*Your submission for this assignment* ***must include your full name*** *(as it appears on cuLearn) and you nine-digit* ***student number as a comment*** *at the top of* ***every source file you submit****.*

*Your submission for question 1 is not code and must be a* ***single pdf file*** *with a* ***file name of 'comp3007_f18_#########_a4_1.pdf'*** *(with the number signs replaced by your nine-digit student number). It* ***must be written*** *using* ***Microsoft Word****,* ***Google Docs****, or* ***LaTeX****.* ***Photographs or scans*** *of handwritten submissions* ***will not be accepted****.*

*Your submissions for questions 2 and 3 will include both Haskell and Prolog and must use the filenames* ***'comp3007_f18_#########_a4_2a.hs', 'comp3007_f18_#########_a4_2b.pl',*** *and* ***'comp3007_f18_#########_a4_3.pl'****.*

***Compress your submission*** *into a zip archive named* ***'comp3007_f18_#########_a4.zip'***

***Late assignments will not be accepted*** *and will* ***receive a mark of 0****.*

***Submissions that crash*** *(i.e., terminate with an error) on execution will* ***receive a mark of 0****.*

***The due date for this assignment is Saturday, December 1, 2018, by 11:00pm.***

## Question 1: "Structural Induction"

For this question, you must consider the following definition and functions for the NTree data type (below) and prove (using structural induction) that for every binary tree Z:

$$\texttt{count Z} \quad \leq \quad 2^{\texttt{(height Z)}} - 1$$

Remember that, if you are trying to prove that b ≤ d, and you know that a ≤ b ≤ c, you can replace b with c (to get c ≤ d) but you cannot replace b with a (to get a ≤ d). To receive full marks for this question you must show every step in your proof and you must use the source code line labels (shown below, in red) whenever you use equational reasoning.

```
data NTree = NilT | Node Int NTree NTree

[ctb]   count NilT = 0
[ctr]   count (Node n x y) = (count x) + (count y) + 1

[htb]   height NilT = 0
[htr]   height (Node n x y) = (max (height x) (height y)) + 1

        max a b
[mc1]         | a >= b = a
[mc2]         | otherwise = b
```

Proving the "Base Case"

Inequality to be Proven $\quad$ count NilT $\leq 2^{(\text{height NilT})} - 1$

LHS: count NilT
= 0 $\quad$ by [ctb]

RHS: $2^{(\text{height NilT})} - 1$
= $2^0 - 1$ $\quad$ by [htb]
= 1 - 1
= 0

Proving the "Inductive Case":

$$(\text{count L}) \leq 2^{(\text{height L})} - 1 \wedge (\text{count R}) \leq 2^{(\text{height R})} - 1 \rightarrow (\text{count (Node n L R)}) \leq 2^{(\text{height (Node n x y)})} - 1$$

...using a Direct Proof

1. $(\text{count L}) \leq 2^{(\text{height L})} - 1 \wedge (\text{count R}) \leq 2^{(\text{height R})} - 1$ $\quad$ by Assumption
2. $(\text{count L}) \leq 2^{(\text{height L})} - 1$ $\quad$ by Simplification 1
3. $(\text{count R}) \leq 2^{(\text{height R})} - 1$ $\quad$ by Simplification 1

Inequality to be Proven $\quad$ count (Node n L R)) $\leq 2^{(\text{height (Node n x y)})} - 1$

LHS: count (Node n L R))
= (count L) + (count R) + 1 $\quad$ by [ctr]
$\leq (2^{(\text{height L})} - 1) + (2^{(\text{height R})} - 1) + 1$ $\quad$ by Inductive Assumption
= $2^{(\text{height L})} + 2^{(\text{height R})} - 1$

RHS: $2^{(\text{height (Node n L R))})} - 1$
= $2^{((\max (\text{height L}) (\text{height R})) + 1)} - 1$ $\quad$ by [htr]

Case 1: $\quad$ (height L) ≥ (height R)

LHS: $2^{(\text{height L})} + 2^{(\text{height R})} - 1$
$\leq 2^{(\text{height L})} + 2^{(\text{height L})} - 1$ $\quad$ by [mc1]
= $2\ (2^{(\text{height L})}) - 1$
= $2^{(\text{height L}) + 1} - 1$

RHS: $2^{((\max (\text{height L}) (\text{height R})) + 1)} - 1$
= $2^{(\text{height L}) + 1)} - 1$ $\quad$ by [mc1]

Case 2: $\quad$ (height L) < (height R)

LHS: $2^{(\text{height L})} + 2^{(\text{height R})} - 1$
$\leq 2^{(\text{height R})} + 2^{(\text{height R})} - 1$ $\quad$ by [mc2]
= $2\ (2^{(\text{height R})}) - 1$
= $2^{(\text{height R}) + 1} - 1$

RHS: $2^{((\max (\text{height L}) (\text{height R})) + 1)} - 1$ $\quad$ by [mc2]
= $2^{(\text{height R}) + 1)} - 1$

## Question 2: "Count Elements from Range"

A function to count the number of elements in a list of integers that are within some range (inclusively) is straightforward in either Haskell or Prolog. In Haskell, it would have three parameters (the list of integers and the lower and upper bound of the) and would return a single value (the number of elements within that range). In Prolog, it would be a predicate of arity four that succeeds when the fourth element is the number of elements in the first list argument that are in the range specified by the second and third arguments (and you could leave the fourth element uninstantiated to have Prolog count it for you).

```
HASKELL > countInRange [2,4,6,8,10,12,14,16,18] 3 11
4

PROLOG ?- countInRange([2,4,6,8,10,12,14,16,18], 3, 11, X).
X = 4 ;
false.
```

For this question, you will write this function twice - once using Haskell and once using Prolog. Both solutions must be recursive, the Haskell solution must be tail-call optimized, and neither solution may use higher order functions.

```
-- this version is not tail-call optimized and is included in the model solution only for your reference
countInRange :: [Int] -> Int -> Int -> Int
countInRange [] _ _ = 0
countInRange (h:t) lower upper
  | h >= lower && h <= upper = 1 + countInRange t lower upper
  | otherwise = countInRange t lower upper

countInRangeHelper :: [Int] -> Int -> Int -> Int
countInRangeHelper (h:t) lower upper = countInRangeOptimized (h:t) lower upper 0

countInRangeOptimized :: [Int] -> Int -> Int -> Int -> Int
countInRangeOptimized [] _ _ accum = accum
countInRangeOptimized (h:t) lower upper accum
  | h >= lower && h <= upper = countInRange t lower upper (accum + 1)
  | otherwise = countInRange t lower upper accum



countInRange([], _, _, 0).
countInRange([H|T], L, U, X) :- H >= L, H =< U, countInRange(T, L, U, Y), X is Y+1.
countInRange([H|T], L, U, X) :- (H < L; H > U), countInRange(T, L, U, X).
```

## Question 3: "You SHALL Pass"

For this question, you must write a Prolog program that will solve the following logic puzzle by modeling it as a finite state machine. Your solution to this question must answer True or False for the first question (i.e., Is it possible...?) and it must provide an unambiguous description of the action plan for the second question (i.e., How?).

*While traveling through Moria, Gandalf, Aragorn, Gilmi, and Legolas approach a great chasm. There is a bridge going across but it can only hold two people at a time. Because of poor planning, they only have a single torch between the four of them, and since the mines are particularly dark, the torch must be carried whenever someone tries to cross the bridge.*

*Legolas is the fastest and can cross the bridge in only one minute, and Aragorn is in pretty good shape too so he can cross it in three minutes. Gimli, being a dwarf, has stubby legs, so it will take him five minutes, and since Gandalf is over 2000 years old, it will take him eight minutes to cross the bridge. Whenever two of the company cross the bridge together, they must stick together so that they both know where they are going. This entails that they move at the pace of the slower of the two.*

*Is it possible for all of them to cross the bridge in 15 minutes or less? How?*

Possible Solution #1

```
% A few simple functions for making life easier - obviously you don't need to implement
these as separate rules but it makes the code easier to read
member(X, [X|_]).
member(X, [H|T]) :- H \= X, member(X, T).

subListOf([], _).
subListOf([H|T], [H|U]) :- subListOf(T, U).
subListOf([H|T], [_|U]) :- subListOf([H|T], U).

permuteOf([], []).
permuteOf([A|B], C) :- permuteOf(B, D), removeFrom(A, C, D).

removeFrom(A, [A|B], B).
removeFrom(A, [B|C], [B|D]) :- removeFrom(A, C, D).

removeAll([], X, X).
removeAll([H|T], X, Z) :- removeFrom(H, X, Y), removeAll(T, Y, Z).
```

```
% this is used to keep the party on either side in sorted order, to simplify certain operations...
sortParty(Unsorted, Sorted) :- permuteOf(Unsorted, Sorted), subListOf(Sorted, [legolas, aragorn, gimli, gandalf]).

% ...such as this one - since the party is always sorted, the cost associated with a list of length 2 is just the cost of the 2nd element
costParty([X], Cost) :- timeRequired(X, Cost).
costParty([_, Y], CostY) :- timeRequired(Y, CostY).

% these are just the facts for the puzzle
timeRequired(legolas, 1).
timeRequired(aragorn, 3).
timeRequired(gimli, 5).
timeRequired(gandalf, 8).

% here is a generic transition for moving a party (of either 1 or 2 people) moving from east to west
crossTheBridge(state(East1, West1, east), Who, Cost, state(East2, West2, west) ) :- subListOf(Who, East1), length(Who, Size), Size > 0, Size =< 2, costParty(Who, Cost), removeAll(Who, East1, East2), append(Who, West1, West2Unsorted), sortParty(West2Unsorted, West2).

% ... and here is a generic transition for moving a party from west to east
crossTheBridge(state(East1, West1, west), Who, Cost, state(East2, West2, east) ) :- subListOf(Who, West1), length(Who, Size), Size > 0, Size =< 2, costParty(Who, Cost), removeAll(Who, West1, West2), append(Who, East1, East2Unsorted), sortParty(East2Unsorted, East2).

% this is the graph search rule (nearly identical to the one used for cannibals and missionaries, etc.)
theMeaningOfHaste(End, End, Time, []) :- Time >= 0.
theMeaningOfHaste(Start, End, Time, [Action | Rest]) :- Start \= End, crossTheBridge(Start, Action, Cost, Next), Diff is Time - Cost, Diff >= 0, theMeaningOfHaste(Next, End, Diff, Rest).

% with a 15-minute time limit, there is no solution
% the query would be ?- theMeaningOfHaste(state([legolas, aragorn, gimli, gandalf], [], east), state([], [legolas, aragorn, gimli, gandalf], _), 15, How).

% with an 18-minute time limit, there are many solutions (albeit they are mostly minor variations on two very different approaches)
% the query would be ?- theMeaningOfHaste(state([legolas, aragorn, gimli, gandalf], [], east), state([], [legolas, aragorn, gimli, gandalf], _), 18, How).
```

Possible Solution #2

```
% A few simple functions for making life easier, repeated here
subListOf([], _).
subListOf([H|T], [H|U]) :- subListOf(T, U).
subListOf([H|T], [_|U]) :- subListOf([H|T], U).

% This tests that only 1 or 2 try to cross at a given time
noMoreThan2([_]).
noMoreThan2([_, _]).

% This is where the actual crossing of the bridge takes place
flyYouFools(east, west).
flyYouFools(west, east).

% This actually generates the different possible groups that could cross
% Notice that no duplicate teams are created; if [x, y] appears then [y, x] will not
chooseTheParty(Party) :- noMoreThan2(Party), subListOf(Party, [legolas, aragorn,
gimli, gandalf]).

% These are the different possible groups that might attempt to traverse the bridge;
the time necessary is also indicated
crossTheBridge( state(T, T, G, W, T), [legolas, aragorn], 3, state(X, X, G, W, X) )
:- flyYouFools(T, X).
crossTheBridge( state(T, A, T, W, T), [legolas, gimli]  , 5, state(X, A, X, W, X) )
:- flyYouFools(T, X).
crossTheBridge( state(T, A, G, T, T), [legolas, gandalf], 8, state(X, A, G, X, X) )
:- flyYouFools(T, X).
crossTheBridge( state(L, T, T, W, T), [aragorn, gimli]  , 5, state(L, X, X, W, X) )
:- flyYouFools(T, X).
crossTheBridge( state(L, T, G, T, T), [aragorn, gandalf], 8, state(L, X, G, X, X) )
:- flyYouFools(T, X).
crossTheBridge( state(L, A, T, T, T), [gimli,   gandalf], 8, state(L, A, X, X, X) )
:- flyYouFools(T, X).
crossTheBridge( state(T, A, G, W, T), [legolas], 1, state(X, A, G, W, X) ) :-
flyYouFools(T, X).
crossTheBridge( state(L, T, G, W, T), [aragorn], 3, state(L, X, G, W, X) ) :-
flyYouFools(T, X).
crossTheBridge( state(L, A, T, W, T), [gimli]  , 5, state(L, A, X, W, X) ) :-
flyYouFools(T, X).
crossTheBridge( state(L, A, G, T, T), [gandalf], 8, state(L, A, G, X, X) ) :-
flyYouFools(T, X).
```

```
% This is the (fairly generic) solver - it checks that the time is =< 15 and then tries an action...
solveTheProblem( state(west, west, west, west, west), [], Time) :- Time =< 18.
solveTheProblem( CurrentState, [Action | Plan], Time) :- Time =< 18, chooseTheParty(Action), crossTheBridge( CurrentState, Action, Cost, NewState ), NewTime is Time + Cost, solveTheProblem(NewState, Plan, NewTime).

% I just added this so you can test without providing anything but a start state.
% Use "?- solveTheProblem( state(east, east, east, east, east) )."
solveTheProblem( StartState ) :- solveTheProblem(StartState, Solution, 0), write(Solution).
```