.com>

## Add WeChat powcoder

doesn'tExist

We handle any other commands with a catchall pattern at the end of the `dispatch` function:

```
{- ### todo.hs part 5: edit dispatch to use as last pattern ### -}
doesn'tExist :: String -> [String] -> IO ()
doesn'tExist command _ =
    putStrLn $ "The " ++ command ++ " does not exist."
```

We may have wanted to be strict about the arguments for the `add` function and implemented it like so:

```
{- ### todo.hs part 6, add v2 ### -}
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
add _ = putStrLn "The add command takes exactly two arguments"
```

54

— Randomness —

Assignment Project Exam Help

55

<https://powcoder.com>

Add WeChat powcoder

## The random Function

We want to generate random numbers for our programs.

Before we get started, you will need to install the `random` package with `stack` (see the `notes.md` file).

- Haskell is a purely functional language
  - so, it has referential transparency
  - (a function always gives back same output when given the same input)
- this seems to go against what we want with randomness

We will explore what is available in the `System.Random` module and its `random` function:

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

56

## Random Generators

- `RandomGen` type class is for types as sources of randomness
- `Random` type class is for type values resulting from random source

Then interpreting the type for the `random` function...

- input is a random generator (or source of randomness)
- output is a pair: random value and another random generator

The pair facilitates a sequence of random values, controlled by generators.

# Assignment Project Exam Help

57

<https://powcoder.com>

## Add WeChat powcoder

## Generating a Random Number

First, we want a `RandomGen` instance named `StdGen`.

- we can get one with the `mkStdGen :: Int -> StdGen` function
- use the `StdGen` returned as input to the `random` function

```
random (mkStdGen 100)
```

The result of the above:

```
`(9216477508314497915,  
  StdGen  
  { unStdGen =  
    SGen 712633246999323047 2532601429470541125  
  })`
```

58

## Specific Types of Random Value

We sometimes need to make sure we have the random number of a concrete type; specify with type annotation:

```
random (mkStdGen 100) :: (Int, StdGen)
```

- notice that it gives the exact same random number in the first element of the pair
- we can change the first element depending on which random generator is passed in

```
random (mkStdGen 101) :: (Int, StdGen)
(5022189999818777644, StdGen
 {unStdGen = SMGen 1869071494976701883 15060681878671775511})
```

# Assignment Project Exam Help

59

<https://powcoder.com>

Add WeChat powcoder

## Other Random Value Types

Type annotation for a random float:

```
random (mkStdGen 101) :: (Float, StdGen)
(0.7901666, StdGen {unStdGen = SMGen 1869071494976701883
15060681878671775511})
```

or even random `Bool`:

```
random (mkStdGen 101) :: (Bool, StdGen)
(False, StdGen {unStdGen = SMGen 1869071494976701883
15060681878671775511})
```

60

— Toss a Coin —

## Assignment Project Exam Help

61

<https://powcoder.com>

## Add WeChat powcoder

### Toss Three Coins

The result of tossing a coin three times can be simulated.

- we use three random values of `Bool` type (T: "tails," F: "heads")
- chain together the random generators within a function we write
- without `random` function returning a random generator, we would instead have to supply three of our own generators

{- v1: they will need to install the "random" package -}

```
import System.Random
```

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
```

```
threeCoins gen =
```

```
  let
```

```
    (firstCoin, newGen) = (random gen) :: (Bool, StdGen)
```

```
    (secondCoin, newGen') = (random newGen) :: (Bool, StdGen)
```

```
    (thirdCoin, newGen'') = (random newGen') :: (Bool, StdGen)
```

```
  in (firstCoin, secondCoin, thirdCoin)
```

62

### \*Probabilities

Could you tally and calculate sample probabilities for the results?

- how often is the first coin `False` vs `True` after 10 executions?
- what about the other coins?
- were any triples repeated?
- with many executions, does the result get closer to 50% probability for each coin?
- is the random generator a fair one?
- there are quite a few things we could analyze mathematically

## Assignment Project Exam Help

63

<https://powcoder.com>

Add WeChat powcoder

— Sequences of Random Numbers —

64



## \*randoms

We want to avoid manually passing in generators to the `random` function to work with some sequence.

- thankfully, there is a `randoms` function
  - the input is a random generator
  - the output is a sequence of random numbers

For example:

```
take 5 $ randomness (mkStdGen 100) :: [Int]
```

- notice that a random generator is not returned.
  - this is due to the recursive design of `randoms`
  - we will implement our own version, so you can see

# Assignment Project Exam Help

65

<https://powcoder.com>

Add WeChat powcoder

## \*Integer Random Sequence

```
randoms' :: StdGen -> [Int]
randoms' gen =
  let (value, newGen) = random gen :: (Int, StdGen)
  in value:(randoms' newGen)
```

The above is not as general as the `randoms` function:

- it has type constraints `(RandomGen g, Random a) => g -> [a]`
- we need more concepts detailed later before we can write a more general version

66

## \*Limiting Recursion (1)

If we want to get back a generator, we need to customize a function to return a finite number of random values:

```
finiteRandoms :: (Num n, Eq n) => n -> StdGen -> ([Int], StdGen)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
  let
    (value, newGen) = (random gen) :: (Int, StdGen)
    (restOfList, finalGen) = finiteRandoms (n-1) newGen
  in (value:restOfList, finalGen)
```

# Assignment Project Exam Help

67

<https://powcoder.com>

## Add WeChat powcoder

## \*Limiting Recursion (2)

- notice in the recursive function above that the recursion takes place within a `let` defined variable
- since `newGen` cannot infer its type from any other information, it must be provided on the right-hand side value assigned
- again, this version is restricted to `Int` random values and the `StdGen` type
- we will get to more generalized code later

68

## \*randomR

We may also want to control the range of possible values we would like as a result for a random number.

- this is provided by function `randomR`

```
randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)
```
- it works pretty much the same as `random`, but it takes a pair of elements describing a range of values
  - e.g.: `randomR (1,6) (mkStdGen 359353)`
  - the above would give a value between `1` and `6`, inclusive

```
take 10 $ randomRs ('a', 'z') (mkStdGen 100)
```

# Assignment Project Exam Help

69

<https://powcoder.com>

Add WeChat powcoder

— Randomness and I/O —

70

## getStdGen

Randomness is not helpful if we execute a program on the command line and it always gives the same random values.

- so far, this would be all we could program
- however, there is a `getStdGen` I/O action that will set up a different *global* generator for us when it is bound during an I/O action
- we can then use the global generator as we have with generators like the `StdGen` type we have already used

```
{- ### randomString.hs ### -}
import System.Random

main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a', 'z') gen)
```

## Assignment Project Exam Help

71

<https://powcoder.com>

## Add WeChat powcoder

## Different Result Each Execution

The `getStdGen` function will not give a different global generator if used within the same program a second time.

- instead use `newStdGen` and bind to another variable

```
{- ### randomString2.hs ### -}
import System.Random

main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a', 'z') gen)
  gen2 <- newStdGen
  putStrLn $ take 20 (randomRs ('a', 'z') gen2)
```

72

## \*Guessing Game (1)

- try swapping in `getStdGen` for `newStdGen` to see repeated string
- however, if you use `getStdGen` to bind again after `newStdGen` binds, we will get yet a third different global generator

We can write a more substantial program to play a game with the user to guess a randomly generated number:

```
{- ### guessGame.hs ### -}
import System.Random
import Control.Monad(when)

main = do
  gen <- getStdGen
  askForNumber gen
```

# Assignment Project Exam Help

73

<https://powcoder.com>

Add WeChat powcoder

## \*Guessing Game (2)

```
askForNumber :: StdGen -> IO ()
askForNumber gen = do
  let
    (randNumber, newGen) = randomR (1, 10) gen :: (Int, StdGen)
  putStrLn "Which number in the range from 1 to 10 am I thinking of? "
  numberString <- getLine
  when (not $ null numberString) $ do
    let number = read numberString
    if randNumber == number
      then putStrLn "You are correct!"
      else putStrLn $ "Sorry, it was " ++ show randNumber
  askForNumber newGen
```

74

## \*Exit Guessing Game

- you can exit the program by not typing a guess and press ``ENTER``
  - (the ``when`` condition is ``False``)
- ``when`` is used for the situation that an ``else`` clause is not wanted
- look up the function ``reads`` and use it to handle bad user input, such as "haha"
  - it returns an empty list upon failure to read a string
  - else, the return fits the pattern ``[(input string, remainder of input)]``

Note that the guessing game program is recursive.

- each time the user guesses, ``askForNumber`` is called another recursive level

# Assignment Project Exam Help

75

<https://powcoder.com>

Add WeChat powcoder

— Bytestrings —

76

## Thunks

We can treat files, lists, and even infinite lists as streams to access them only when a program absolutely needs to.

- `[1,2,3,4]` is just presentation for `1:2:3:4:[]`
- i.e. the right expression is a *promise* of a list (it is not evaluated to one yet)
- such expressions that are deferred computations are called **thunks**
- this means we can work with collections of infinite elements
- obviously, this behaviour is not likely to be efficient
  - most of the time the extra work by a computer does not get in our way
  - it does become a problem for big files, including their manipulation

# Assignment Project Exam Help

77

<https://powcoder.com>

## Add WeChat powcoder

## Bytestrings

There is a special type for mitigating the laziness of Haskell when working with large files.

- such are called **bytestrings**
- each element is exactly `8` bits
- they handle laziness differently

78

— Strict and Lazy ByteString —

Assignment Project Exam Help

79

<https://powcoder.com>

Add WeChat powcoder

## Levels of Laziness

There are two kinds of bytestrings:

- Strict (no laziness)
- Lazy (but not as lazy as lists)

The two kinds are kept in separate modules, respectively:

- `Data.ByteString``
- `Data.ByteString.Lazy``

80



## Chunk or Thunk

For strict bytestrings:

- we cannot have an infinite number of byte elements
- evaluating one element means evaluating all others
  - i.e.: there is *no* laziness, and no thunks involved at all

For lazy bytestrings:

- each element is a **thunk**, so they are slow for some uses
- elements are stored in **chunks (not thunks)** of 64 KB each
  - evaluating a byte in a chunk means the whole chunk will be evaluated
  - processing a file with a lazy bytestring will be done chunk by chunk
  - this way, memory usage will not suffer
  - 64 KB likely fits within the CPU L2 cache

# Assignment Project Exam Help

81

<https://powcoder.com>

Add WeChat powcoder

## Lazy versus Strict Bytestrings

There are many common functions found in both `Data.List` and `Data.ByteString.Lazy`:

- the types in signatures correspond to `ByteString` instead of `[a]` and `Word8` instead of `a`

We will do qualified imports because the naming is the same between the two modules:

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

82

## Data within Bytestrings

Then ``B`` refers to the lazy bytestring module and ``S`` the strict module

- (we mostly will show use of the lazy version of functions)

The first function ``pack` :: [Word8] -> ByteString`` converts:

- a list (very lazy) to a less lazy form (only at 64 KB chunks)
- ``Word8`` is similar to ``Int``, but restricted to values from ``0`` to ``255``
  - also categorized within the ``Num`` type class
  - e.g.: literal ``5`` is polymorphic, and can act like a ``Word8`` type value

# Assignment Project Exam Help

83

<https://powcoder.com>

Add WeChat powcoder

## Simple Examples of Lazy Bytestrings

```
B.pack [99, 97, 110]
```

- the result should be "can"
- the result fits inside one chunk with 65533 more bytes
- no need to specify ``Word8`` type for elements, because this is inferred
- values larger than 255 will cause an error message

```
B.pack [97..122]
```

- the above gives the Latin lowercase alphabet characters

84

## Conversion of Data

If we want access to the elements of the lazy bytestring, we must first unpack it into a list of `Word8` elements:

```
let by = B.pack [98,111,114,116]
B.unpack by
```

There are functions such as `fromChunks` to convert a list of strict bytestrings into a one lazy bytestring:

```
B.fromChunks [S.pack [40..42], S.pack [43..45], S.pack [46..48]]
```

- the result looks like a string
- `fromChunks` converts a list of strict bytestrings to a lazy bytestring
- the inverse operation `toChunks` can convert a lazy bytestring into a list of strict ones

# Assignment Project Exam Help

85

<https://powcoder.com>

## Add WeChat powcoder

## Prefix Like a List

The bytestring prefix operation is called `cons` instead of the special symbol `:` used for lists.

- inputs a byte and bytestring
- outputs a bytestring with the byte prefixed at front

```
B.cons 85 $ B.pack [80..82,84]
```

86

## Other Modules and Functions

Conversion between lazy bytestrings, strings (lists of chars), and strict bytestings has other modules you may want:

- the data type for `Word8` is found in `Data.Word` (but not that many functions)
- `Data.Text` has `chunksOf` function to split text into smaller pieces (use with `Data.Text.pack` to convert strings to `Text` type)
- `Data.ByteString.Builder` has functions to help with efficient processing of text data (example in `builder.hs`)

# Assignment Project Exam Help

87

<https://powcoder.com>

## Add WeChat powcoder

## Files and ByteString

More of analogous functions exist between ``System.IO``, but ``String`` type are replaced with ``ByteString``:

```
readFile :: FilePath -> IO String
```

would instead be:

```
readFile :: FilePath -> IO ByteString
```

88

— Copying Files with Bytestrings —

## Assignment Project Exam Help

89

<https://powcoder.com>

## Add WeChat powcoder

Example of  
Reading a File  
(1)

Strict bytestrings would mean a file is read into memory all at once. We will instead see how to read in a file one chunk at a time, as needed:

```
{- ### bytestringcopy.hs ### -}
import System.Environment
import System.Directory
import System.IO
import Control.Exception
import qualified Data.ByteString.Lazy as B

main = do
  (filename1:filename2:_) <- getArgs
  copy filename1 filename2
```

90

## Example of Reading a File (2)

```
copy source dest = do
  contents <- B.readFile source
  bracketOnError
    (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
      hClose tempHandle
      removeFile tempName
    )
    (\(tempName, tempHandle) -> do
      B.hPutStr tempHandle contents
      hClose tempHandle
      renameFile tempName dest
    )
```

# Assignment Project Exam Help

91

<https://powcoder.com>

## Add WeChat powcoder

## Notes on Processing Files

- the exception handler makes sure
  - if anything goes wrong, there is no damage done,
  - because we are only working with a temporary file
  - until the work of the program is done with the source file
  - in the event of an error, the temporary file is just deleted
- we must always close resources, such as a file handle
- only the read and write functions needed to use the bytestream versions, and not opening or closing the file

(System.Directory: renameFile, removeFile)

(System.IO: openTempFile, hClose)

(Control.Exception: bracketOnError)

92

## When Processing Files

Ultimately, for a program to switch from normal file processing to a bytestring version would be to:

- make the necessary imports
- prefix the read and writes with corresponding module for desired bytestring type

Consider switching for better performance on larger files if you are noticing slowdown.

NLP will likely require processing of very large text corpus.

# Assignment Project Exam Help

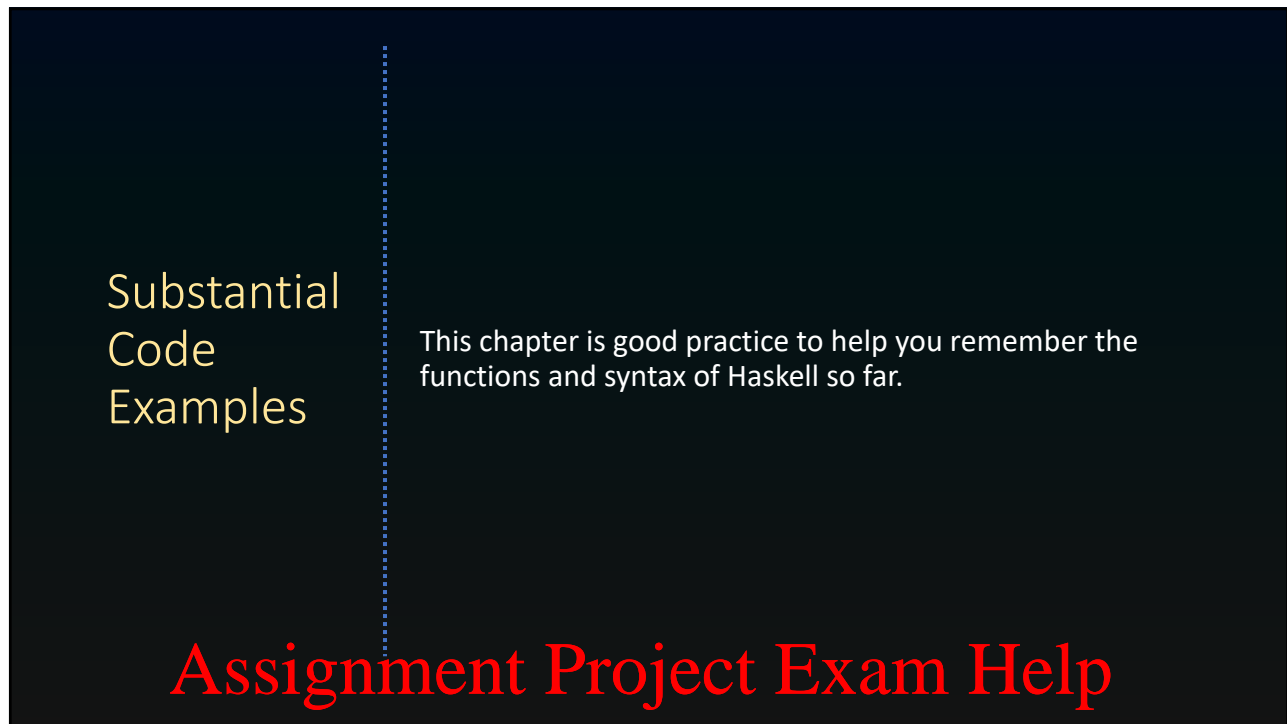
93

<https://powcoder.com>

Add WeChat powcoder

— Chapter 10 —

94



Substantial  
Code  
Examples

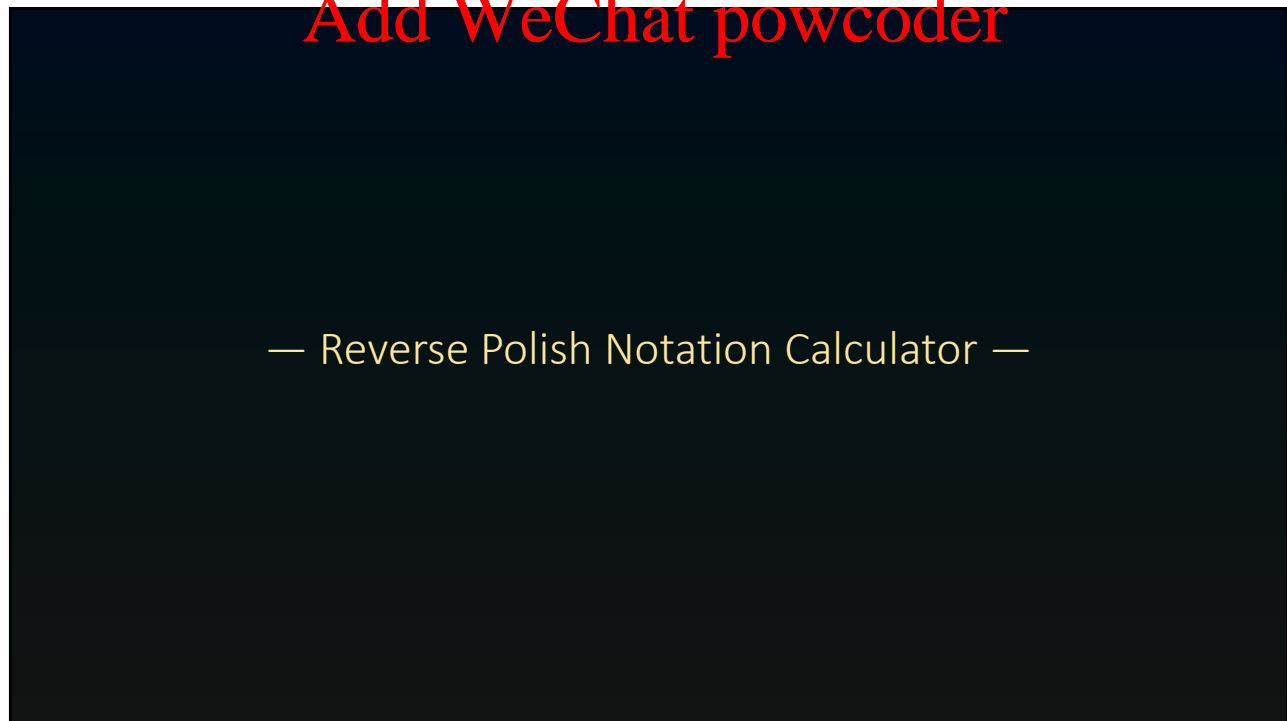
This chapter is good practice to help you remember the functions and syntax of Haskell so far.

Assignment Project Exam Help

95

<https://powcoder.com>

Add WeChat powcoder



— Reverse Polish Notation Calculator —

96



## Infix Notation

Infix notation for writing the common algebraic operations as functions:

- e.g.: ``3 + (6 * 2)``

We have seen how to write them using infix with backticks:

- for example: ``elem``

Assignment Project Exam Help

97

<https://powcoder.com>

Add WeChat powcoder

## Reverse Polish Notation

Reverse polish notation (RPN) operators go after operands:

- e.g.: ``4 3 +``
- consider multiplying the result of the above by ``10``
- e.g.: ``4 3 + 10 *``

Perform calculations by:

- placing each number in an expression onto a stack
- if an operator is encountered, pop two numbers off the stack
  - make sure operator is applied in the order numbers were pushed
- perform calculation and push the result back on the stack
- continue, until the entire expression is read
- there should only be one number left on the stack
  - (this assumes the expression is well-formed)

98

- let's step through to the result of the expression

``10 4 3 + 2 * -``

- the result should be ``-4``

Assignment Project Exam Help

99

<https://powcoder.com>

Add WeChat powcoder

— Writing an RPN Function —

100

## Stack Processing

We want to parse a string and return a numeric result, so a function that evaluates an RPN expression is described as:

```
solveRPN :: String -> Double
```

- should have a space between values and operators
- parse so each token in expression will be an element of a list
  - e.g.: ``["10", "4", "3", "+", "2", "*", "-"]``
- process expression element by element, left to right, with ``foldl``
  - use accumulator as our stack, so result of the ``foldl`` should be a stack
- how do we want to represent a stack?
  - if we use a list, its head could be the top of a stack
  - the front of a list is much faster to add an element than at the back

## Assignment Project Exam Help

101

<https://powcoder.com>

## Add WeChat powcoder

## RPN Calculator Design

```
solveRPN :: String -> Double
solveRPN expression =
  head (foldl foldingFunction [] (words expression))
  where foldingFunction stack item = ...
```

- ``words` expression` will split input string into list of elements
- the ``[]`` is our empty stack/accumulator
- ``head`` will remove the final element result out of our stack

Let's clean things up a bit with point-free style:

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
  where foldingFunction stack item = ...
```

102

## RPN Calculator Code

Now let's implement the `foldingFunction` that:

- is applied to each element of a list via `foldl`
- so we must work with accumulator as its first parameter (our stack)
- an element from the top of the stack as its second parameter (either an operation or a number)
  - remember, this element will be the top of the stack

```
solveRPN :: String -> Double
solveRPN =
  head . foldl foldingFunction [] . words
  where
    foldingFunction (x:y:ys) "+" = (y + x):ys
    foldingFunction (x:y:ys) "*" = (y * x):ys
    foldingFunction (x:y:ys) "-" = (y - x):ys
    foldingFunction (x:y:ys) "/" = (y / x):ys
    foldingFunction xs numberString = read numberString:xs
```

# Assignment Project Exam Help

103

<https://powcoder.com>

Add WeChat powcoder

## Example of Using RPN

Go ahead and try out the RPN calculator:

```
solveRPN "10 4 3 + 2 * -"
-4.0
solveRPN "2 3.5 +"
5.5
solveRPN "90 34 12 33 55 66 + * - +"
-3947.0
solveRPN "90 34 12 33 55 66 + * - + -"
4037.0
solveRPN "90 3.8 -"
86.2
```

104

— Adding More Operations —

## Assignment Project Exam Help

105

<https://powcoder.com>

## Add WeChat powcoder

### More Operators

We can easily add more functionality to the RPN calculator:

```
solveRPN :: String -> Double
solveRPN =
  head . foldl foldingFunction [] . words
  where
    foldingFunction (x:y:ys) "+" = (y + x):ys
    foldingFunction (x:y:ys) "*" = (y * x):ys
    foldingFunction (x:y:ys) "-" = (y - x):ys
    foldingFunction (x:y:ys) "/" = (y / x):ys
    foldingFunction (x:y:ys) "^" = (y ** x):ys
    foldingFunction (x:xs) "ln" = (log x):xs
    foldingFunction xs "sum" = [sum xs]
    foldingFunction xs numberString = read numberString:xs
```

106

## Example with More Operations

- the operators do not even need to be binary
- even `sum` can just pop all numbers off the stack and push back the single result

```
solveRPN "2.7 ln"
0.9932517730102834
solveRPN "10 10 10 10 sum 4 /"
10.0
solveRPN "10 10 10 10 10 sum 4 /"
12.5
solveRPN "10 2 ^"
100.0
```

There is no error handling with this calculator, but as it is, having such concise code with Haskell is evident in this application.

# Assignment Project Exam Help

107

<https://powcoder.com>

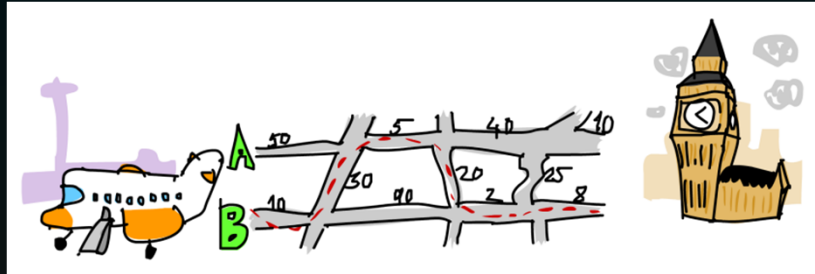
Add WeChat powcoder

— Heathrow to London —

108

## Solving Problems with Data Types

Observe the road system with times (in minutes) for travelling each road toward the famous "Big Ben" clock tower in London:



## Assignment Project Exam Help

109

<https://powcoder.com>

## Add WeChat powcoder

## Data Types

The optimal sequence of roads is to start at "B" and take those roads labelled with times in the order:

- `10`, `30`, `5`, `20`, `2`, `8` (for a total of `75` minutes)
- we are always choosing the smallest time at each intersection
- but this greedy approach may not work

Store the road times in a Section, as pattern of threes:

- road along A,
- road along B, and
- road connecting A and B (which we just describe as C)

Yet, matching with C for "Big Ben" value `0` at the end of the sequence has no connection between A and B roads.

110

## Section and RoadSystem

Design a data structure describing three values at a time:

```
data Section = Section {
  getA :: Int
  , getB :: Int
  , getC :: Int
} deriving (Show)

type RoadSystem = [Section]
```

- `Section` record stores three `Int` values for road sections A, B, and C
- `RoadSystem` list has elements of type `Section`

# Assignment Project Exam Help

111

<https://powcoder.com>

## Add WeChat powcoder

## Road Data, Label, and Path Type

Now we can represent our road times with the following:

```
roadsToTower :: RoadSystem
roadsToTower =
  [ (Section 50 10 30)
  , (Section 5 90 20)
  , (Section 40 2 25)
  , (Section 10 8 0)
  ]
```

An enum to help label which values go with which roads:

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

112



## Toward a Solution

A final function to describe a solution:

```
optimalPath :: RoadSystem -> Path
```

A solution with `roadsToTower` as input should evaluate to:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

Each step we want optimal path to either intersection along A or B, in parallel, to then move on to the next step.

- we build up the best route, intersection by intersection, in parallel
- as a function, this would look like  
`roadStep :: (Path, Path) -> Section -> (Path, Path)`
- we are keeping track of two paths

# Assignment Project Exam Help

113

<https://powcoder.com>

## Add WeChat powcoder

## roadStep

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
  let
    timeA = sum (map snd pathA)
    timeB = sum (map snd pathB)
    forwardTimeToA = timeA + a
    crossTimeToA = timeB + b + c
    forwardTimeToB = timeB + b
    crossTimeToB = timeA + a + c
    newPathToA =
      if forwardTimeToA <= crossTimeToA
      then (A, a):pathA
      else (C, c):(B, b):pathB
    newPathToB =
      if forwardTimeToB <= crossTimeToB
      then (B, b):pathB
      else (C, c):(A, a):pathA
  in (newPathToA, newPathToB)
```

114

## roadStep Notes

- ``sum (map snd pathA)`` totals times for the roads chosen in a path
- note that ``A`` is just enum label, while ``a`` is a variable of type ``Int``
- forming paths as we step will prefix each new pair, instead of concatenation at the end
  - ``:`` operation at start of list is faster than ``++`` at end of list
  - the paths are unfortunately backward order, but we can reverse the result easily

Assignment Project Exam Help

115

<https://powcoder.com>

Add WeChat powcoder

## Using roadStep

Execute ``roadStep`` function on the first section of ``roadsToTower``:

```
roadStep ([],[]) (head roadsToTower)
```

- the first parameter is the start of our empty paths
- the second parameter is the first ``Section``
- read the result for each path list from right to left order

116

foldl to  
a Solution

Use the `roadStep` function to write the `optimalPath` solution, by repeatedly calling `roadStep` per `Section`:

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in if sum (map snd bestAPath) <= sum (map snd bestBPath)
        then reverse bestAPath
        else reverse bestBPath
```

Give the solution a try:

```
optimalPath roadsToTower
```

- the last pair of the path `(C, 0)` can be ignored

Assignment Project Exam Help

117

<https://powcoder.com>

Add WeChat powcoder

— Using Input for the Road System —

118

## groupsOf

Design a few utility functions to help with parsing standard input for setting up the `RoadSystem` data.

The following will help split a list into equal-sized sections:

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

for input `[1..10]`, `groupsOf 3` should evaluate to:

```
groupsOf 3 [1..10]
[[1,2,3],[4,5,6],[7,8,9],[10]]
```

## Assignment Project Exam Help

119

<https://powcoder.com>

## Add WeChat powcoder

## Path Solution

Altogether:

```
import Data.List

main = do
  contents <- getContents
  let
    threes = groupsOf 3 (map read $ lines contents)
    roadSystem = map (\[a, b, c] -> Section a b c) threes
    path = optimalPath roadSystem
    pathString = concat $ map (show . fst) path
    pathTime = sum $ map snd path
  putStrLn $ "The best path to take is: " ++ pathString
  putStrLn $ "Time taken: " ++ show pathTime
```

120

## Road Data

Save the following in `paths.txt` for the `'Section'` times:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

# Assignment Project Exam Help

121

<https://powcoder.com>

## Add WeChat powcoder

## Finishing Up

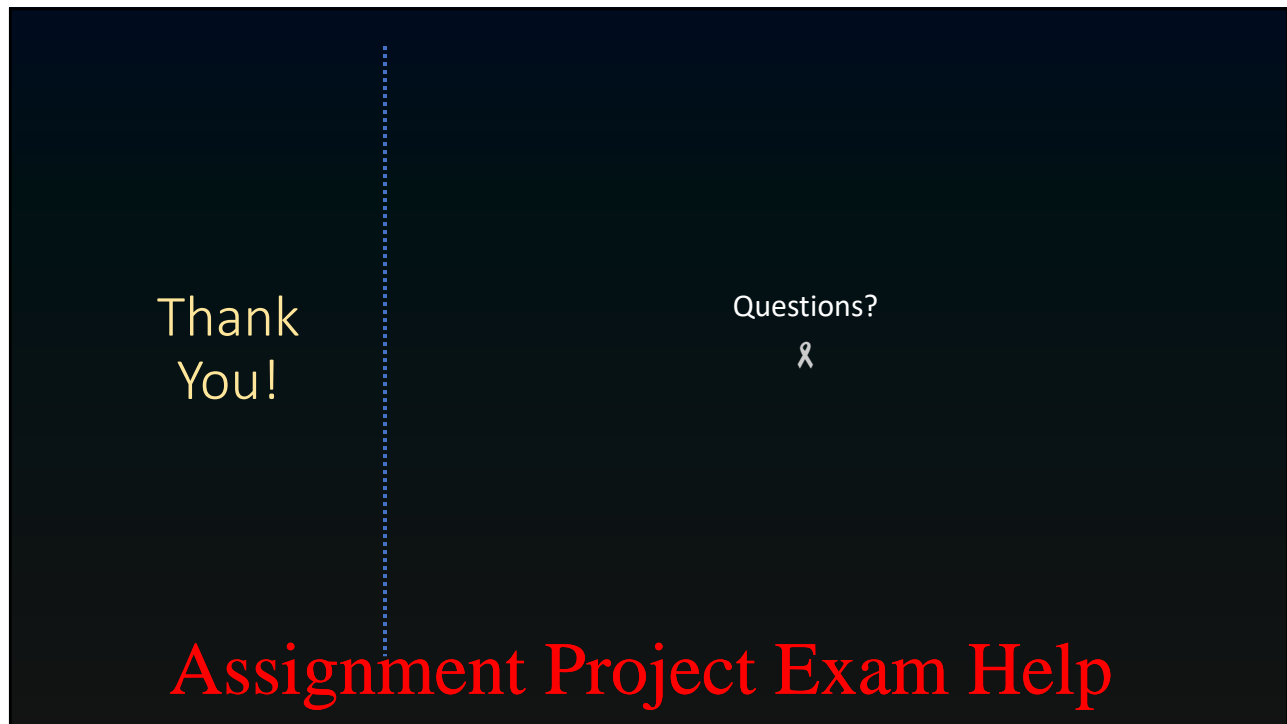
Run the program with `paths.txt` redirected as input.

You should get the result:

```
The best path to take is: BCACBBC
Time taken: 75
```

- try making a random `'Section'` of roads
- if stack overflow, use `'foldl''` and `'foldl' (+)'` instead of `'sum'`

122



123

<https://powcoder.com>

Add WeChat powcoder