```
from itertools import combinations
import numpy as np
import random
import math

# A function that does union of two sets of x and y
# (uses union by rank)
def unionSubTree(parent, rank, x, y):
    xroot = lookParent(parent, x)
    yroot = lookParent(parent, y)
    # Union subtree Rank
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else :
        parent[yroot] = xroot
        rank[xroot] += 1

def adjacency_matrix1(n):
    A = []
    Graph = [] #array edges with weight
    for i in range(0,n):                    #complexity of loop: big
O(n)
        A.append(i)
    TupleList = list(combinations(A, 2))
    for i in TupleList:                             #complexity of
loop: big O(n)
        i = i + (random.uniform(0, 1),)
        Graph.append(i)
    return Graph                            #complexity of 2 loops: 2*
big Olog(n)

def adjacency_matrix2(n):
    A = []
    Graph = []
    for i in range(0,n):                   #complexity of loop: Olog(n)
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        w = 0
        A.append(i)
        A[i] = [x,y,i]
    TupleList = list(combinations(A, 2)) #produced a array

    for i in TupleList:                             #complexity of
loop: Olog(n)
        weight = math.sqrt((i[1][0]-i[0][0])**2 + (i[1][1]-
i[0][1])**2)
```

```
        i = i + (weight,)
        Graph.append(i)
    u = Graph[0][1]
    return Graph

#Find an element i in set
# uses path compression method
def lookParent(parent, i):                                  #
recurssion complexity is linear time.
    if parent[i] == i:
        return i
    return lookParent(parent, parent[i])


def kruskal(n, x):
    MST =[]
    sorted_edges_i = 0
    e = 0
    parent = []
    rank = []

    # Sorted edges by weight in increasing order
    if x == 1:                                      # the input is 1 will
calculate the question a) of problem 4.
        graph = sorted(adjacency_matrix1(n), key=lambda x: x[2])

        # Create V subsets with single elements
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
            rank.append(0)
        # Number of edges to be taken is equal to V-1
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u,v,w = graph[sorted_edges_i]
            x = lookParent(parent, u)
            y = lookParent(parent ,v)
            if x != y:
                e +=1
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

        else:                                                   # the
input is not 1, will calculate the question b) of problem 4
        graph = sorted(adjacency_matrix2(n), key=lambda x: x[2])
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
```

```
            rank.append(0)
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u, v, w = graph[sorted_edges_i] #([x,y],[a,b],w)
            u = graph[sorted_edges_i][0]
            v = graph[sorted_edges_i][1]
            x = lookParent(parent,u[2])
            y = lookParent(parent,v[2])
            if x != y:
                e +=1
                u = graph[sorted_edges_i][0]
                u = [u[0],u[1]]
                v = graph[sorted_edges_i][1]
                v = [v[0],v[1]]
                w = graph[sorted_edges_i][2]
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

    return MST
#complexity of two loops is 2*O(n) with if statement is 2*O(N*M)

def aveWeight():
    print("entered")
    n = [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
    j = n[9];
    lengthRun = [kruskal(j,2), kruskal(j,2), kruskal(j,2),
         kruskal(j,2), kruskal(j,2), kruskal(j,2),
        ]
    total_weight = 0
    for i in lengthRun:
#complexity of two loops: big O(n^2)
        for j in i:
            total_weight += j[2]
    print("------------", total_weight/6)
    return total_weight/6

aveWeight()
```

**Problem 1** (10 pts). *Chapter 4, Exercise 24, on p. 200. Provide a greedy algorithm to solve this problem, ideally in linear time in the size of the tree. You need to argue for the running time and prove that your algorithm gives a correct solution that minimizes the sum of the added edge lengths.*

**24.** Timing circuits are a crucial component of VLSI chips. Here's a simple model of such a timing circuit. Consider a complete balanced binary tree with $n$ leaves, where $n$ is a power of two. Each edge $e$ of the tree has an associated length $\ell_e$, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

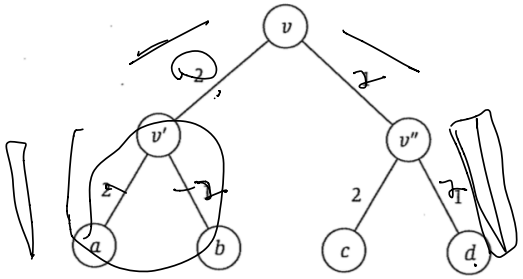



**Figure 4.20** An instance of the zero-skew problem, described in Exercise 23.

The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

**Example.** Consider the tree in Figure 4.20, in which letters name the nodes and numbers indicate the edge lengths.

The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

The Algorithm of this question use divide and conquer.

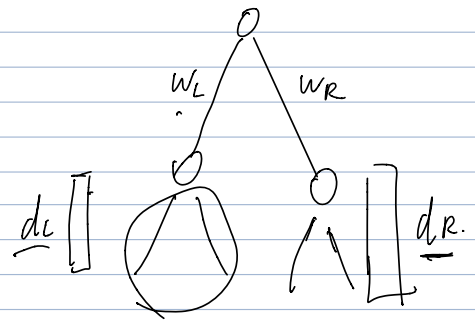
Let. $d_L$ = length of left leaves.

$d_R$ = length of right leaves.

$S_L$ = Sum of left edges

$S_R$ = sum of Right edges
$M$ = Total length of layers of the tree.
Subtree = two leaves share a node

While any two leaves share same node $< M$: $\Longrightarrow$ complexity $O(n)$
  if subtree not visit yet.
    if: $d_L <= d_R$:
      $S_L$ += $d_R - d_L$.
    else: $S_R$ += $d_L - d_R$
  else: $m = m - 1$
end while;
if: $S_L > S_R$.
  reture $S_L - S_R$
else:
  return $S_R - S_L$
Endif
end code;




Inductive proof:

Base case: when a tree with one node, two leaves L, & R, one layer the increase length of certain edge is the difference of L. & R.

Inductive step:
Assume. tree G has n+1 nodes, m+1 layers. the algorithm is correct. when G has n nodes. will be the m layer. the total increase length of certain edge at m layer equals the length of the certain edge plus the difference of length of left edge and right edge in the m+1 layer. Thus total. accumulate of the increasing length will be the minimum length to meet zero skew. the complexity is the linear time of the tree G.

b). correctness: 1st, Find any node with two leaves $d_L, d_R$ in tree G. the increase length is the difference of its two leaves, which should add to the edge with smaller length. then we get $d_L = d_R$. The we say the increase length $I_L$, let $d_L, d_R$ meet zero skew. which is correct for the subtree. similarly. we apply this process to every leave nodes. this step is correct.

Second). To one of the subtree in step 1. we add the increase length $I_L$. to the previous edge $W_L$. which connect with the subtree node, we say the previous edge length equal $W_L + S_L$ similiar, we find the length of sibling of the previous edge $W_R$. To these siblings, the increase of the length is $(W_L + S_L) - (W_R + S_R)$. then we update the Increase length $S_L$ and $S_L$. to its previous Edge, which in previous layer $(M-2)$. which is correct.

third). repeat step 2. until to the root node. which is no more previous edge. the total increasing length is the different of $S_L$ and $S_R$. Base on the requirement, we only can increase the length of edge. therefore, this algorithm is correct.

**19.** A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge $e$ is a communication link, with a given available bandwidth $b_e$.

For each pair of nodes $u, v \in V$, they want to select a single $u$-$v$ path $P$ on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path $P$ is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair $u, v$ in $G$ is simply the maximum, over all $u$-$v$ paths $P$ in $G$, of the value $b(P)$.

It's getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree $T$ of $G$ so that for *every* pair of nodes $u, v$, the unique $u$-$v$ path in the tree actually attains the best achievable bottleneck rate for $u, v$ in $G$. (In other words, even if you could choose any $u$-$v$ path in the whole graph, you couldn't do better than the $u$-$v$ path in $T$.)

This idea is roundly heckled in the offices of CluNet for a few days, and there's a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree $T$ in which, for each $u, v \in V$, the bottleneck rate of the $u$-$v$ path in $T$ is equal to the best achievable bottleneck rate for the pair $u, v$ in $G$.

Due to Kruskal Algorith, we need to constrct the maximum spanning tree. The complexity $O(m \log n)$

1. Sort the edges of G into decreasing order by weight. Let T be the set of edges comprising the maximum weight spanning tree. Set $T = \emptyset$.

2. Add the first edge to T.

3. Add the next edge to T if and only if it does not form a cycle in T. If there are no remaining edges exist and report G to be disconnected.
4. If T has n-1 edges (where n is the number of vertices in G) stop and output T. otherwise, go to step 3.

Correctness by contradiction:

Step1: Assume there exist one edge $m_1$ less than any edge in the maximum spanning tree, G. By the algorithm defined, all edges are sorted, either the edge would be delete, or it will construct a cycle, which less than any edge in G. Thus, the Assume is not correct.

step2: The maximum spanning tree can delete the smallest edge, which is the bottleneck. Assume there is a better path B better than the path A we find. Then in path B, the smallest edge will greater than the smallest edge in path A. then the path B will not exist, Because path B must exist one edge, already deleted by our algorithm.

3). If the edge costs are not all distinct, we perturb all edge bandwidths by extremely small amount so they become distinct, and then find a minimum spanning tree. we therefore refer for each edge e, to a real bandwidth $b_e$ and a perturbed bandwidth $b_e'$. In particular, we choose perturbations small enough so that if $b_e > b_f$ for edges e and f, then also $b_e' > b_f'$ our tree has the best bottleneck rate for all pairs, under the perturb bandwidth. But suppose that the u-v path P in this tree did not have the best bottleneck rate if we consider the original, real bandwidths: say there is a better path P', then it exists an edge E on P for $b_e > b_f$ for all edges f on P'. But the perturbations were so small that they did not cause any edges with distinct bandwidths to change the relative order of their bandwidth values, so it would follow that $b_e' > b_f'$ for all edges f on P' Therefore, P had the best bottleneck rate with we want to the perturbed bandwidth.

Proof: Induction. Step 1: if the graph $G_1$ has three edges, sorted. the smallest edge will not connect with the spanning tree T, unless it will not produces a cycle. That is the maximum spanning tree.

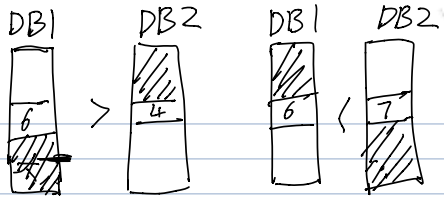
Inductive step: Assume the (n+1)th edge in $G_1$ without produce a circle in the spanning tree, that is correct. the nth edge connect with the spanning tree without produces a cycle. cannot be deleted, as nth edge is greater than (n+1) edge, as it is sorted. that is should be in maximum spanning tree., also correct.

---

1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains $n$ numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the $n^{th}$ smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value $k$ to one of the two databases, and the chosen database will return the $k^{th}$ smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.




Algorithm: Median finding algorithm for joint databases.

```
1. p1 = p2 = n/2   // points of two query in two DB.
2. for i = 2 to log n do
     m1 = Query DB (BD1 P1)
     m2 = Query DB (BD2 P2)
     if m1 > m2
        p1 ← p1 − n/2^i
        p2 ← p2 + n/2^i
     else:
        p1 ← p1 + n/2^i
        p2 ← p2 − n/2^i
     end if
   endfor
   return min(q1, q2)
```

} get median from each database.

Complexity:
let T(n) be total number of query.
for each round,

$P_1$ and $P_2$ are two query pointing for both databases. We first query the median of both database to obtain $m_1, m_2$. To acheive this we show the median of the joint database in between $m_1, m_2$

observe that there are $n$ records in $DB_1$ and $DB_2$ which are smaller or equal $\max(m_1, m_2)$. Thus, the median of joint database is not greater than $\max(m_1, m_2)$. Similiary, it not smaller than $\min(m_1, m_2)$ move the pointer $P_1$ & $P_2$ accordingly. By the end of the loop, $m_1$ & $m_2$ are the $n$th and $(n+1)$th smallest number of the joint database. Thus.

we reduce the problem size by half using two query, we have $T(n) = T(n/2) + 2$ by recurrence. $T(n) = O\log(n)$

proof: Contradiction. Assume the values pick is not the median. the number in $DB_1$ or $DB_2$ which will not greater than equal $\max(m_1, m_2)$ or smaller than $\min(m_1, m_2)$. In other words, the query value K to one of the $DB_1$ or $DB_1$ will not return the $k^{th}$ smallest value. Thus, the assume is wrong

4. **Problem 4** (20 pts). *In this programming problem, we will be considering MSTs on complete, undirected graphs. A graph with $n$ vertices is complete if all possible $\binom{n}{2}$ edges are present in the graph. Consider the following two types of graphs:*

- *Complete graphs on $n$ vertices, where the weight of each edge is a real number chosen uniformly at random from $[0, 1]$.* $(n-1)$ edge
- *Complete graphs on $n$ vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are $(x, y)$, with $x$ and $y$ each a real number chosen uniformly at random from $[0, 1]$.) The weight of an edge is the Euclidean distance between its endpoints.* run multiple time → find aver

*Your goal is to determine how the average weight of the minimum spanning tree grows as a function of $n$ for each of these families of graphs. You will need to implement an MST algorithm and procedures that generate the appropriate random graphs. Run your program on at least five random graph instances for*

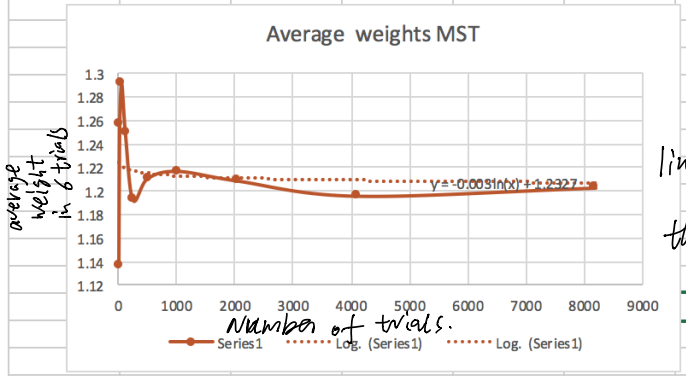
$$n = 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 \quad (2^n)$$

*and report the average for each value of $n$. (It is possible, that depending on your implementation you won't be able to run it for the largest graphs. ) For each family of graphs, generate an appropriate figure that depicts your results. Clearly label the axes and think carefully about the most effective representation (should it be a bar chart, line chart, scatterplot, etc.). Also, interpret your results by giving (and plotting) two sample functions that characterize each of your depicted results. For example, your answer might be $f(n) = 2\log n$, $f(n) = 1.5\sqrt{n}$, etc. Also, provide a few sentences of intuition for why the growth rate of your functions $f(n)$ are reasonable. Do this for both types of graphs separately.*

*As per usual, submit both the written answer (and charts) as well as your code as part of your pdf submission in Gradescope.*

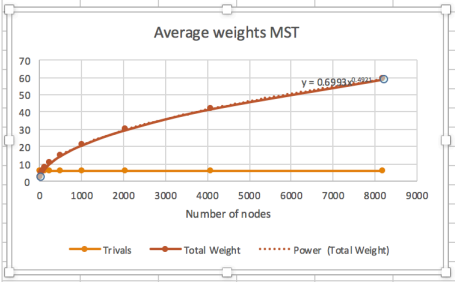
*You will work on this assignment as individuals. However, the same collaboration policy applies as with any other assignment; you may discuss with classmates as long as the conclusions you write are your own and you disclose your collaboration.*

| N edges | trivals | TotalWeight |
|---|---|---|
| 16 | 6 | 1.13635548 |
| 32 | 6 | 1.25689264 |
| 64 | 6 | 1.29073811 |
| 128 | 6 | 1.24908148 |
| 256 | 6 | 1.19256974 |
| 512 | 6 | 1.21056924 |
| 1024 | 6 | 1.21656288 |
| 2048 | 6 | 1.20845153 |
| 4096 | 6 | 1.19564456 |
| 8192 | 6 | 1.20244777 |




a) the graph Observation: in number of n vertices, the weight of edges randomly chose from [0, 1]. after find the sum of total minimum spanning tree for n = 16, 32 .... 8192. each n for 6 times, we calculate the average weight of MST. as the graph shows. with best fitting on excel. the best fitting line is linear in logarithm. $y = -0.0032\ln(x) + 1.2327$ that is the number of nodes increasing, the graph will convegant to 1.2.

| N edges | Trivals | Total Weight |
|---|---|---|
| 16 | 6 | 2.53761099 |
| 32 | 6 | 4.17196422 |
| 64 | 6 | 5.44020045 |
| 128 | 6 | 7.65204265 |
| 256 | 6 | 10.7355074 |
| 512 | 6 | 15.0012159 |
| 1024 | 6 | 21.041924 |
| 2048 | 6 | 29.6541337 |
| 4096 | 6 | 41.7790665 |
| 8192 | 6 | 58.8879613 |




b). the vertices. n is 16, 32 ... 8192, chose randomly from a unit square. [0, 1] with the distance. calculated by $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. after 6 trivals on each number of vertices. the graph as showing. it runs time growth in power, $y = 0.6993x^{0.9971}$.

Please see the code on the first page. Thank you

Please see the code above

Pelase ignore the code below

```
from itertools import combinations
import numpy as np
import random
import math

# A function that does union of two sets of x and y
# (uses union by rank)
def unionSubTree(parent, rank, x, y):
    xroot = lookParent(parent, x)
    yroot = lookParent(parent, y)
    # Union subtree Rank
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else :
        parent[yroot] = xroot
        rank[xroot] += 1

def adjacency_matrix1(n):
    A = []
    Graph = [] #array edges with weight
    for i in range(0,n):                    #complexity of loop: big
O(n)
        A.append(i)
    TupleList = list(combinations(A, 2))
    for i in TupleList:                             #complexity of
loop: big O(n)
        i = i + (random.uniform(0, 1),)
        Graph.append(i)
    return Graph                            #complexity of 2 loops: 2*
big Olog(n)

def adjacency_matrix2(n):
    A = []
    Graph = []
    for i in range(0,n):                   #complexity of loop: Olog(n)
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        w = 0
        A.append(i)
        A[i] = [x,y,i]
    TupleList = list(combinations(A, 2)) #produced a array

    for i in TupleList:                             #complexity of
loop: Olog(n)
        weight = math.sqrt((i[1][0]-i[0][0])**2 + (i[1][1]-
i[0][1])**2)
```

```
        i = i + (weight,)
        Graph.append(i)
    u = Graph[0][1]
    return Graph

#Find an element i in set
# uses path compression method
def lookParent(parent, i):                                  #
recurssion complexity is linear time.
    if parent[i] == i:
        return i
    return lookParent(parent, parent[i])


def kruskal(n, x):
    MST =[]
    sorted_edges_i = 0
    e = 0
    parent = []
    rank = []

    # Sorted edges by weight in increasing order
    if x == 1:                                      # the input is 1 will
calculate the question a) of problem 4.
        graph = sorted(adjacency_matrix1(n), key=lambda x: x[2])

        # Create V subsets with single elements
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
            rank.append(0)
        # Number of edges to be taken is equal to V-1
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u,v,w = graph[sorted_edges_i]
            x = lookParent(parent, u)
            y = lookParent(parent ,v)
            if x != y:
                e +=1
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

        else:                                                   # the
input is not 1, will calculate the question b) of problem 4
        graph = sorted(adjacency_matrix2(n), key=lambda x: x[2])
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
```

```
            rank.append(0)
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u, v, w = graph[sorted_edges_i] #([x,y],[a,b],w)
            u = graph[sorted_edges_i][0]
            v = graph[sorted_edges_i][1]
            x = lookParent(parent,u[2])
            y = lookParent(parent,v[2])
            if x != y:
                e +=1
                u = graph[sorted_edges_i][0]
                u = [u[0],u[1]]
                v = graph[sorted_edges_i][1]
                v = [v[0],v[1]]
                w = graph[sorted_edges_i][2]
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

    return MST
#complexity of two loops is 2*O(n) with if statement is 2*O(N*M)

def aveWeight():
    print("entered")
    n = [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
    j = n[9];
    lengthRun = [kruskal(j,2), kruskal(j,2), kruskal(j,2),
         kruskal(j,2), kruskal(j,2), kruskal(j,2),
        ]
    total_weight = 0
    for i in lengthRun:
#complexity of two loops: big O(n^2)
        for j in i:
            total_weight += j[2]
    print("------------", total_weight/6)
    return total_weight/6

aveWeight()
```

```
from itertools import combinations
import numpy as np
import random
import math

# A function that does union of two sets of x and y
# (uses union by rank)
def unionSubTree(parent, rank, x, y):
    xroot = lookParent(parent, x)
    yroot = lookParent(parent, y)
    # Union subtree Rank
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else :
        parent[yroot] = xroot
        rank[xroot] += 1

def adjacency_matrix1(n):
    A = []
    Graph = [] #array edges with weight
    for i in range(0,n):                    #complexity of loop: big
O(n)
        A.append(i)
    TupleList = list(combinations(A, 2))
    for i in TupleList:                             #complexity of
loop: big O(n)
        i = i + (random.uniform(0, 1),)
        Graph.append(i)
    return Graph                            #complexity of 2 loops: 2*
big Olog(n)

def adjacency_matrix2(n):
    A = []
    Graph = []
    for i in range(0,n):                   #complexity of loop: Olog(n)
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        w = 0
        A.append(i)
        A[i] = [x,y,i]
    TupleList = list(combinations(A, 2)) #produced a array

    for i in TupleList:                             #complexity of
loop: Olog(n)
        weight = math.sqrt((i[1][0]-i[0][0])**2 + (i[1][1]-
i[0][1])**2)
```

```
        i = i + (weight,)
        Graph.append(i)
    u = Graph[0][1]
    return Graph

#Find an element i in set
# uses path compression method
def lookParent(parent, i):                                  #
recurssion complexity is linear time.
    if parent[i] == i:
        return i
    return lookParent(parent, parent[i])


def kruskal(n, x):
    MST =[]
    sorted_edges_i = 0
    e = 0
    parent = []
    rank = []

    # Sorted edges by weight in increasing order
    if x == 1:                                      # the input is 1 will
calculate the question a) of problem 4.
        graph = sorted(adjacency_matrix1(n), key=lambda x: x[2])

        # Create V subsets with single elements
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
            rank.append(0)
        # Number of edges to be taken is equal to V-1
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u,v,w = graph[sorted_edges_i]
            x = lookParent(parent, u)
            y = lookParent(parent ,v)
            if x != y:
                e +=1
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

        else:                                                   # the
input is not 1, will calculate the question b) of problem 4
        graph = sorted(adjacency_matrix2(n), key=lambda x: x[2])
        for i in range(0,n):
#complexity of loop: big O(n)
            parent.append(i)
```

```
            rank.append(0)
        while e < n-1 :
#complexity of loop: big O(n)
            sorted_edges_i += 1
            u, v, w = graph[sorted_edges_i] #([x,y],[a,b],w)
            u = graph[sorted_edges_i][0]
            v = graph[sorted_edges_i][1]
            x = lookParent(parent,u[2])
            y = lookParent(parent,v[2])
            if x != y:
                e +=1
                u = graph[sorted_edges_i][0]
                u = [u[0],u[1]]
                v = graph[sorted_edges_i][1]
                v = [v[0],v[1]]
                w = graph[sorted_edges_i][2]
                MST.append([u,v,w])
                unionSubTree(parent, rank, x, y)

    return MST
#complexity of two loops is 2*O(n) with if statement is 2*O(N*M)

def aveWeight():
    print("entered")
    n = [16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
    j = n[9];
    lengthRun = [kruskal(j,2), kruskal(j,2), kruskal(j,2),
         kruskal(j,2), kruskal(j,2), kruskal(j,2),
        ]
    total_weight = 0
    for i in lengthRun:
#complexity of two loops: big O(n^2)
        for j in i:
            total_weight += j[2]
    print("------------", total_weight/6)
    return total_weight/6

aveWeight()
```