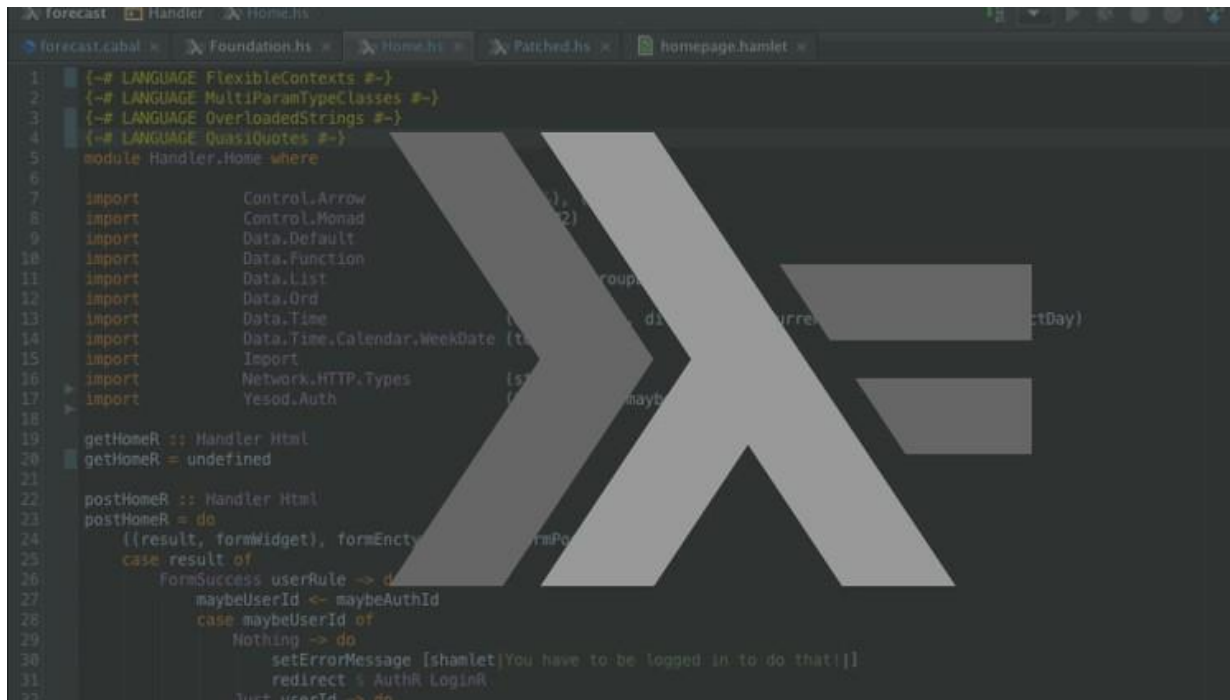


Graph Problems



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Monad
8 import Control.Monad
9 import Data.Default
10 import Data.Function
11 import Data.List
12 import Data.Ord
13 import Data.Time
14 import Data.Time.Calendar.WeekDate (toDayOfYear, dayToWeekDate)
15 import Network.HTTP.Types
16 import Yesod.Auth
17
18 getHomeR :: Handler Html
19 getHomeR = undefined
20
21 postHomeR :: Handler Html
22 postHomeR = do
23   ((result, formWidget), formEnctype) <- runFormPost
24   case result of
25     FormSuccess userRule -> do
26       maybeUserId <- maybeAuthId
27       case maybeUserId of
28         Nothing -> do
29           setErrorMessage [shamlet|You have to be logged in to do that!|]
30           redirect % AuthR.LoginR
31         Just userId -> do
```

Problem 3



```
forecast.cabal x Handler x Home.hs
forecast.cabal x Foundation.hs x Home.hs x Patched.hs x homepage.hamlet x
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow
8 import Control.Monad
9 import Data.Default
10 import Data.Function
11 import Data.List
12 import Data.Ord
13 import Data.Time
14 import Data.Time.Calendar.WeekDate (toDayOfYear, dayToWeek)
15 import Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype, formPost) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

Problem 3



Write a predicate *connectedcomponents* $:: \text{Graph} \rightarrow [[\text{Node}]]$ that splits a graph into its connected components.

Use the following Graph Notation:

```
type Node = Int
type Edge = (Node,Node)
type Graph = ([Node],[Edge])
```

Problem 3



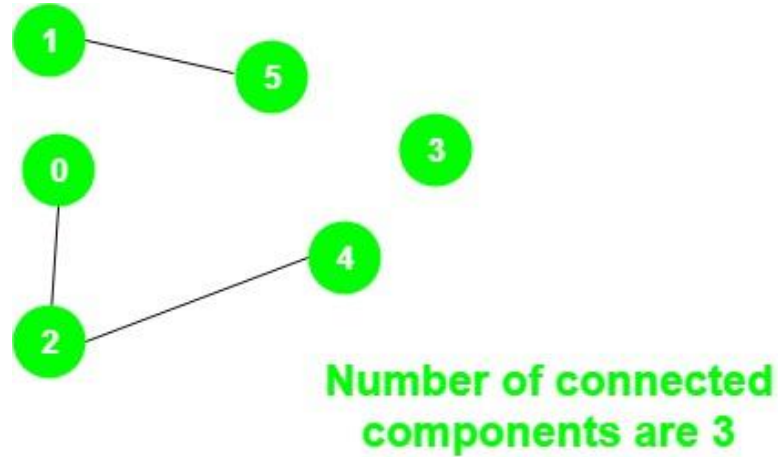
Write a predicate *connectedcomponents* :: *Graph* -> *[[Node]]* that splits a graph into its connected components.

Examples

```
connectedcomponents  
([1,2,3,4,5,6,7], [(1,2),(2,3),(1,4),(3,4),(5,2),(5,4),(6,7)])
```

```
-> [[1,2,5,4,3],[6,7]]
```

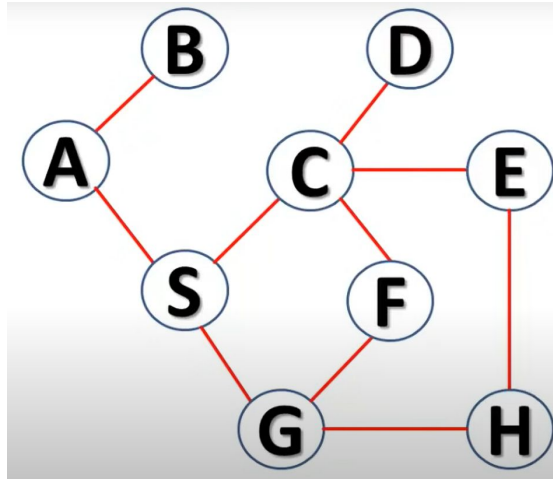
Connected Component



Depth-First Search



The algorithm starts at a starting node and explores as far as possible along each branch before backtracking.



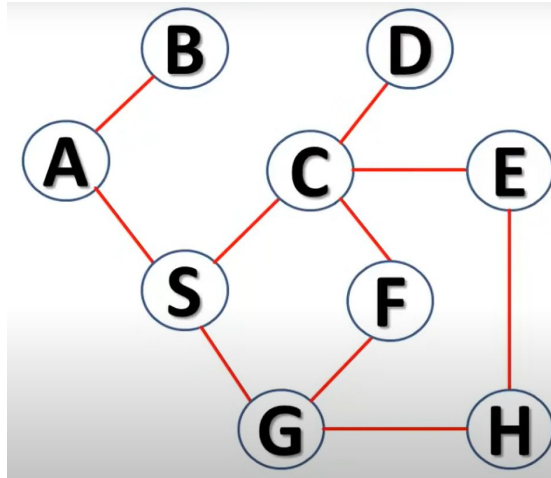
A B S C D E H G F

Depth-First Search



The algorithm starts at a starting node and explores as far as possible along each branch before backtracking.

We start at A



A B S C D E H G F