1. Use Dijkstra's shortest path algorithm. Start with initializing the array. Set G as the starting vertex with weight of 0. Then update the immediate adjacent vertices from G. Pick the last updated weight and visit that neighbor. Repeat pushing and popping of neighbors and paths until no path is smaller than previously recorded.
    a. S: {G, E, H, D, B, C, F, A}

Steps. G -> set all V to infinity, (check neighbor (C, D, E, F, H) and record weights (9, 7, 2, 8, 3)) -> visit last (H) -> (check +weight to weight (B:6)) -> visit B (check, no updates smaller) -> visit F -> update(A:15) -> visit A, non-found -> visit E -> update(D:5) -> visit D (C:8) -> visit C (A:12) -> visit A -> D -> C -> finished

	dv	pv
A	12	C
B	6	H
C	8	D
D	5	E
E	2	G
F	8	G
G	0	-
H	3	G

    b. Dijkstra.py = O(n^3)

```
def dijkstra(graph, V):
 distances = {vertex: float('infinity') for vertex in graph}
 distances[V] = 0 # distance from V to V is 0
 tmp = [(V, 0)] # load start vertex, stack
 while len(tmp) > 0:
   currVertex, currDist = tmp.pop()
   if currDist > distances[currVertex]:
     continue # if shorter distance found previously, skip replacement
   for nextV, weight in graph[currVertex].items(): # check in order
     distance = currDist + weight # weight of (walk + new vertex)
     if distance < distances[nextV]: # relaxation
       distances[nextV] = distance # replace shorter walk
       tmp.append((nextV, distance))
 return distances # array of dictionary ('A': weight, ..)
```

```
def main():
 V = ['A', ..]
 graph = {'A': {'C': 4, 'F': 7}, ..}}
 oVertex = []
 Viter = 0
 for source in V:
   arr = dijkstra(graph, source) # get shortest paths
   ver = ''
   max = 0
   for vertex, path in arr.items():
     if(path > max): # find longest path in V arr
       max = path
       ver = V[Viter]
   oVertex.append([ver, max])
   Viter += 1
 ver = ''
 min = float('infinity')
 for vertex, path in oVertex: # find shortest longest path of V's
   if(path < min):
     min = path
     ver = vertex
 print(ver, min)
```

c. Most optimal would be E, with the largest walk being 10
d. dijkstra2.py, They should be placed at the two vertices with the shortest route to any given other vertex that do not need to pass through the other optimal vertex. Perform the Dijkstra algorithm to find the most optimal for each vertex, then merge each vertex's shortest paths into single arrays. Find the merge with the largest path. Running time of, O(n^3)

```
def dijkstra(graph, V):
 [..] # same as above
 return distances # array of dictionary (‘A’: weight, ..)

# returns merged paths v with largest weight path between them
def getDouble(sources, paths):
 arr = []
 for i in range(len(sources)): # point 1
   tmp1 = []
   for v, w in paths[i].items():
     tmp1.append(w)
```

```
  for ii in range(i+1, len(sources)): # point 2
    tmp2 = tmp1[:] # copy
    iter = 0
    for v, w in paths[ii].items(): # merge point 1 & 2
      if(w < tmp2[iter]):
        tmp2[iter] = w
      iter += 1
    max = 0
    for w in tmp2: # get largest path
      if(w > max):
        max = w
    arr.append((sources[i]+sources[ii], max))
 return arr

# get paths from dijkstra, merge and sort double points, print best
def output(sources, graph):
 paths = []
 for source in sources:
   paths.append(dijkstra(graph, source)) # get shortest paths
 oVertex = getDouble(sources, paths)
 min = float('infinity')
 id = ''
 for v, w in oVertex:
   if(w < min):
     min = w
     id = v
 print(id)

def main():
 sources = ['A', ..]
 graph = {'A': {'C': 4, 'F': 7}, ..}}
 output(sources, graph)
```

e. Best two points are C and H as they have the lowest, largest path out of the merges of each vertex.

2. MST.py, used prim algo, O(ElogV) + O(n^2) = O(ElogV)

```
def minW(W, MST, V):
 min = float('infinity')
```

```
 for i in range(V):
   if(W[i] < min and MST[i] == False): # smaller edge found
     min = W[i]
     index = i
 return index

def prim(V, graph):
 W = [float('infinity')]*V # array of weights
 W[0] = 0 # set first to 0
 optimalV = [0]*V # array of optimal V's
 MST = [False]*V # array of checked V's with found smallest edge to it
 for count in range(V):
   min = minW(W, MST, V) # get index of closest V
   MST[min] = True # current V
   for i in range(V): # check for smaller weights of V[min] neighbors
     if((graph[min][i] > 0) and (MST[i] == False) and (W[i] > graph[min][i])):
       W[i] = graph[min][i]
       optimalV[i] = min
 return optimalV

def output(V, graph):
 max = 0
 optimalV = prim(V, graph)
 for i in range(V): # add weights together
   max += graph[i][optimalV[i]]
 print(max)

def main():
 arr = [] # master array of file ints
 with open('graph.txt', 'r') as f: # get file as master array
   while True:
     line = f.readline() # get line from file
     if(not line): # check if line is eof
       break
     for i in line.split(): # get chars between space
       arr.append(int(i))
 cases = arr.pop(0) # get num of test cases
 for count in range(cases):
   print("Test case", count+1, ": MST weight", end = ' ')
```

```
V = arr.pop(0) # get number of vertices
sets = []
for i in range(V): # get x, y of V's,put in sets
  tmp = []
  for ii in range(2):
    tmp.append(arr.pop(0))
  sets.append(tmp)
graph = []
for x1, y1 in sets: # get weights between V's, put in graph
  tmp = []
  for x2, y2 in sets:
    tmp.append(round(math.sqrt(pow((x1-x2),2) + pow((y1-y2),2))))
  graph.append(tmp)
output(V, graph)
```