Helpers:

- **in(X,L) –** Checks if X is an element of list L. Recurses through L until the head is equal to X, showing that there is an occurrence of X.
- **first(L,X) –** Gets first element of list by getting the head.
- **len(L,X) –** Gets length X of list L, works by recursing through L and adding one for each element until list is empty.
- **noMatches(X,Y) –** Takes two lists X and Y, and returns true if there are no shared elements. Works by recursing through X, and checking if its head is in Y, failing if so.
- **concat(X,Y,Z) –** Takes lists X and Y and concatenates them to form Z. Works by recursing through X and Z, checking their heads are the same. The base case is X being an empty list and Y and the tail of Z matching, resulting in Y being added to Z.

Q1:

- **multiple_lines(S) –** Checks S is a station and then checks that it is a stop on two different lines.

Q2:

- **termini(L,S1,S2) –** Gets start station S1 and end station S2 of line L. Works by taking two stations S1 and S2, checks S1 is number 1 on its line as this is always the start and check that the number of S2 does not evaluate true on not_end_stop, showing it is a maximum.
- **not_end_stop(L,SN) –** Evaluates true if stop number SN is lower than another stop number on line L. Takes two stops with shared line, one with number SN the other with number S2. If S2 is higher than SN, it evaluates true.

Q3:

- **get_stops(L,List,I) –** Gets the stops for list_stops. Works by taking line L, List and depth I. Works by recursing through List checking that List's head's station number is 1 less than the head of the tail of list. This recurses, increasing I by 1, until I is equal to the length of the line.
- **lineLength(L,Length) –** Takes a stop on line L and gives the length of it by getting the maximum stop with not_end_stop from Q2.
- **List_stops(L,List) –** Calls get_stops with initial I of 1.

Q4:

- **segmentTime(L,S1,S2,T) –** Represents a traversal of stations on one line. Takes line L, start station S1, end station S2 and time T. Works by increasing station number by 1 for each recursion until base case of S1 being immediately before S2. T is calculated by summing time from ttime recursively.
- **stationsBetween(L,S1,S2,List) –** Gets list of stations (List) between start S1 and end S2 on Line L, including both ends. Works by recursing through List, taking a stop after the current one and calling that as the new S1. Recursion stops when S2 is equivalent to the next station (end reached).
- **get_path(S1,S2,Path,Visited,PreL) –** Gets the path list for path(). Takes start station S1, end station S2, list of segmentTime Visited, list of stations Visited and line PreL representing the last line taken. Works by recursing through path, checking the head is a segmentTime and evaluating its T value. The two stops of the segment must be in order, S2 having a higher station number than S1. The segment must not contain previously visited nodes, so the stations between the start and penultimate station of a segment are checked against the visited stations, failing if there is a match. The line of the segment must not be the same as PreL or collapsible segment lists would occur. The stations visited (except the last stop) are added to Visited and get_path is called again with the new S1, visited nodes and visited line. Recursion continues until S2 is the end station of a segment.
- **Path(S1,S2,L) –** Shorthand for calling get_path with an empty list for visited nodes and a non-existent starting line.

Q5:

- **not_easiest_path(S1,S2,Path) –** Evaluates true if a path exists between S1 and S2 that is of lower length (fewer segments) than the passed in Path. If so, it is not the easiest path.
- **easiest_path(S1,S2,Path) –** Checks if a path is the easiest path by checking Path is a path between S1 and S2 and that Path is not of greater length than other paths.

Q6:

- **stationTotal(Path,Scount) –** Gets the number of stations in a path by summing the stations between S1 and S2 in a segment, recursively adding for each entry in the list. Last stops of a segment are not counted except for the last one as they would repeat otherwise.
- **not_shortest_path(S1,S2,Path) –** Same as easiest_path, but with stationTotal instead of len.
- **shortest_path(S1,S2,Path) –** Same as easiestPath, but with not_shortest_path.

Q7:

- **timeTotal(Path,Ttotal) –** Gets the total amount of time in a path by recursing through Path and returning the sum of a segment's time added to the next call of timeTotal.
- **not_fastest_path(S1,S2,Path) –** Same as easiest_path, but with timeTotal.
- **fastest_path(S1,S2,Path) –** Same as easiestPath, but with not_fastest_path.

Code Listing:

```
% station appears on two different stops where the line is different
multiple_lines(S) :- station(S),stop(X,_,S),stop(Y,_,S),X\=Y.



% s1 and s2 are stations, s1 is a start stop (1), s2 is an end stop (max Y)
termini(L,S1,S2) :- station(S1),station(S2),
                    stop(L,1,S1),stop(L,Y,S2),\+ not_end_stop(L,Y).

% checks if not max station for that line to allow calculation of max (end of a line)
not_end_stop(L,SN) :- stop(L,SN,_),stop(L,S2,_),S2>SN.



% base case, has number of recursed relations is line length, tail is blank
get_stops(L,[_|T],I) :- lineLength(L,I), T = [].
% next stop in list is next stop on line (L), recurses until line end
get_stops(L,[X|T],I) :- stop(L,N1,X),stop(L,N2,Y), first(T,Y),
                        N2 is N1+1,I2 is I+1, get_stops(L,T,I2).

% shorthand func
list_stops(L,List) :- get_stops(L,List,1).

% gets length of a line by grabbing the num of the end stop
lineLength(L,Length) :- stop(L,Length,_), \+ not_end_stop(L,Length).



% base case, S1 and S2 are next to eachother
segmentTime(L,S1,S2,T) :- stop(L,N1,S1),stop(L,N2,S2), N1 is N2-1, ttime(L,N1,N2,T).
% gets next station and distance, recursively adds to next station's times
segmentTime(L,S1,S2,T) :- stop(L,N1,S1),Nnext is N1+1, stop(L,Nnext,Snext),
ttime(L,N1,Nnext,TCur),
                          segmentTime(L,Snext,S2,Tnext), T is Tnext+TCur.


% base case, if final station is same as end station of segment
% and remaining segments empty
% segmentTime is repeated as otherwise T is a singleton and isn't evaluated
get_path(S1,S2,[Seg|Path],Visited,PreL) :- Seg = segmentTime(L,S1,S2,T),
                                           segmentTime(L,S1,S2,T), Path =[],
                                           stop(L,N1,S1),stop(L,N2,S2),N2>N1,
                                           stationsBetween(L,S1,S2,New),noMatches(Visited,New),
```

```
                                        PreL \= L.

% finds a segment on not previously visited stations,
%forbids line repeats, recurses with new visits
get_path(S1,S2,[Seg|Path],Visited,PreL) :- Seg = segmentTime(L,S1,Snext,T),
                                        segmentTime(L,S1,Snext,T),
                                        stop(L,N1,S1),stop(L,N2,Snext),N2>N1,
                                        stop(L,Nlast,Slast), Nlast is N2-1,
                                        stationsBetween(L,S1,Slast,New),noMatches(Visited,New),
                                        PreL \= L,
                                        concat(Visited,New,NewVis),
                                        get_path(Snext,S2,Path,NewVis,L).

% shorthand for get_path with empty visits and non-existent starting line
path(S1,S2,Path) :- get_path(S1,S2,Path,[],hs2).

% gets stations between S1,S2 (inclusive) on a line as a list
%lines only go forward so +1 works
stationsBetween(L,S1,X,[X|List]) :- stop(L,N,S1),stop(L,N,X),List=[].
stationsBetween(L,X,S2,[X|List]) :- stop(L,N1,X),Ni is N1+1, stop(L,Ni,Si),
                                    stationsBetween(L,Si,S2,List).

% gets easiest path by checking it isn't of greater length than others
easiest_path(S1,S2,Path) :- path(S1,S2,Path), \+ not_easiest_path(S1,S2,Path).
not_easiest_path(S1,S2,Path) :- path(S1,S2,Path), path(S1,S2,Path2),
                                len(Path,L1), len(Path2,L2), L1>L2.

% gets shortest path by checking it doesn't have more stations than others
shortest_path(S1,S2,Path) :- path(S1,S2,Path), \+ not_shortest_path(S1,S2,Path).
not_shortest_path(S1,S2,Path) :- path(S1,S2,Path), path(S1,S2,Path2),
                                 stationTotal(Path,T1),stationTotal(Path2,T2), T1>T2.

% gets the total number of stations on a path by summing stations between segment ends
% end nodes aren't counted except for the very last one, as otherwise they'd be repeated
stationTotal([],1).
stationTotal([Seg|Path],Scount) :- Seg = segmentTime(L,S1,S2,_), stationsBetween(L,S1,S2,Sts),
                                   len(Sts,SL), stationTotal(Path,Scountn),
                                   Scount is (SL-1)+Scountn.

% gets the fastest path by checking it doesn't have a greater time than others
fastest_path(S1,S2,Path) :- path(S1,S2,Path), \+ not_fastest_path(S1,S2,Path).
not_fastest_path(S1,S2,Path) :- path(S1,S2,Path), path(S1,S2,Path2),
                                timeTotal(Path,T1),timeTotal(Path2,T2), T1>T2.

% gets the total time in a path by adding time of each segment recursively
timeTotal([],0).
timeTotal([Seg|Path],Ttotal) :- Seg = segmentTime(L,S1,S2,T), segmentTime(L,S1,S2,T),
                                timeTotal(Path,Tn),Ttotal is T+Tn.


% ----------- HELPERS ------------------

% returns true if no element in X is in Y
noMatches([],_).
noMatches([X|XS],Y) :- \+ in(X,Y), noMatches(XS,Y).

% gets the first element of the list
first([Fst|_],Fst).

% concatenates two lists
concat([],L,L).
concat([X|XS],YS,[X|ZS]) :- concat(XS,YS,ZS).

% checks if element occurs in list by matching head
in(X,[X|_]).
in(X,[_|XS]) :- in(X,XS).

% gets length of a list by adding 1 until empty
len([],0).
len([_|XS],L) :- len(XS,Ln), L is Ln+1.
```