CO202: Coursework 1

Autumn Term, 2019

1 *Introduction*

This goal of this coursework is to implement algorithms that will be used to control your army in a game of Imperial Conquest: a real-time strategy game that has been created for this course. It takes place in a galaxy far, far away, where space travel to different planets is made possible through a network of wormholes. Planets are able to produce fleets of ships that can be sent to divide and conquer the enemy.

Both coursework assignments will use the same basic infrastructure. This first assignment concerns itself with choosing a basic initial strategy and some simple pathfinding. The second assignment will deal with more complex strategies and dealing with opponents.

2 *Submission*

The code in this specification can be found in `Submission.lhs`, which is a trimmed down version of what is presented here, that contains all the relevant code. Your task is to modify `Submission.lhs` by implementing the solution to the questions and writing a report explaining your work.

As it happens, `Submission.lhs` is a *Literate Haskell* file (as suggested by the `.lhs` extension), which means it doubles up as a LaTeX document and a `Haskell` source file that `GHC` understands. You should be able to compile your `Submission.lhs` into a `pdf` by invoking `pdflatex Submission.lhs`.

Your coursework submission will consist of:

1. `Submission.lhs`
2. `Submission.pdf`

You will need to have an installation of both `GHC 8.0.2` or better and a version of `pdflatex` to work on this coursework.

Since the source file is also valid Haskell, it can be loaded into `GHCi` by running `ghci Submission.lhs`. This way, you can test your solutions, and ensure there are no compilation errors as you're developing your code.

To add a new code block, simply put it between `\begin{code}` and `\end{code}`, which ensures that `GHC` recognises it as source code, and it will be properly syntax highlighted in the resulting `pdf` document.

When you want to omit code from the produced document, you can do so by putting the `code` block between `\begin{comment}` and `\end{comment}`. This will hide it from the document, but `GHC` will still load it. As an example, the module header and import lines are hidden in this way.

Since your submission will be tested by automatically linking it against a test suite, it is very important that you do not modify the type signatures of top-level functions. To verify you have not accidentally modified anything crucial, you can invoke

```
ghc -fforce-recomp -c Submission.lhs-boot Submission.lhs
```

which tells you if there are any mismatches between your source file and the expected interface. Do not modify `Submission.lhs-boot` in any way. Only submissions that compile against this interface will be awarded marks, so please double check that this is the case. However, feel free to define new helper functions as you see fit.

The `pdf` you submit should describe the interesting parts of your solutions, and does not need to contain all your code explicitly. You should aim for your report to be precise, concise, and well presented. All problems should be solved, and the structure should clearly indicate which problem is begin solved. As an absolute page limit it should not exceed 4 pages, but you should aim to keep it lower than this. You will be judged on quality not quantity.

3 *Background*

Imperial Conquest is a game inspired by Galcon, a two player real-time strategy game. A game of Imperial Conquest is played between two players, and time progresses in a series of turns.

All of the datatypes in this section are relied upon for communication with the server and should not be modified.

The `Player` data type represents the two players of the game.

```haskell
data Player = Player1 | Player2
```

This first coursework involves no interaction between players, but the data types will be shared between the second coursework, and so are defined here.

The game is played on a map with a number of planets. Planets are represented by the `Planet` type.

A **`newtype`** uses an existing type as the basis of a new type by wrapping values in a new constructor. Here it means that a `Ship` cannot be confused with a `Growth`, even though both are fundamentally storing an **`Int`**.

```haskell
data Planet = Planet Owner Ships Growth
newtype Ships = Ships Int
newtype Growth = Growth Int
```

Thus, a planet is a value `Planet owner ships growth`, where an `owner` can be either neutral, or one of the two players:

```haskell
data Owner = Neutral | Owned Player
```

Planets have a number of spaceships in garrison, represented by `ships` that belong to the owner of the planet. Finally, each turn, planets that are owned by a player have factories which can produce a number of ships given by the `growth` value.

In order to refer to planets, they each have an identifying number, which is presented as a value of type `PlanetId`.

```
newtype PlanetId = PlanetId Int
```

The `Planets` type represents a collection of planets as a key-value map from planet identifiers to the planet structures.

A `Map` is a data structure that is provided in the `Data.Map` module, which is documented at `https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Lazy.html`, which gives operations and their complexities.

```
type Planets = Map PlanetId Planet
```

The planets in this galaxy are too far away for spaceships to travel between them within a reasonable amount of time using traditional propulsion systems, but certain planets are connected by wormholes, allowing spaceships to travel through them faster than light.

```
data Wormhole = Wormhole Source Target Turns

newtype Source = Source PlanetId
newtype Target = Target PlanetId
newtype Turns  = Turns Int
```

Crucially, a `Wormhole` connects two planets in a *directed* way. That is, wormholes are one-way streets with a set value of type `Source` for the source, and `Target` for the target. The value of type `Turns` indicates how many turns it takes for a spaceship to travel through the wormhole.

Similarly to `Planets`, `Wormholes` are also referred to by an identifier which is just a wrapper around an **`Int`**:

```
newtype WormholeId = WormholeId Int
```

These are collected into a key-value map in much the same way:

```
type Wormholes = Map WormholeId Wormhole
```

As spaceships go through wormholes, the number of turns left on their journey before they arrive is tracked. The `Fleet` data type represents a fleet of ships by storing whose ships they are, how many of them are there, which wormhole they're in, and how many turns left before they reach their target.

```
data Fleet = Fleet Player Ships WormholeId Turns
```

When a fleet of ships arrives at the target of the wormhole, they join forces with the ships that are there if they belong to the same owner. Otherwise, the fleet attacks the ships on the defending planet, cancelling each other out one-to-one. If there are more ships in the fleet than on the planet then the remaining ships from the fleet establish themselves on the planet garrison, and the planet is owned by the fleet owner.

The `Fleets` type represents a collection of fleets. Note that these do not have identifiers, as they are not referred to anywhere.

```
type Fleets = [Fleet]
```

The state of the map at any given time is represented by the `GameState` data type:

```
data GameState = GameState Planets Wormholes Fleets
```

Finally, at every turn, the server takes a list of orders and begins to execute them.

```
data Order = Order WormholeId Ships
```

The exact rules of the game are not important for this assignment, though they will become relevant in the next.

4 *Dynamic Programming*

Dynamic programming is a technique for optimising the runtime performance of a recursive algorithm that has overlapping subproblems. The speedup comes from storing the subsolutions for later use instead of recomputing them every time they are needed.

The strategy is developed in two stages:

1. Write an inefficient recursive algorithm that solves the problem.

2. Improve efficiency by storing intermediate shared results.

As a simple example, dynamic programming can be applied to speed up a naive implementation of the Fibonacci function.

While the input to `fib` is usually bound by an **`Int`**, even for small inputs the function will overflow in its output, hence the return type is **`Integer`**, which can represent arbitrarily large integers.

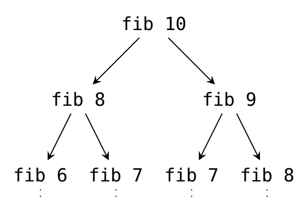

Figure 1: Call tree of `fib 10`, showing multiple repeated computations.

Each Fibonacci number for a given value `n` is given by `fib n` given by the following recursive algorithm:

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

This code is remarkably inefficient, since there are repeated calls to computations that recalculate the same value. Figure 4 shows that `fib 8` is called twice in the calculation of `fib 10`. In turn, `fib 7` is called three times, `fib 6` five times, etc. As an approximation, assume that the two subcalls of `fib` have the same cost, giving the recurrence relation:

$$T_{\texttt{fib}}(n) \leq 1 + 2 \times T_{\texttt{fib}}(n-1)$$

Solving this recurrence gives $T_{\texttt{fib}}(n) \in O(2^n)$. This is remarkably expensive and due to the fact that there are large overlaps in the solutions of subproblems, which keep getting recalculated.

This recalculation can be avoided by building up the intermediate values up to the result. More concretely, the computation of `fib n`, involves defining an array containing the values of `fib` for *all* numbers from 0 up to `n`.

The **`Array Int Integer`** type represents an array indexed by **`Int`** containing values of type **`Integer`**.
The function `(!)` provides constant-time random access.
The function **`array`** takes a range `(u,v)` of values as its bounds, and a list of pairs where each pair `(i,x)` is used to place `x` at index `i` in the table.

```
table :: Int -> Array Int Integer
table n = array (0,n) [ (0, 0)
                      , (1, 1)
                      , (2, table ! 0 + table ! 1)
                      , (3, table ! 1 + table ! 2)
                      , ... ]
```

Notice that every element of `table` refers to solutions of previous problems that eventually lead to a base case: there are no circular references so the self-referential definition is well-defined. Here is a version of `fib` that builds the right `table` and returns the last element of the array.

This is, of course, not the best way to calculate Fibonacci numbers (which can be done in sublinear time), but it illustrates how a recursive algorithm can be made more efficient.

```
fib' :: Int -> Integer
fib' n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0, n) mfib

    mfib 0 = 0
    mfib 1 = 1
    mfib n = table ! (n-1) + table ! (n-2)
```

It is important that `mfib` is in the same level of scope as `table`: if it were top-level then a new table would be created on each call!

The table given by `tabulate (x,y) f` contains the results of applying `f` to all the values between `x` and `y`. It is implemented as an array which gives constant time access to its elements.

The constraint **`Ix`** `i` allows the values of type `i` to be drawn from those given by the `range` function, as well as enabling values to be indexed over in an array. This allows arrays to be indexed by types other than **`Int`**. For instance, a valid index is a tuple `(Int, Int)` for a two-dimensional array indexed by pairs of **`Int`**s.

```
tabulate :: Ix i => (i,i) -> (i -> a) -> Array i a
tabulate (u,v) f = array (u,v) [ (i, f i) | i <- range (u, v)]
```

The cost of building this table is the sum of all the individual calls of `f`. The key to efficiency is that the function `f` can itself refer to the table that is being constructed. If the cost of `f` is constant, such that $T_{\texttt{f}}(i) \in O(1)$, and the table has n elements, then the cost of its construction is $T_{\texttt{table}}(n) \in O(n)$.

In the code above, `mfib` is a local version of `fib` that finds values in the table rather than by recursion. The function takes constant time since `(!)` is a constant time operation. Thus, the time complexity of evaluating `fib' n` is given by:

$$T_{\texttt{fib'}}(n) = 1 + T_{\texttt{table}}(n) + T_{!}(n)$$

where $T_{\texttt{table}}(n)$ is the time it takes to construct the table, and $T_{!}(n)$ is the time it take to look up a value in that table. Therefore, the overall cost is $T_{\texttt{fib'}}(n) \in O(n)$, which is much better than before.

The slogan for dynamic programming algorithms is to *trade space for speed*: the table takes space in memory to construct, but results in a much faster algorithm.

5 *Planet Picking*

The first task of this coursework is to plan which planets to conquer first. To simplify things, assume that there are a number of planets that are equally reachable from your forces.

As an example, here is a map with 5 planets, planet `0` being your home planet, and planets `1..4` being the neutral planets that you can conquer, each in a single turn. Furthermore, the planets are not reachable from one another.

```haskell
example1 :: GameState
example1 = GameState planets wormholes fleets where
  planets = M.fromList
    [ (PlanetId 0, Planet (Owned Player1) (Ships 300) (Growth 0))
    , (PlanetId 1, Planet Neutral         (Ships 200) (Growth 50))
    , (PlanetId 2, Planet Neutral         (Ships 150) (Growth 10))
    , (PlanetId 3, Planet Neutral         (Ships 30)  (Growth 5))
    , (PlanetId 4, Planet Neutral         (Ships 100) (Growth 20))
    ]
  wormholes = M.fromList
    [ (WormholeId 0, Wormhole homePlanet (Target 1) (Turns 1))
    , (WormholeId 1, Wormhole homePlanet (Target 2) (Turns 1))
    , (WormholeId 2, Wormhole homePlanet (Target 3) (Turns 1))
    , (WormholeId 3, Wormhole homePlanet (Target 4) (Turns 1))
    ] where homePlanet = Source 0
  fleets = []
```

The information in a `GameState` can be queried. For instance, the `targetPlanets` function lists the planets that can be reached from a given source, and the `shipsOnPlanet` function finds how many ships are garrisoned on a planet.

```haskell
targetPlanets :: GameState -> Source -> [(PlanetId, Ships, Growth)]
targetPlanets st s
  = map (planetDetails . target) (M.elems (wormholesFrom s st))
  where
    planetDetails :: PlanetId -> (PlanetId, Ships, Growth)
    planetDetails pId = (pId, ships, growth)
      where Planet _ ships growth = lookupPlanet pId st
```

```
shipsOnPlanet :: GameState -> PlanetId -> Ships
shipsOnPlanet st pId = ships
  where Planet _ ships _ = lookupPlanet pId st
```

Both of these functions use `lookupPlanet` to extract a planet from the `GameState`:

```
lookupPlanet :: PlanetId -> GameState -> Planet
lookupPlanet pId (GameState ps _ _) = fromJust (M.lookup pId ps)
```

It is also possible to determine the wormholes that correspond to a planet, whether that be wormholes to or from that planet:

```
wormholesFrom :: Source -> GameState -> Wormholes
wormholesFrom pId (GameState _ ws _)
  = M.filter (\(Wormhole s _ _) -> s == pId) ws

wormholesTo :: Target -> GameState -> Wormholes
wormholesTo pId (GameState _ ws _)
  = M.filter (\(Wormhole _ t _) -> t == pId) ws
```

Here is an example of how `ghci` can be used to query some values:

```
ghci> targetPlanets example1 (Source (PlanetId 0))
[(1,200,50),(2,150,10),(3,30,5),(4,100,20)]
```

If the overall number of defenders among all these planets is overwhelming, you will have to pick which planets to conquer first. Planets have a different intrinsic value, which is represented by their growth rate. The question is which planets to conquer in the first turn to maximise the growth rate. In the case of `example1`, the optimal strategy if your fleet has `300` ships is to conquer planet `1` and planet `4`, resulting in a growth rate of `70`.

Planets have different intrinsic value, so you should aim to maximise the value of your conquest given your capacity to attack. This is optimisation problem is an example of the classic *knapsack* problem.

5.1 Unbounded Knapsack

The unbounded knapsack problem concerns itself with packing a knapsack with some given capacity `c` with elements of some weight and value drawn from a list. There is no limit to the number of times an element can be picked from the list. The goal is to maximise the value of the items that can be placed in the list, without going over the capacity.

Thus, the available items are represented as a list of triples, where each item has a name, some weight, and some value.

The `name` parameter is not used in this first function, but it will become useful later.

```
[(name, weight, value)]
```

As a recursive algorithm, `knapsack` is easily stated: take an item `(name, weight, value)`, and consider what would happen if we put it in the knapsack. The total value would be increased by the `value` of the item, but the remaining capacity would be decreased by the `weight` of the item. So, if the starting capacity is `c`, then picking this item would result in a maximum value of the item's `value` plus the maximum value of the subproblem where the capacity is `c-w` (since after putting this item in, we need to maximise the value by filling in the remaining capacity). The optimal solution is achieved by trying every item as the first one, then recursively optimising the remaining capacity, ultimately finding the maximum one.

This recursive description can be written as the following:

The `knapsack` function is highly polymorphic, as it works on any types of inputs, as long as they can be compared (as stated by the **`Ord`** constraint) and support basic arithmetic (as stated by the **`Num`** constraint). Here the abstraction helps make clear exactly which properties of the types are relied upon.

```
knapsack :: (Ord weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> weight -> value
knapsack wvs c = maximum 0 [ v + knapsack wvs (c - w) | (_,w,v) <- wvs , w <= c ]
```

That is, from the input list `wvs`, find all elements whose weight is less than the capacity, and try them by recursively solving the smaller problem.

Finally, take the largest element of this list using **`maximum`**.

```
maximum :: Ord a => a -> [a] -> a
maximum x xs = foldr max x xs
```

The maximum value here is bounded below by `0`.

This implementation of `knapsack` correctly produces the maximum value, which can be tested in `ghci`:

```
ghci> knapsack [("a", 35, 10), ("b", 153, 200), ("c", 100, 20)] 800
1010
```

However, the efficiency of this algorithm is quite terrible when the knapsack can be filled to a given capacity by picking different combinations of elements, since this leads to repeated computations. Try increasing the capacity to see how slow it gets.

Problem 1: *Dynamic Knapsack*

Use dynamic programming to improve the running time of `knapsack`, by giving a definition of `mknapsack` in the code below.

```
knapsack' :: forall name weight value .
  (Ix weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> weight -> value
knapsack' wvs c = table ! c
  where
    table :: Array weight value
    table = tabulate (0,c) mknapsack
```

Hint: compare the recursive version of `fib` with the dynamic programming version to see how to translate one to the other.

The `forall` is used to put the type variables `name`, `weight`, and `value` in scope so that they can be referred to in the **`where`** clause.

Notice the additional (**`Ix`** `weight`) type class constraint in the function's signature. This indicates that the `weight` will be used as an array index when building the table for the subproblem solutions.

```
    mknapsack :: weight -> value
    mknapsack c = undefined
```

While this correctly and efficiently calculates the maximum value of the knapsack, it does not announce what the items that are picked actually are (in other words, the `name` elements are ignored). To do this, the return type `Value` needs to be modified to something that holds both the value and the index of the item that was chosen. Then, you will have to modify the algorithm so that the index does not get in the way, and so that the indices are properly combined.

Problem 2: *Knapsack Elements*

Implement `knapsack''` as a modified version of `knapsack'` so that it outputs the maximum value of the knapsack, as well as a list of the element indices that are chosen to obtain that value.

```
knapsack''
  :: forall name weight value
  . (Ix weight, Num weight, Ord value, Num value)
  => [(name, weight, value)] -> weight -> (value, [name])
knapsack'' wvs c = table ! c
  where
    table :: Array weight (value, [name])
    table = tabulate (0,c) mknapsack

    mknapsack :: weight -> (value, [name])
    mknapsack c = undefined
```

Hint: you may need to make use of **`maximumBy`** to be able to ignore the indices that are being used. A fantastic site for finding useful functions is `https://hoogle.haskell.org/`, where even type signatures can be typed in.

If your solution is correct, you should see something like this:

```
ghci> knapsack'' [("a", 35, 10), ("b", 153, 200), ("c", 100, 20)] 800
(1010,["b","b","b","b","b","a"])
```

5.2 *Bounded Knapsack*

The unbounded knapsack problem allows the items to be used multiple times. However, the planets can only be conquered once, so it doesn't make much sense to allow them to be picked multiple times. The goal of this section is to implement the *bounded knapsack problem* (sometimes called the 0-1 knapsack problem) to help decide which planets should be conquered.

Problem 3: *Bounded Knapsack*

This is the type of `bknapsack`, which is similar to `knapsack` except that the elements are not replaced when picked.

Hint: Perform case analysis on the list of items. After using an item, remove it from the recursive call.

```haskell
bknapsack
  :: (Ord weight, Num weight, Ord value, Num value)
  => [(name, weight, value)] -> weight -> (value, [name])
bknapsack = undefined
```

To implement this function, first write a *recursive* definition, where the result of `bknapsack wvs c` is the maximum value that can be packed into a capacity of `c` from the elements of `wvs` by only using each element at most once.

The solution to `bknapsack` (hopefully) looks simpler, or at least not much more difficult, than for `knapsack`. However, it turns out to be trickier to apply dynamic programming to the bounded case than the unbounded case. To understand why, consider the previous applications of dynamic programming. Both in the case of `fib` and `knapsack`, the solution was to inductively iterate through the space of inputs until reaching the desired value. In case of `fib`, the **`Int`**s were iterated, and in the case of `knapsack`, the `weight` values. The reason this approach works for `knapsack` is because in the recursive call, only the `weight` parameter changes, so the table can be built only indexed by that parameter alone.

In the case of `bknapsack`, there are two varying parameters: the `weight` as before, but also the input list, as the picked element gets removed in every iteration.

Problem 4: *Reasonable Indexes*

A first approach might be to simply make the array indexed by not only the `weight` parameter, but also the `[(name, weight, value)]` list. Explain in words why this approach is not reasonable.

Hint: Think about how the array is created and what might make a suitable index.

Problem 5: *Bounded Knapsack Revisited*

Write another recursive solution, `bknapsack'`, to the bounded knapsack problem. This time, do not change the list in the recursive case, but instead try to introduce another parameter that keeps track of progress.

Hint: Take a careful look at the type signature, and notice the extra **`Int`** parameter. Use this to keep track of which part of the list has been processed.

```haskell
bknapsack' :: forall name weight value .
  (Ord weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> Int ->
  weight -> (value, [name])
bknapsack' = undefined
```

Hint: The **`Array`** type can be indexed by any type that has an **`Ix`** instance. Notably, tuples of indexable types are also indexable, which means for example it is possible to have an array of type `Array (Int, Int) Bool`, which can be thought of as a two dimensional array.

Problem 6: *Dynamic Bounded Knapsack*

Using dynamic programming implement `bknapsack''`, which is the efficient version of `bknapsack'` that makes use of tables rather than repeated recursion.
Hint: when populating the table, use the new **`Int`** parameter instead of the input list as part of the index.

```
bknapsack'' :: forall name weight value .
  (Ord name, Ix weight, Ord weight, Num weight,
   Ord value, Num value) =>
  [(name, weight, value)] -> weight -> (value, [name])
bknapsack'' = undefined
```

To put all the pieces together, the optimal conquering strategy can finally be calculated from any source planet:

```
optimise :: GameState -> Source -> (Growth, [PlanetId])
optimise st s@(Source p) = bknapsack'' (targetPlanets st s) (shipsOnPlanet st p)
```

For example, the `example1` map state should yield

```
ghci> optimise example1 (Source 0)
(70,[4,1])
```

6 *Directed Graphs*

Navigating through the network of wormholes will require a representation of that network, and this is nicely achieved by using a *graph*. A graph consists of vertices and edges. For simplicity, we will focus on the definition of a weighted directed simple graph: since it is weighted and directed, each edge has a specified source, target, and weight. Weights will be assumed to simply be of type **`Integer`**.

There are many kinds of graphs, depending on whether there are cycles, whether edges are directed, whether there can be only up to one edge between to vertices, and whether the edges have weights.

```
type Weight = Integer
```

This is encapsulated by the following class, where `e` is the type of edges, and `v` is the type of vertices:

The `e -> v` part of the class declaration is called a *functional dependency*, and it describes a relation between edges and vertices. Here it says that knowing the type of an edge determines the type of the vertex.

```
class Eq v => Edge e v | e -> v where
  source :: e -> v
  target :: e -> v
  weight :: e -> Weight
```

This abstraction means that different types can be considered to be edges, which will become useful later on.

As an example, a triple (**`String`**, **`String`**, **`Integer`**) can be thought of as an edge between vertices that are **`String`**s.

```
instance Edge (String, String, Integer) String where
  source (s, _, _) = s
  target (_, t, _) = t
  weight (_, _, i) = i
```

Concretely, `("here","there",10)` is an edge from the source `"here"` to the target `"there"` with weight `10`.

For Imperial Conquest, the `Wormhole` type can readily be made an instance of `Edge` with vertices of type `PlanetId`.

```
instance Edge Wormhole PlanetId where
  source (Wormhole (Source s) _ _)    = s
  target (Wormhole _ (Target t) _)    = t
  weight (Wormhole _ _ (Turns turns)) = toInteger turns
```

Here, the weight of the edge is the number of turns it takes to travel through the wormhole.

Describing edges in this abstract sense using a type class has some nice benefits. It makes it possible to talk about graphs and algorithms on graphs in a way that is not tied to any specific representation. The advantages of abstraction here are twofold: the implemented algorithms are easier to understand because they do not refer to low-level details, and they are also more reusable. Just like `knapsack`, the graph algorithms here will also work on our galaxy, even though the implementations do not mention anything about planets or ships.

It is also possible to attach additional payload to edges when convenient. For instance, since a `Wormhole` is an edge, so is a pair of a `WormholeId` and a `Wormhole`.

This is convenient because when talking about the wormholes in the galaxy, it will be useful to know the identifier as well as the value.

```
instance Edge (WormholeId, Wormhole) PlanetId where
  source (_, w) = source w
  target (_, w) = target w
  weight (_, w) = weight w
```

An edge connects two points in a graph, and is a primitive building block of a graph. Edges compose together when the target of one agrees with the source of the other, making it possible to follow a path from a source vertex to a target vertex through a series of connected edges. In this sense, a *path* between two vertices can be thought of as a list of edges.

Here is a concrete implementation of a `Path`

```
data Path e = Path Weight [e]
```

As an optimisation, it is convenient for a path to also keep track of the sum of all the weights of its edges as an additional parameter. This is redundant information since since the total weight could always be recomputed from just the list alone, but it makes it easier to inspect a useful value.

An edge can trivially be turned into a path:

```
pathFromEdge :: Edge e v => e -> Path e
pathFromEdge e = Path (weight e) [e]
```

This is a path with just one edge.

Since the `Path` type contains a list, it is a lot more efficient to prepend a new element at the front (since `(:)` is $O(1)$) than it is to append at the end (which will cost $O(n)$ to traverse). A path in general can grow on either end (an edge going out of the target, or an edge going into the source), so there is a choice of which operation should be supported more efficiently.

There are of course other data structures such as double-ended queues that support both operations in amortized constant time.

For the purposes of this coursework, a fixed source for paths will be more useful, so a path will only be extended by adding edges out of the target. To support this operation efficiently, the source will be stored at the tail end of the list, and the target is at the head. This means that a path can be extended by with an edge going out of the current target of the path.

```
extend :: Edge e v => Path e -> e -> Path e
extend (Path _ []) _ = error "extend:␣Empty␣path"
extend (Path d (e:es)) e'
  | target e == source e' = Path (d + weight e') (e':e:es)
  | otherwise = error "extend:␣Incompatible␣endpoints"
```

Given a list of edges `es = [e`$_0$ `, ..., e`$_n$`]` where each `e`$_i$ has source vertex `v`$_i$, target vertex `v`$_{i+1}$, and weight `w`$_i$, they can be stitched together into a path:

```
pathFromEdges :: Edge e v => [e] -> Path e
pathFromEdges (x : xs) = foldl extend (pathFromEdge x) xs
pathFromEdges [] = error "pathFromEdges:␣Empty␣list␣of␣edges"
```

```
ghci> pathFromEdges [("a", "b", 10), ("b", "c", 20)]
Path 30 [("b","c",20),("a","b",10)]
```

Here the (**`String`**, **`String`**, **`Integer`**) instance is used. Notice how the list is reversed in order to expose the target of the path at the head of the list.

The `Path` type can actually be thought of as an edge, since it has a source, a target, and a weight. The vertex type is the same as the underlying edge's vertex type.

```
instance Edge e v => Edge (Path e) v where
  source (Path _ es) = source (last es)
  target (Path _ es) = target (head es)
  weight (Path w _)  = w
```

This instance is only possible if, just like edges, paths themselves can be compared for equality.

```
ghci> weight (pathFromEdges [("a", "b", 10), ("b", "c", 20)])
30
```

Building on this abstraction of edges and vertices, a graph can be given by the following class interface:

There are many choices for describing a graph interface. This interface focuses on querying rather than constructing graphs.

```
class Edge e v => Graph g e v | g -> e where
  vertices  :: g -> [v]
  edges     :: g -> [e]
  edgesFrom :: g -> v -> [e]
  edgesTo   :: g -> v -> [e]
  velem     :: v -> g -> Bool
  eelem     :: e -> g -> Bool
```

Usually it is best for the more stable parameters of a function to come first, so graphs before vertices. However, the "elem" functions usually have their arguments in the order so that they can be written as `x ‘elem‘ xs`, to mimic the mathematical notation $x \in X$.

Instances of this type class can be thought of as graphs: creating an instance for a graph takes three type parameters: the first indicates the representation of the graph, the second is for the representation of an edge, and the third is the representation of a vertex.

A simplistic representation of a graph is simply as a list of edges. This can certainly be achieved if the edges can be compared for equality.

```
instance (Eq e, Edge e v) => Graph [e] e v where
  vertices es = nub (map source es ++ map target es)
  edges es    = es
  edgesFrom es v = [ e | e <- es, v == source e ]
  edgesTo   es v = [ e | e <- es, v == target e ]
  velem v es = v ‘elem‘ vertices es
  eelem v es = v ‘elem‘ edges es
```

The `nub` function, from `Data.List`, removes duplicate elements

Thus, the following example is a legitimate graph:

```
example2 :: [(String, String, Integer)]
example2 = [("s","t",10), ("s","y",5), ("t","x",1), ("t","y",2), ("y","t",3),
            ("y","x", 9), ("x","z",4), ("z","x",6), ("y","z",2), ("z","s",7)]
```

It is possible to extract the vertices from this example:

```
ghci> vertices example2
["s","t","y","x","z"]
```

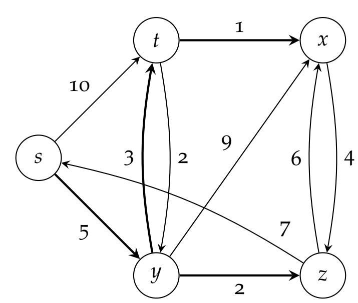

Figure 2: Graph of `example2`, bold lines indicate edges in the shortest path from s to other vertices.

The `GameState` type is an example of a graph: it holds information about planets, which are the vertices, and wormholes, which are the edges. The class instance shows how to extract this information.

```
instance Graph GameState (WormholeId, Wormhole) PlanetId where
  vertices (GameState ps _ _) = M.keys ps
  edges    (GameState _ ws _) = M.assocs ws
  edgesTo   st pId = M.toList (wormholesTo (Target pId) st)
  edgesFrom st pId = M.toList (wormholesFrom (Source pId) st)
  velem pId      (GameState ps _ _) = M.member pId ps
  eelem (wId, _) (GameState _ ws _) = M.member wId ws
```

This is not an efficient implementation, but it does allow for rapid prototyping, and querying the structure directly.

```
ghci> edgesFrom example1 0
[(0,Wormhole 0 1 1),(1,Wormhole 0 2 1),(2,Wormhole 0 3 1),(3,Wormhole 0 4 1)]
```

There are several representations of graphs that are possible and to this end, it is useful to work with a common interface for graphs, thus allowing experimentation with different implementations.

7 *Shortest Paths*

Players start with bases at some given distance away from each other. Knowing the distance to every planet helps to estimate how long it will be before there can possibly be any conflict there.

Working out the shortest distance through the galaxy can be reduced to an instance of Dijkstra's algorithm. The version explored here finds a list of the shortest paths from a root vertex to each other reachable vertex in the graph.

There are three main datastructures at the heart of Dijkstra's algorithm: a graph, a priority queue, and a set. The graph is required to pull out edges for consideration when constructing the shortest path. The priority queue holds tentative shortest paths to vertices neighbouring the targets of shortest paths that have already been found. The set holds vertices that have yet to be explored.

Before working with Dijkstra's algorithm, it will be useful to understand priority queues.

7.1 *Priority queues*

A priority queue is a structure that contains elements which have a given priority. Elements are extracted from the priority queue in ascending order.

The ordering imposed by a priority queue is given by a function which compares two elements and returns an **`Ordering`**. The values of type **`Ordering`** are either **`LT`** (less than), **`EQ`** (equal), or **`GT`** (greater than). Here are some functions that turn a binary operation that returns an ordering into a function that corresponds to the more traditional `(<=)` and `(==)`.

```
lte :: (a -> a -> Ordering) -> (a -> a -> Bool)
lte cmp x y = cmp x y /= GT

eq :: (a -> a -> Ordering) -> (a -> a -> Bool)
eq cmp x y = cmp x y == EQ
```

A useful intuition is that a priority queue is abstractly simply an ordered list. Element order must be maintained when elements are added or removed from this list. The `PQueue` interface allows for different implementations of a priority queue:

```
class PQueue pqueue where
  toPQueue   :: (a -> a -> Ordering) -> [a] -> pqueue a
  fromPQueue :: pqueue a -> [a]

  priority :: pqueue a -> (a -> a -> Ordering)

  empty :: (a -> a -> Ordering) -> pqueue a
  isEmpty :: pqueue a -> Bool

  insert :: a -> pqueue a -> pqueue a
  delete :: a -> pqueue a -> pqueue a

  extract :: pqueue a -> a
  discard :: pqueue a -> pqueue a
  detach  :: pqueue a -> (a, pqueue a)
```

This intuition can be firmed up by giving laws that govern a valid priority queue instance:

These laws appeal to operations on lists, which can be found at `https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html`.

$$\texttt{fromPQueue (toPQueue cmp xs)} = \texttt{sortBy cmp xs} \tag{1}$$

$$\texttt{fromPQueue (empty cmp)} = \texttt{[]} \tag{2}$$

$$\texttt{isEmpty xs} = \texttt{nil (fromPQueue xs)} \tag{3}$$

$$\texttt{fromPQueue (insert x xs)} = \texttt{insertBy (eq (priority xs)) (fromPQueue xs)} \tag{4}$$

$$\texttt{fromPQueue (delete x xs)} = \texttt{deleteby (eq (priority xs)) (fromPQueue xs)} \tag{5}$$

$$\texttt{extract xs} = \texttt{head (sortBy (priority xs) (fromPQueue xs))} \tag{6}$$

$$\texttt{discard xs} = \texttt{tail (sortBy (priority xs) (fromPQueue xs))} \tag{7}$$

$$\texttt{detach cs} = \texttt{(extract xs, discard xs)} \tag{8}$$

Notice that `toPQueue . fromPQueue` is not necessarily the identity function: there is space for different underlying representations of the same sorted list.

Perhaps the most natural implementation of a priority queue is a sorted list, together with the function that will determine the ordering of elements:

```
data PList a = PList (a -> a -> Ordering) [a]
```

To witness that this is indeed a priority queue requires a `PList` instance of `PQueue`:

```haskell
instance PQueue PList where

  toPQueue cmp xs = PList cmp (sortBy cmp xs)

  fromPQueue (PList _ xs) = xs

  empty cmp = PList cmp []

  isEmpty (PList _ xs) = null xs

  priority (PList cmp _) = cmp

  insert x (PList cmp []) = PList cmp [x]
  insert x ps@(PList cmp xs)
    | x <= y    = cons x ps
    | otherwise = cons y (insert x ys)
    where (<=) = lte cmp
          (y, ys) = detach ps
          cons x (PList cmp xs) = PList cmp (x:xs)

  delete x (PList cmp []) = PList cmp []
  delete x ps@(PList cmp _)
    | x == y    = ys
    | otherwise = cons y (delete x ys)
    where (==) = eq cmp
          (y, ys) = detach ps
          cons x (PList cmp xs) = PList cmp (x:xs)

  extract (PList cmp (x:xs)) = x

  discard (PList cmp (x:xs)) = PList cmp xs

  detach  (PList cmp (x:xs)) = (x, PList cmp xs)
```

In the implementation of Dijkstra's shortest path algorithm, the priority queue holds paths, ordered by their total length. This ordering is implemented by the `cmpPath` function.

```haskell
cmpPath :: Path v -> Path v -> Ordering
cmpPath (Path d _) (Path d' _) = compare d d'
```

7.2 *Dijkstra's Algorithm*

The purpose of Dijkstra's algorithm is to find shortest paths. Rather than finding the shortest path from a source vertex to a particular target, the algorithm presented below finds shortest paths from a source vertex to all reachable vertices in the graph.

The `shortestPaths` function operates on a given graph `g` and source vertex `v` to return a list of the shortest paths from `v`.

```
shortestPaths :: forall g e v. Graph g e v => g -> v -> [Path e]
shortestPaths g v = dijkstra g (vertices g \\ [v]) ps
 where
  ps :: PList (Path e)
  ps = foldr insert (empty cmpPath) (map pathFromEdge (edgesFrom g v))
```

This calls the `dijkstra` function where the graph is `g`, the list containing all unvisited vertices has all the vertices in `g` except `v`, and the initial candidates for the shortest paths put into the priority queue are all the edges from `v`, represented as paths.

The function `dijkstra` does most of the hard work. Here it is in full, followed by a description of how it operates.

Once again, the `dijkstra` function is highly polymorphic, and it works against any graph and priority queue implementation.

```
dijkstra :: (Graph g e v, PQueue pqueue) =>
  g -> [v] -> pqueue (Path e) -> [Path e]
dijkstra g [] ps = []
dijkstra g us ps
  | isEmpty ps  = []
  | v `elem` us = p : dijkstra g (us \\ [v])
                                 (foldr insert ps' (map (extend p) (edgesFrom g v)))
  | otherwise  = dijkstra g us ps'
  where
    (p, ps') = detach ps
    v = target p
```

If the list of visited nodes or the priority queue of candidate edges is empty, then the algorithm is finished and returns the empty list.

Otherwise, the algorithm selects the minimum path `p` from the priority queue `ps`. In the simple case, the target vertex `v` of the path `p` has already been visited, so a shortest path has already been found to that node. The algorithm therefore proceeds with the remaining priority queue that does not include `p`.

If the target vertex `v` of the path `p` is in the list of unvisited vertices in `us`, then the path `p` is added to the list of solutions: this is the shortest path to `v`. The remaining solutions are then found by recursively calling the algorithm with an updated set of unvisited nodes and an updated priority queue. The set of unvisited nodes is updated to remove `v`. Since `p` is a shortest path, new candidate shortest paths are those that extend `p` by the edges from `v`.

The shortest path between any two planets in `example1` can be computed by simply calling `shortestPaths`, since `GameState` is a graph.

```
ghci> shortestPaths example1 0
[ Path 1 [(0,Wormhole 0 1 1)]
, Path 1 [(1,Wormhole 0 2 1)]
, Path 1 [(2,Wormhole 0 3 1)]
, Path 1 [(3,Wormhole 0 4 1)] ]
```

This is a rather uninformative example all of the planets are connected to the source and not to each other.

A more useful try is with `example2`, and this also works with `shortestPaths` because lists of edges are graphs:

```
ghci> shortestPaths example2 "s"
[ Path 5 [("s","y",5)]
, Path 7 [("y","z",2),("s","y",5)]
, Path 8 [("y","t",3),("s","y",5)]
, Path 9 [("t","x",1),("y","t",3),("s","y",5)] ]
```

Although the implementation here is correct, one problem with the code is that it is very inefficient. This is largely due to the incorrect choice of datastructures: the graph representation, the priority queue of paths, and the set of unvisited vertices all use basic structures that have suboptimal complexities.

Problem 7: *Dijkstra Dualized*
Describe in words or otherwise, what the necessary modifications are so that the `dijkstra` algorithm above finds the path *to* the root vertex `v`, rather than *from* the root?

7.3 *A Heap of Paths*

The first optimisation will be to replace the `PList` datatype with a *binary heap*. At its simplest, this is a balanced binary tree that maintains the minimum element at the root of the tree. When a new element is inserted, its correct location is found and the tree is rebalanced. Similarly, when an element is removed the remaining tree must be rebalanced.

The datastructure for a heap is given as follows:

```
data Heap a = Heap (a -> a -> Ordering) (Tree a)
data Tree a = Nil | Node Int (Tree a) a (Tree a)
```

The idea is that a node `Node h l x r` maintains its height in the variable `h`, and that the element `x` is the smallest among those in the subtrees `l` and `r`.

In order to maintain this invariant the tree will have to be rotated appropriately when elements are inserted or deleted.

Problem 8: *Heap Operations*

Provide the instance of `PQueue` that shows how a heap can be used as a priority queue, by filling out the following definitions:

```
instance PQueue Heap where
  toPQueue = undefined
  fromPQueue = undefined

  priority :: Heap a -> (a -> a -> Ordering)
  priority = undefined

  empty :: (a -> a -> Ordering) -> Heap a
  empty p = undefined

  isEmpty :: Heap a -> Bool
  isEmpty = undefined

  insert :: a -> Heap a -> Heap a
  insert = undefined

  delete :: a -> Heap a -> Heap a
  delete = undefined

  extract :: Heap a -> a
  extract = undefined

  discard :: Heap a -> Heap a
  discard = undefined

  detach :: Heap a -> (a, Heap a)
  detach = undefined
```

Provide a description of the complexities of the operations.

The definition of `shortestPaths'` below uses a `Heap` instead of a `PList` for the underlying priority queue representation.

```
shortestPaths' :: forall g e v . Graph g e v => g -> v -> [Path e]
shortestPaths' g v = dijkstra g (vertices g) ps
 where
  ps :: Heap (Path e)
  ps = foldr insert (empty cmpPath) (map pathFromEdge (edgesFrom g v))
```

This is exactly the same code as for `shortestPaths`, except that the type annotation for `ps` specifies that it should use a `Heap`.

7.4 *Adjacency List Graph*

The graph representation used so far has been the simple `GameState` instance. The problem here is that it is not very efficient for the `dijkstra` algorithm. There, the `edgesFrom` function is used to find all the edges from a vertex. This operation happens each time a shortest path has been found. In the instance of `Graph` for `GameState`, the implementation filters through the list of all the wormholes.

A different representation of a graph is known as the *adjacency list*. An adjacency list is a list of pairs containing each vertex paired with all edges from that vertex. The type can be given by `AdjList`:

```
newtype AdjList e v = AdjList [(v, [e])]
```

In other words, each pair `(v, es)` in the adjacency list is a vertex `v`, and each edge `e` in the edges `es` is such that `source e = v`.

Problem 9: *Adjacency List Graphs*

Implement an adjacency list representation that supports efficient lookup of the edges from a planet by filling out the following interface. Discuss the complexity of these operations.

```
instance (Eq e, Edge e v) => Graph (AdjList e v) e v where
  vertices (AdjList ves)    = undefined
  edges (AdjList ves)       = undefined
  edgesFrom (AdjList ves) s = undefined
  edgesTo   (AdjList ves) t = undefined
  velem v (AdjList ves)     = undefined
  eelem e (AdjList ves)     = undefined
```

Hint: You may use the standard list operations given in `Data.List`. You may alternatively find list comprehensions useful.

7.5 *Conflict Zones*

Having established an algorithm for finding the shortest paths from a root vertex, it is possible to calculate the planets which each player can reach first. This can be achieved by running the shortest path algorithm from each player home base. As a final task, you should write an algorithm that calculates this for a given `GameState`.

Problem 10: *Conflict Zones*

Provide the definition of `conflictZones`, where `conflictZones st p q` takes in the game state `st`, and two planet IDs `p` and `q`, and returns a triple `(ps, pqs, qs)` where `ps` are the identities of planets that can be reached by `p` first, `qs` are the identities of planets that can be reached by `q` first, and `pqs` are the identities of planets that can be reached by both at the same time.

```
conflictZones :: GameState -> PlanetId -> PlanetId
```

```
  -> ([PlanetId], [PlanetId], [PlanetId])
conflictZones g p q = undefined
```

Note that this should only be a function of the topology of the game: it does not need to take into account the number of ships garrisoned on planets, whether conquering is needed, the positions of fleets that are in flight or any other details.