Pattern for reducing a member of class P to any other class (P or NPH or NPC) Need two instances!
yesInstance <- create/pick somehow
noInstance <- create/pick somehow
if isMemberOfSP(G, max) = yes then
    return yesInstance
return noInstance

Reduce Shortest Path to MST

Shortest Path (SP): Given a graph G and two vertices u and v and a max value. Does there exist a path from u to v with total weight at most max?
Minimum Spanning Tree (MST): Given a graph G and a max value. Does there exist a spanning tree with total weight at most max?

SP -> MST
Interface input and output structure:
    (G, u, v, max) -> (T, max)
T <- new Graph
x <- T.insertVertex("X")
y <- T.insertVertex("Y")
e <- T.insertEdge(x, y, 2) // 2 is the weight
yesInstance <- (T, 2) // MST instance
noInstance <- (T, 1) // MST instance

yesInstance <- (T, x, y, 2) // instance of SP
noInstance <- (T, x, y, 1) // instance of SP

To prove that a reduction B -> Q is valid, we have to show (argue informally at least) that
    $b \in B$ if and only if $R(b) \in Q$ when R is the reduction algorithm and R must run in $O(n^k)$ time.
$b \in B$ means that there is a solution to instance b of problem B
$b \notin B$ means that there is no solution to instance b of problem B

Therefore, it suffices to show (argue at least informally) that
if **arbitrary** instance b of problem B has a solution, then the specific instance R(b) of problem Q must have a solution.
and
if **arbitrary** instance $q \in R(B)$ (all instances mapped from B to instances of Q using R) has a solution, then there exists an instance b of B such that q=R(b) and b must also have a solution for problem B.

Solution to Homework Problem, Set Partition:

1. Randomly partition into P1 and P2 (flip a coin, heads into P1 and tails into P2), (P1,P2) is the guess.

```
Algorithm verifyPartition(S, P1, P2)
        sum1 <- 0
        For each e in P1 do
                sum1 <- sum1 + e
        sum2 <- 0
        For each e in P2 do
                sum2 <- sum2 + e
        if sum1 = sum2 then
                return yes
        return NOT_A_Solution
```

Version 2:
1. Randomly pick a subset P1 of S (flip a coin, heads into P1)
(we assume that other partition is P2=set difference of S and P1, i.e., P2=S-P1)

```
Algorithm verifyPartition(S, P1)
        sum1 <- 0
        For each e in P1 do
                sum1 <- sum1 + e
        sum <- 0
        For each e in S do
                sum <- sum + e
        if  sum1 = (sum-sum1)  then
                return yes
        return NOT_A_Solution
```

Version 3:
1. Randomly pick a subset P1 of S (flip a coin, heads into P1)

```
Algorithm verifyPartition(S, P1)
        sum1 <- 0
        For each e in P1 do
                sum1 <- sum1 + e
        P2 <- setDiff(S, P1)  // P2 is the set difference, elements from S that are not in P1.
        sum2 <- 0
        For each e in P2 do
                sum2 <- sum2 + e
        if  sum1 = sum2  then
                return yes
        return NOT_A_Solution
```

// need to define function **setDiff**.

Invalid Version of verify algorithm since Set Partition is not a member of P!

Randomly partition into P1 and P2 (flip a coin, heads into P1 and tails into P2)

```
Algorithm verifyPartition(S, P1, P2)
// not valid (means polynomial) since this algorithm is not polynomial, so cannot ignore guess!!!
        (T1, T2) <- setPartition(S)     // O(2^n)  so Not a valid NP algorithm because this line is exponential,
        For each e in T1 do
                sum1 <- sum1 + e
        sum2 <- 0
        For each e in T2 do
                sum2 <- sum2 + e
        if sum1 = sum2 then
                return yes
        return no
```

randomly pick a sequence of vertices from G.vertices and put into SV

```
Algorithm verifySP(G, u, v, max, SV)
1       if SV.elemAtRank(0) != u \/ SV.last().element() != v then return NOT_A_Solution
1       i <- 1
1       x <- SV.first().element()
1       sum <- 0
n       while i < SV.size() – 1 do
n               y <- SV.elemAtRank(i)
n               if ! G.areAdjacent(x, y) then return NOT_A_Solution
m               edges <- G.incidentEdges(x)
n               e <- edges.nextObject()
m               while edges.hasNext() /\ G.opposite(e, x) != y do  (m)
m                       e <- edges.nextObject()
n               sum <- sum + weight(e)
n               x <- y
n               i <- i + 1
1       if sum > max then return NOT_A_Solution
1       return yes
```