Part 1:

```
% knowledge base
flight(edirne,edremit).

flight(edremit,edirne).
flight(edremit,erzincan).

flight(erzincan,edremit).

flight(istanbul,izmir).
flight(istanbul,antalya).
flight(istanbul,gaziantep).
flight(istanbul,ankara).
flight(istanbul,van).
flight(istanbul,rize).

flight(antalya,istanbul).
flight(antalya,konya).
flight(antalya,gaziantep).

flight(burdur,isparta).

flight(isparta,burdur).
flight(isparta,izmir).

flight(izmir,isparta).
flight(izmir,istanbul).

flight(gaziantep,istanbul).
flight(gaziantep,antalya).

flight(konya,antalya).
flight(konya,ankara).

flight(rize,van).
flight(rize,istanbul).

flight(van,ankara).
flight(van,istanbul).
flight(van,rize).

flight(ankara,konya).
flight(ankara,istanbul).
flight(ankara,van).
```

All flights as facts.

```
%rules
route(A, B) :- visit(A, B, []).
visit(A, B, Visited) :- flight(A, C), not(member(C, Visited)), (B = C; visit(C, B, [A|Visited])).
```

Checks is there a flight from A to C and if there is, then it add to Visited, and for each step it checks C is in visited or not to prevent infinite loops

```
?- route(istanbul,izmir)
|    .
true .

?- route(istanbul,rize).
true .

?- route(edirne, istanbul).
false.

?- 
```

Tests.

Part 2:

```
% knowledge base
flight(edirne,edremit).

flight(edremit,edirne).
flight(edremit,erzincan).

flight(erzincan,edremit).

flight(istanbul,izmir).
flight(istanbul,antalya).
flight(istanbul,gaziantep).
flight(istanbul,ankara).
flight(istanbul,van).
flight(istanbul,rize).

flight(antalya,istanbul).
flight(antalya,konya).
flight(antalya,gaziantep).

flight(burdur,isparta).

flight(isparta,burdur).
flight(isparta,izmir).

flight(izmir,isparta).
flight(izmir,istanbul).

flight(gaziantep,istanbul).
flight(gaziantep,antalya).

flight(konya,antalya).
flight(konya,ankara).

flight(rize,van).
flight(rize,istanbul).

flight(van,ankara).
flight(van,istanbul).
flight(van,rize).

flight(ankara,konya).
flight(ankara,istanbul).
flight(ankara,van).

% knowledge base
distance(edirne,edremit,914.67).

distance(edremit,edirne,914.67).
distance(edremit,erzincan,736.34).

distance(erzincan,edremit,736.34).

distance(istanbul,izmir,328.80).
distance(istanbul,antalya,482.75).
distance(istanbul,gaziantep,847.42).
distance(istanbul,ankara,351.50).
distance(istanbul,van,1262.37).
distance(istanbul,rize,967.79).

distance(antalya,istanbul,482.75).
distance(antalya,konya,192.28).
distance(antalya,gaziantep,592.33).

distance(burdur,isparta,24.60).

distance(isparta,burdur,24.60).
distance(isparta,izmir,308.55).

distance(izmir,isparta,308.55).
distance(izmir,istanbul,328.80).
```

All flights and distances as facts.

```
sroute(X, Y, N) :-
    distance(X, Y, N).
sroute(X, Z, N) :-
    distance(X, Y, N0),
    sroute(Y, Z, N1),
    N is N0 + N1.
```

Finds the distance between 2 cities as minimum.

```
?- sroute(istanbul, burdur, X).
X = 661.95
```

Test.

Part 3:

```
%facts

enrollment(a,102).
enrollment(a,108).
enrollment(b,102).
enrollment(c,108).
enrollment(d,341).
enrollment(e,455).

whens(102, 10).
whens(108, 12).
whens(341, 14).
whens(455, 16).
whens(452, 17).

where(102, z23).
where(108, z11).
where(341, z06).
where(455, 207).
where(452, 207).
```

All class information as facts.

```
%rules

schedule(Student, P, T) :- enrollment(Student, Class), where(Class, P), whens(Class,T).

usage(Room, T) :- where(Lesson, Room), whens(Lesson, T).

conflict(X,Y) :- (enrollment(X,LessonX), where(LessonX, Room), enrollment(Y,LessonY), where(LessonY, Room) ;
                  enrollment(X,LessonX), whens(LessonX, Time), enrollment(Y,LessonY), whens(LessonY, Time)),X\=Y.

meet(X,Y) :- enrollment(X, Lesson), enrollment(Y, Lesson), X\=Y.
```

All associated rules. conflict(X,Y) gives true if X and Y conflicts due to classroom or time. meet(X,Y) that gives true if student X and student Y are present in the same classroom at the same time

```
?- schedule(b,X,Y).
X = z23,
Y = 10.

?- schedule(c,X,Y).
X = z11,
Y = 12.

?- meet(a,b).
true .

?- meet(a,c)
|    .
true.

?- meet(a,d).
false.

?- conflict(a,d).
false.

?- conflict(a,b).
true
```

Test.

Part 4:

```
element(E,S) :- member(E,S).

delete(E, [E|S], S).
delete(E, [H|S], [H|NewSet]) :- delete(E, S, NewSet).

p([], []).

p(PermutationX,[H|T]) :- p(T, NewList), delete(H, PermutationX, NewList).

equivalent(F,S) :- permutation(F, S).

intersectx([], _, []).
intersectx([_|List1tail], List2, List3) :- intersectx(List1tail, List2, List3).
intersectx([H|List1tail], List2, List3) :- member(H, List2),!, List3 = [H|List3tail], intersectx(List1tail, List2, List3tail).

intersect(List,List2,List3):-  equivalent(List3,List4), intersectx(List,List2,List4), !.

unionx([], List, List).
unionx([H|List1tail], List2, [H|List3tail]) :- union(List1tail, List2, List3tail).
unionx([H|List1tail], List2, List3) :- member(H, List2), union(List1tail, List2, List3), !.

union(List1,List2,List3):- equivalent(List3,List4), unionx(List1,List2,List4), !.
```

Elemet: checks if an E is in S or not
Delete: deletes E from set
Equivalent: Uses delete and compares if two sets are permutation of each other.
equivalent(F,S) checks if a set is permutation of other set. If it satisfies, sets are equivalent. If one of the sets are empty, intersection set is empty too. Then by splitting list1 and checking if List2 has its head it creates List3 recursively.

```
?- element(4,[2,4,5,6]).
true
Unknown action: 0 (h for help)
Action? .

?- element(1,[2,4,5,6]).
false.

?- equivalent([1,2,6,7],[6,2,1,7]).
true .

?- equivalent([1,2,6,7],[6,2,1,7,9]).
false.

?- union([1,2,6],[1,4],U)
|
U = [2, 6, 1, 4].

?- intersection([5,6,1,2],[1,2,6,4,7,125],I).
I = [6, 1, 2].

?- intersection([5,6,1,2],[4,7,125],I).
I = [].
```

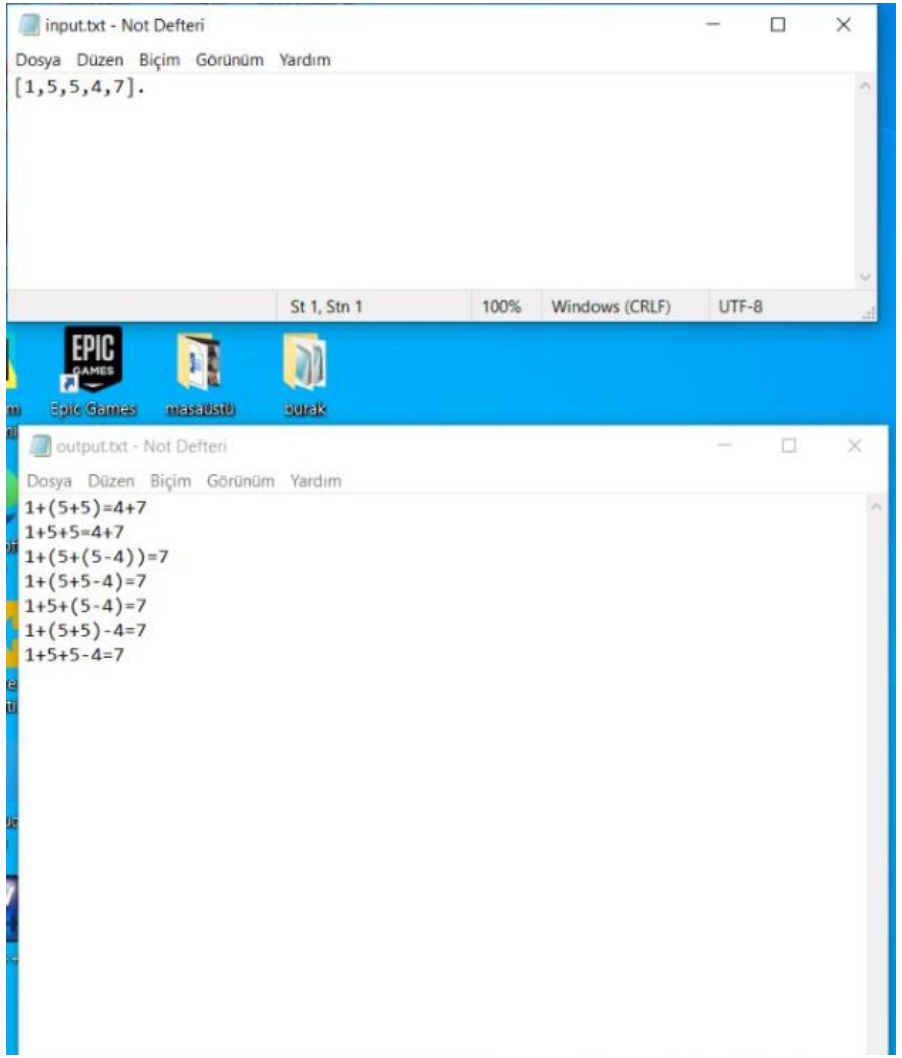
Test.

Part 5:

```
% splits list into 2 sections and tries possible solutions
section([X],X).
section(ItemList,Sec) :-
    split(ItemList,LeftSec,RightSec),
    section(LeftSec,LeftX),
    section(RightSec,RightX),
    sectT(LeftX,RightX,Sec).

% seperates list
seperate(ItemList,LeftX,RightX) :-
    split(ItemList,LeftB,RightB),
    section(LeftB,LeftX),
    section(RightB,RightX),
    LeftX =:= RightX.

% tries possiblities
sectT(LeftX,RightX,LeftX+RightX).
sectT(LeftX,RightX,LeftX-RightX).
sectT(LeftX,RightX,LeftX*RightX).
sectT(LeftX,RightX,LeftX/RightX) :- RightX =\= 0.

split(ItemList,List1,List2) :- List1 = [_|_], List2 = [_|_], append(List1,List2,ItemList).

%reads input.txt and invokes arithmatic function with input paramater
do :-
    open('input.txt',read,Str),
    read(Str,ItemList),
    close(Str),
    arithmatic(ItemList).

%does seperations and writes results in output.txt
arithmatic(ItemList) :-
    open('output.txt',write,Stream),
    close(Stream),
    seperate(ItemList,LeftX,RightX),
    open('output.txt',append,Out),
    write(Out, LeftX = RightX ),
    write(Out, "\n"),
    close(Out),
    fail.
arithmatic(_).
```

In this algorithm it seperates input list into 2 different list and tries all possible operators to obtain a proper equation that left side and right side of the equation is equal. And applies that method for all possible combinations of seperated lists.Lets say if input is [1,2,3,4]. Then it seperates it as [1] and [2,3,4] and it tries make them equal with operations. After that it tries it for [1,2] and [3,4] and so on. And writes all possible combinations into output file.

Example input and output file.



How to run program.

```
| ?- do.

yes
| ?- |
```