Project: Crossing Narrow Bridges

Contents

[Project Description: 2](#_Toc57159540)

[Notes and Assumptions: 2](#_Toc57159541)

[References: 2](#_Toc57159542)

[Strategy 3](#_Toc57159543)

[Alternative Strategy 3](#_Toc57159544)

[Architecture and Design 4](#_Toc57159545)

[Project Type, Frameworks and Components 4](#_Toc57159546)

[DB Context 4](#_Toc57159547)

[Models 4](#_Toc57159548)

[Profiles 4](#_Toc57159549)

[Providers (Business Logic) 4](#_Toc57159550)

[Controllers 5](#_Toc57159551)

[Testing 5](#_Toc57159552)

[Project Type Project Type, Frameworks and Components 5](#_Toc57159553)

[Coverage 5](#_Toc57159554)

[Data Sources 5](#_Toc57159555)

[Explanation 6](#_Toc57159556)

# Project Description:

Write a C# program that simulates a team hiking through a forest at night. The team encounters a series of narrow bridges along the way. At each bridge they may meet additional hikers who need their help to cross the bridge.

The narrow bridge can only hold two people at a time. They have one torch and because it is night, the torch must be used when crossing the bridge. Each hiker can cross the bridge at different speeds. When two hikers cross the bridge together, they must move at the slower person's pace.

Determine the fastest time that the hikers can cross each bridge and the total time for all crossings. The input to the program will be a **yaml** file that describes the speeds of each person, the bridges encountered, their length, and the additional hikers encountered along the way.  Your solution should assume any number of people and bridges can be provided as inputs.

Demonstrate the operation of your program using the following inputs: the hikers cross 3 bridges,

1. at the first bridge (100 ft long) 4 hikers cross
   1. A can cross at 100 ft/minute,
   2. B at 50 ft/minute,
   3. C at 20 ft/minute, and
   4. D at 10 ft/minute,
2. at the second bridge (250 ft long) an additional hiker crosses with the team
   1. E at 2.5 ft/minute,
3. at the last bridge (150 ft long) two hikers are encountered
   1. F at 25 ft/minute and
   2. G at 15 ft/minute.

## Notes and Assumptions:

1. Assume that the goal is to move all hikers over every bridge, and they are always crossing the same direction, except when one must bring the torch back to help the next hiker.
2. Assume that the torch handler must cross the bridge completely with each companion before being able to cross back to assist the next hiker.
3. Assume that the hikers that crossed a bridge join the team of hikers at the next bridge. Meaning that the hikers at the first bridge will cross all bridges, the hikers at second bridge will cross all bridges except for the first one, etc.…
4. Assume that the torch initially available to the team at the first bridge.
5. Assume that there is only one torch for ALL hikers, and it originally handled by team at the first bridge. If each team at every bridge had a torch, then the Strategy and the Architecture described below would change.
6. **NOTE** that for the current exercise, I modified the order in which the bridges are crossed to simulate transfer of the torch at some point to the fastest hiker that just joined the team.

## References:

GitHub Source Code: <https://github.com/eitkin72/Geojam.CrossingBridges>

# Strategy

(There are several ways to solve the problem, you can provide more than one. The goal is to show us how you think)

1. Input data must provide the order in which the bridges are crossed. I have defined an additional Bridge.Order property to the yaml file. The alternative approach would be to use the original order of data in the yaml file.
2. Generate a SortedList<Bridge> with the Key being the Bridge.Order value.
3. Generate a SortedList<Hiker> that are ready to cross the bridge. Maintain the order in this list based on each Hiker’s bridge crossing speed in ascending order. This list will contain all Hikers that are originally assigned to the Bridge by the input data and all Hikers that crossed previous Bridges. The last Hiker in this SortedList is the torch handler for current bridge; this Hiker is the fastest of the current team according to the SortedList order.
4. The team’s fastest Hiker will always carry the torch to make round trips until all Hikers have crossed the Bridge.
5. The order of Hikers crossing each bridge does not matter as soon as the fastest Hiker is assigned to be the torch handler.

Alternative Strategy:

If a torch was available to every team at each bridge the following changes could be addressed:

1. Maintain several threads – one at each bridge -- to handle crossings of constantly changing teams.
2. Each Hiker object would maintain its status that would track if the trip is over (are there next bridge?)
3. The threads fire on simultaneously and will execute until all Hikers status is complete.
4. Each thread would maintain its own SortedList<Hiker> that can be modified by another thread when Hikers cross previous bridge. The thread safety must be preserved.
5. Need to make sure that Hikers cross the bridge in descending order of crossing speed.
6. Need to make sure that each time a Hiker crosses the Bridge, it is removed from this team of this Bridge to the team of the next Bridge. This transfer must keep track of time to make sure that transfers from previous Bridge that are affecting the team composition is handled.

# Architecture and Design

(We want to see how well you architect and design solutions to complex problems)

## Project Type, Frameworks and Components

* Use ASP.NET Core Web App to create WebApi Service application.
* Use Microsoft.EntityFrameworkCore and Microsoft.EntityFrameworkCore.InMemory to decouple the implementation from the original from data source (yaml file) and to create in memory DB Context
* Use YamlDotNet.NetCore nuget package for yaml deserialization.
* Use AutoMapper.Extensions.Microsoft.DependencyInjection for models mappings
* Use xUnit to implement Unit Testing

## DB Context

* Bridge – defines bridge entity model and relationships.
* Hiker - defines hiker entity model and relationships.
* GoejamDbContext – defines datasets for the DB Context.

## Models

* Bridge – defines bridge application model.
* Hiker - defines hiker application model.

## Profiles

* AutoMapperProfile – initializes mapping expressions

## Providers (Business Logic)

* BridgeProvider implements IBridgeProvider
  1. Constructor
     + Handles dependency injections.
     + Seeds the data from data source (yaml file in this case).
  2. GetBridgesAsync
     + Return: List<Bridge> mapped from the data source.
  3. GetSortedBridgesAsync
     + Param: List<Bridge>.
     + Return: SortedList<Bridge>.
  4. CrossBridgeAsync
     + Param: SortedList<Hikers> - additional Hikers (from previous bridges).
     + Param: Bridge – bridge to cross.
     + Return:
       - Crossing Time for this bridge by additional Hikers and Bridge.Hikers.
       - New set of hikers = additional hikers + the team of hikers at the current bridge.
  5. CrossBridgesAsync
     + Return: Total Crossing Time for all bridges by all hikers.
* HikerProvider implements IHikerProvider
  1. Constructor
     + Handles dependency injections.
  2. CrossBridgeAsync
     + Param: Hiker
     + Param: Bridge
     + Return: Bridge Crossing Time by Hiker

## Controllers

* BridgeController:
  1. Constructor –
     + Handles dependency injections.
  2. CrossBridgesAsync - gets total time to cross all bridges for all teams.
  3. GetBridgesAsync - (not related to the problem) gets List<Bridge>
  4. GetSortedBridgesAsync – (not related to the problem) gets SortedList<Bridge>

# Testing

(We want to see how you approach testing of the solution. Standards and best practices)

## Project Type Project Type, Frameworks and Components

* Use ASP.NET Core Web App to create WebApi Service application.
* Use Microsoft.EntityFrameworkCore
* Use xUnit Test Project (.NET Core).

## Coverage

* BridgesServiceTest - Selective Unit Tests of the BridgeProvider.
  1. Constructor:
     + Initialize some test data for bridges and hikers lists
     + Configure and Initialize the AutoMapper using AutoMapperProfile from the main project
  2. GetBridgesReturnsAllBridges
  3. GetSortedBridgesReturnsSortedBridgesCollection
  4. CrossBridgesReturnsTimeToCrossAllBridgesByAllHikers
     + Param: dbContextUniqueIndex – unique index that is used to initialize the GeojamDbContext to avoid elements collisions during multiple test executions. Each test run spins out a new Context database.
     + Param: includeBridgeIds – filter BridgeIds that are selected from the bridges list to be included in the test case.
     + Param: expected – expected result

## Data Sources

* Each Test Case initializes its own in-memory DB Context using private readonly bridges and hikers lists.
* Test Theory for BridgeProvider.CrossBridges uses BridgesServiceData.Data data source to feed the test parameters.

# Explanation

(As a C# thought leader in the organization you will be working with and mentoring other engineers. How well you can describe and explain your solution is very important)