Write a function **Find_Min_Difference(L, P)** that accepts a list **L** of integers and **P** (positive integer) where the size of `L` is greater than `P`. The task is to pick `P` different elements from the list `L`, where the difference between the maximum value and the minimum value in selected elements is minimum compared to other differences in possible subset of `p` elements. The function returns this minimum difference value.

Note - The list can contain more than one subset of p elements that have the same minimum difference value.

Example

Let `L = [3, 4, 1, 9, 56, 7, 9, 12, 13]` and `P = 5`

If we see the following two subsets of 5 elements from `L`

[3, 4, 7, 9, 9] or [7, 9, 9, 12, 13]

Here, the difference between the maximum value and the minimum value in both subset is `9 - 3 = 6` or `13 - 7 = 6` which is minimum. So the output will be `6`.

Sample Input

```
[3, 4, 1, 9, 56, 7, 9, 12]
5
```

Output

```
6
```

```
def find_Min_Difference(L,P):
  L.sort()
  N = P
  M = len(L)
  min_diff = max(L) - min(L)
  for i in range(M-N+1):
    if L[i+N-1] - L[i] < min_diff:
      min_diff = L[i+N-1] - L[i]
  return min_diff
```

Goldbach's conjecture is one of the oldest and best-known unsolved problems in number theory. It states that every even number greater than 2 is the sum of two prime numbers.

For Example:

12 = 5 + 7

26 = 3 + 23 or 7 + 19 or 13 +13

Write a function **Goldbach(n)** where `n` is a positive even number(`n > 2`) that returns a list of tuples. In each tuple `(a,b)` where `a <= b`, `a` and `b` should be prime numbers and the sum of `a` and `b` should be equal to `n` .

Sample Input 1

```
12
```

Output

```
[(5,7)]
```

Sample Input 2

```
26
```

Output

```
[(3,23),(7,19),(13,13)]
```

```
def prime(n):
 if n < 2:
  return False
 for i in range(2,n//2+1):
  if n%i==0:
   return False
 return True


def Goldbach(n):
 Res=[]
 for i in range((n//2)+1):
  if prime(i)==True:
   if prime(n-i)==True:
```

```
    Res.append((i,n-i))
return(Res)
```

Write a function named `odd_one(L)` that accepts a list `L` as argument. Except for one element, all other elements in `L` are of the same data type. The function `odd_one` should return the data type of this odd element.

For example, if `L` is equal to `[1, 2, 3.4, 5, 10]`, then the function should return the string `float`. This is because the element `3.4` is the odd one here, all other elements are integers.

Note

(1) `L` has at least three elements.

(2) For each test case, the elements in the list will only be of one of these four types: `int`, `float`, `str`, `bool`.

(3) The function must return one of these four strings: `int`, `float`, `str`, `bool`.

(4) Hint: `type(1) == int` evaluates to True.

```
def eltype(x):
    if type(x)==int:
        return 'int'
    elif type(x)==float:
        return 'float'
    elif type(x)==bool:
        return 'bool'
    else:
        return 'str'


def odd_one(L):
  odd=type(L[0])
  c=0
  for i in range(1,len(L)):
      if type(L[i])!=odd:
          c+=1
          item=L[i]
  if c==1:
```

```
        return eltype(item)
    else:
        return eltype(L[0])
```

OR

```
def odd_one(L):
  P = {}
  for elem in L:
    if type(elem) not in P:
      P[type(elem)] = 0
    P[type(elem)] += 1
  for key, value in P.items():
    if value == 1:
      return key.__name__
```

Write a Python function ***combinationSort(strList)*** that takes a list of unique strings `strList` as an argument, where each string is a combination of a letter from `a` to `z` and a number from `0` to `99`, the initial character in string being the letter. For example `a23`, `d5`, `q99` are some strings in this format. This function should sorts the list and return two lists `(L1, L2)` in the order mentioned below.

`L1`: First list should contain all strings sorted in ascending order with respect to the first character only. All strings with same initial character should be in the same order as in the original list.

`L2`: In the list `L1` above, sort the strings starting with same character, in descending order with respect to the number formed by the remaining characters.

Example:

Sample input 1:

```
d34, g54, d12, b87, g1, c65, g40, g5, d77
```

Sample output 1:

```
L1: b87, c65, d34, d12, d77, g54, g1, g40, g5
L2: b87, c65, d77, d34, d12, g54, g40, g5 ,g1
```

#python

```
import string
def combinationSort(strList):
  # Create a dictionary with 26 keys from characters 'a' to 'z', each key has an empty list as value.
  groups = {k: [] for k in string.ascii_lowercase}

  # Using this dictionary to group strings with same initial character.
  for i in range(len(strList)):
    char=strList[i][0]
    groups[char].append(strList[i])

  strList=[]
  # Recreate the list from all the strings in groups, iterating on groups from a to z.
  for char in groups.keys():
    for s in groups[char]:
      strList.append(s)

  L1 = strList.copy() # Saving intermediate result to return later.
  i = 1
  left = 0
  # As there can be no more than 100 strings with same initial character.
  # Using insertion sort within group.
  while i<len(strList):
    right = i
    while(right>left and strList[right][0] == strList[right-1][0] and int(strList[right-
1][1:])<int(strList[right][1:])):
      strList[right], strList[right-1] = strList[right-1], strList[right]
      right -= 1
    i += 1

  return L1, strList
```

Complete the python function **findLargest(L)** below, which accepts a list L of unique numbers, that are sorted(ascending) and rotated n times, where n is unknown, and returns the largest number in list `L`. Rotating list `[2, 4, 5, 7, 8]` one time gives us list `[8, 2, 4, 5, 7]`, and rotating the second time gives list `[7, 8, 2, 4, 5]` and so on. Try to give an $O(log\ n)$ solution. Hint: One of the $O(log\ n)$ solutions can be implemented using binary search and using 'first or last' element to know, the direction of searching further.

```
# input<L>: List L sorted and rotated.
# out: Return the largest number in list L.
def findLargest(L):
    # Your code goes here
```

Sample input:

```
7, 8, 2, 4, 5
```

Sample output:

```
8
```

#python

```
def findLargest(L):
 left = 0
 s = len(L)
 right = s-1

 # If list has only one element, that is the max.
 if (s==1):
  return L[0]

 while (left<=right):
  mid=(left+right)//2

  # if mid is at last index, next element to compare will be at index 0
  if (mid == s-1):
   nextToMid = 0
  else:
   nextToMid = mid+1
```

```
if (L[mid] > L[nextToMid]):
  return L[mid]
elif (L[mid] < L[0]):
  # our element is in left of mid
  right = mid-1
else:
  # our element is in right of mid
  left = mid+1
```

Merging two sorted arrays in place.

Given a custom implementation of list named `MyList`. On `MyList` objects you can perform read operations similar to the in-build lists in Python, example use `A[i]` to read element at index `i` in `MyList` object `A`. The only possible operation that you can use to edit data in `MyList` objects is by calling the `swap` method. For instance, `A.swap(indexA, B, indexB)` will swap values at `A[indexA]` and `B[indexB]` and `A.swap(index1, A, index2)` will swap values at `A[index1]` and `A[index2]`, where `indexA`, `indexB`, `index1`, `index2` are all integers.

Complete the Python function `mergeInPlace(A, B)` that accepts two `MyLists` `A` and `B` containing integers that are sorted in ascending order and merges them in place(without using any other list) such that after merging, `A` and `B` are still sorted in ascending order with the smallest element of both `MyLists` as the first element of `A`.

```
# Complete this function
def mergeInPlace(A, B):
  # Your code goes here
```

Sample Input:

```
2 4 6 9 13 15
1 3 5 10
```

Sample Output:

```
[1, 2, 3, 4, 5, 6]
[9, 10, 13, 15]
```

Sample Input:

```
4 6
1 3 6 10
```

Sample Output:

```
1 3
4 6 6 10
```

```
#python
def mergeInPlace(A, B):
  m = len(A)
  n = len(B)
  if (m < 1 or n < 1):
    return

  # Find the smaller list of A and B.
  for i in range(0, m):
    # A and B are already sorted. B[0] will always be least in B,
    # as we will maintain its sortedness .
    if (A[i] > B[0]):
      A.swap(i, B, 0)

      # move `B[0]` to its correct position in B to maintain the sortedness of B
      j = 0
      while(j < n - 1 and B[j] > B[j + 1]):
```

```
    B.swap(j+1, B, j)
    j += 1
```

A restaurant always prepares dishes with the most orders before others with a lesser number of orders. Each dish in the restaurant menu has a unique integer ID. The restaurant receives `n` orders in a particular time period. The task is to find out the order of dish IDs according to which the restaurant will prepare them. Assume that restaurant has the following unique dish IDs in its menu:

`[1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009]`

Write a function **DishPrepareOrder(order_list)** that accepts `order_list` in the form of a list of dish IDs and returns a list of dish IDs in the order in which the restaurant will prepare them. If two or more dishes have the same number of orders, then the dish which has a smaller ID value will be prepared first.

Sample input

```
1 [1004,1003,1004,1003,1004,1005,1003,1004,1003,1002,1005,1002,1002,1001,1002,1
  002,1002]
```

Output

```
1 [1002, 1003, 1004, 1005, 1001]
```

```
def insertionsort(L): #use this because it is stable sort
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        j = i
        while(j > 0 and L[j][1] > L[j-1][1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
    return(L)


def DishPrepareOrder(order_list):
    order_count = {}
```

```
R = []
for order in order_list:
    if order in order_count:
        order_count[order] += 1
    else:
        order_count[order] = 1
for ID in sorted(order_count.keys()):
    R.append((ID,order_count[ID]))
R=insertionsort(R)
Res = []
for tup in R:
    Res.append(tup[0])
return Res
```

An arithmetic expression can be written in postfix form where each operator appears after the operands.

For example:

- `A + B` is written as `A B +`
- `(A + B) * (C - D)` is written as `A B + C D - *`.

The expressions written in the given form are easier to evaluate compared to normal expressions, as parenthesis are not required.

A postfix expression can be evaluated with a stack using the following algorithm :

- Create a stack to store operands (or numbers).
- Scan each element in the given expression left to right.
    - If the element is a number (operand), push it into the stack.
    - If the element is an operator, pop two operands (Assume that the first popped element is `B` and the second popped element is `A`) for the operator `op` from the stack. Evaluate the operation (`A op B`) and push the result back to the stack.
- When the expression is ends, the number in the stack is the final answer.

Write a function **EvaluateExpression(exp)**, that accepts an expression `exp` in string format, where each items are separated by the space. The function returns the evaluated value.

Note: Assume that the expression has only `+` , `-` , `*` , `/` and `**` operators.

Sample Input

```
2 3 1 * + 9 -
```

Output

```
-4.0
```

```
class create_stack:
  def __init__(self):
    self.stack = []
  def push(self,d):
    self.stack += [d]
  def pop(self):
    t = self.stack[-1]
    self.stack = self.stack[:-1]
    return t

def EvaluateExpression(exp):
  opt = ['+','-','*','/','**']
  stk = create_stack()
  L = exp.split(' ')
  for i in L:
    if i not in opt:
      stk.push(i)
    else:
      b = stk.pop()
      a = stk.pop()
      res = eval(a + i + b)
      stk.push(str(res))
  return stk.pop()
```

Complete the below function `reverse(root)` that will reverse the linked list with the first node passed as an argument, and return the first node of the reversed list. Each node in the linked list is an object of class `Node`. Class `Node` members are described below.

Class Node:

- `value` - stored value.
- `next` - points to next node.
- `isEmpty()` - returns True if linked list is empty, False otherwise.

Note- The list is to be reversed only by changing `next` pointer of existing nodes. No values to be changed and no new nodes to be created.

```
# Implement this function
def reverse(root):
  # Your code goes here.
```

Sample Input 1

```
64,7,28,43,3
```

Sample Output 1

```
3,43,28,7,64
```

Sample Input 2

```
12
```

Sample Output 2

```
12
```

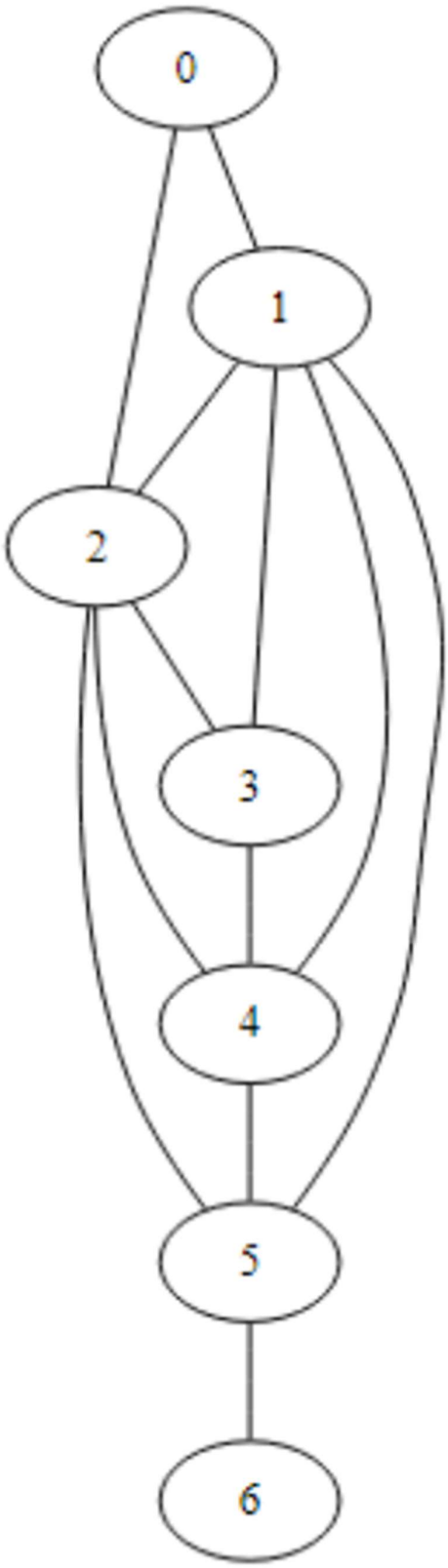
```
def reverse(root):
 if (root.isEmpty()):
  return root
 temp = root
 prev = None
 while (temp):
  next, temp.next = temp.next, prev
  prev, temp = temp, next
```

return prev

Consider a social network of friends/relatives, most of whom are closely connected. Visualize this as a graph where each vertex denotes a person, and if two people know each other there is an edge between the vertices denoting them. If persons `x` and `y` know each other directly, then there is an edge connecting `x` and `y` and level of connectivity between them is `1`. If person `x` is a friend of person `y` and person `y` is friend of person `z`, but `x` is not a friend of `z`, then the level of connectivity between `x` and `z` is `2`, and so on. The connectivity between people is always two way, i.e. if x directly knows y, then y also knows x directly.

Complete the Python function `findConnectionLevel(n, Gmat, Px, Py)` that takes 4 arguments, number of persons `n` (n persons numbered from `0` to `n-1`), `Gmat` an adjacency matrix representation of `n` persons and their connections(if `Gmat[x][y] = 1`, then person `x` and `y` are directly connected), two persons `Px` and `Py` both numbers, and returns the minimum level of connectivity between `Px` and `Py`. Return 0 if `Px` and `Py` are not connected through anybody in the group.

For example, for the graph below representing 7 persons from 0 to 6 and their connectivity.


0
1
2
3
4
5
6

```
n: 7
Gmat: [[0, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0],
       [0, 1, 1, 1, 0, 1, 0],
       [0, 1, 1, 0, 1, 0, 1],
       [0, 0, 0, 0, 0, 1, 0]]
Px: 0
Py: 6
```

Return: (level of connectivity)

```
3
```

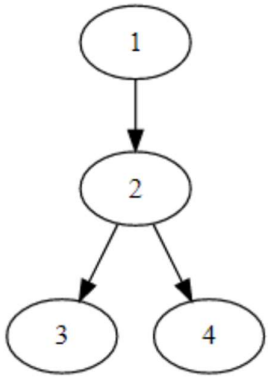
```
# Helper function
from collections import deque
class myQueue:
  def __init__(self):
    self.Q = deque()

  def deQueue(self):
    return self.Q.popleft()

  def enQueue(self, x):
    return self.Q.append(x)

  def isEmpty(self):
    return False if self.Q else True

def findConnectionLevel(n, Gmat, Px, Py):
  q = myQueue()
  visited = [False for i in range(n)]
  q.enQueue(Px)
  q.enQueue(-1) #using -1 in queue to distinguish between levels in BFS.
  visited[Px]=True
```

```
level=1;


while not q.isEmpty():
  v = q.deQueue()
  if (v == -1):
    level+=1
    if (not q.isEmpty()):
      q.enQueue(-1)
    continue
  for i in range(n):
    if(i==Py and Gmat[v][i] == 1):
      return level
    if(Gmat[v][i] and (not visited[i])):
      q.enQueue(i)
      visited[i]=True


return 0
```

Consider a system of `n` water tanks on a hill, connected via `m` pipes. Water can flow through these pipes only in one direction. We have a source of water that can be connected to only one of these water tanks. We need to find if there exists a master tank such that all the tanks in this group can be filled by connecting the water source to this master tank. The tanks are numbered from `1` to `n`.

Write a Python function `findMasterTank(tanks, pipes)` that accepts arguments `tanks` which is a list of tanks, and `pipes` which is a list of tuples that represents connectivity through pipes, between tanks. Each tuple `(i,j)` in `pipes` represents a pipe such that, water can flow from tank `i` to tank `j` but not vice versa. Your function should find a master tank and return the number representing it, else should return `0` if no master tank exists in the system. If there are more than one master tanks return any one of them. Try to implement an algorithm that executes in linear time ($O(n+m)$).

For e.g. In the graph `GT` below representing pipe connectivity between 4 water tanks, tank `1` is the only master tank. For this your function should return `1`.



`GT`

Sample input:

```
1 2 3 4 # Vertices
3       # Number of edges
1 2     # edge
2 3     # edge
2 4     # edge
```

Return:

```
1
```

from collections import deque

class myStack:

def __init__(self):

self.stack = deque()

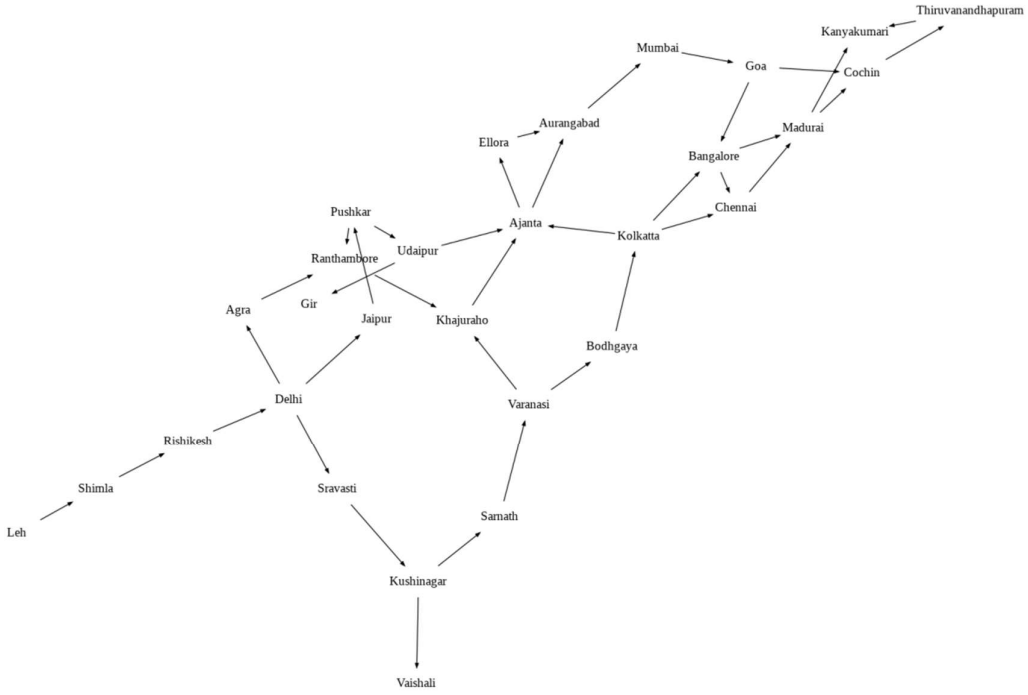
```
  def pop(self):
    return self.stack.pop()

  def push(self, x):
    return self.stack.append(x)

  def isEmpty(self):
    return False if self.stack else True

# Run depth first search starting from vertex t and mark all nodes that are visited.
def runDFSForTankT(tanks, GList, t, visited):
  s = myStack()
  s.push(t)
  visited[t] = True

  while not s.isEmpty():
    i = s.pop()
    for p in GList[i]:
      if not visited[p]:
        s.push(p)
        visited[p] = True;

# Brute force approach is to do a DFS for all vertices and check if all other vertices can be reached.
# Time complexity for this approach will be O(n(n+m)).
# Below function finds the master tank in O(n+m) time complexity. Here v is number of vertices and
e is number of edges.
def findMasterTank(tanks, pipes):
  # Create an adjacency list for graph representing the system of pipes and tanks.
  GList = {}
  for i in tanks:
    GList[i]=[]
```

```
 for (i,j) in pipes:
  GList[i].append(j)

 # Mark every tank not visited.
 visited = {t:False for t in tanks}

 lastVisited = tanks[0]
 # Traverse the tanks through depth first search method and keep track of last visited tank.
 for t in tanks:
  if not visited[t]:
   runDFSForTankT(tanks, GList, t, visited)
   lastVisited = t

 # Check if this last visited tank has paths to all other tanks in the system by doing another depth first search.
 visited = {t:False for t in tanks}
 runDFSForTankT(tanks, GList, lastVisited, visited)

 # Check visited to verify if all tanks are visited.
 for v in visited:
  if not visited[v]:
   return 0
 return lastVisited

 # Although we are calling DFS multiple times in a loop, the visited marking is common between all these calls. This ensures that we will traverse tanks only once, hence running the solution in linear time.
```

Long journey

A tourist wants to travel around India from north to south. He has a policy that he never travels back towards the north. Write a Python function `longJourney(AList)` to find him a route with which he can visit the maximum number of cities according to his policy, where `AList` represents a graph of cities and routes between them. Every edge in adjacency list `AList` is a feasible route between one city to another from north to south. The function should return a list in the order the cities are to be visited to visit maximum cities.

An example of cities and route between them(as edge) is shown below.



```
 'Sravasti': ['Kushinagar'],
 'Leh': ['Shimla'],
 'Sarnath': ['Varanasi'],
 'Delhi': ['Jaipur', 'Agra', 'Sravasti'],
 'Goa': ['Cochin', 'Bangalore'],
 'Kanyakumari': [],
 'Kushinagar': ['Sarnath', 'Vaishali'],
 'Khajuraho': ['Ajanta'],
 'Jaipur': ['Pushkar'],
 'Mumbai': ['Goa'],
 'Ajanta': ['Ellora', 'Aurangabad']}
```

Sample Output

```
['Leh', 'Shimla', 'Rishikesh', 'Delhi', 'Sravasti', 'Kushinagar', 'Sarnath',
'Varanasi', 'Bodhgaya', 'Kolkatta', 'Ajanta', 'Ellora', 'Aurangabad',
'Mumbai', 'Goa', 'Bangalore', 'Chennai', 'Madurai', 'Cochin',
'Thiruvanandhapuram', 'Kanyakumari']
```

```
# A Queue class to keep track of the visited nodes
class Queue:
  def __init__(self):
    self.queue = []

  def addq(self, v):
    self.queue.append(v)

  def delq(self):
    v = None
    if not self.isempty():
      v = self.queue[0]
      self.queue = self.queue[1:]
      return v

  def isempty(self):
    return self.queue == []
```

```
  def __str__(self):
    return str(self.queue)

# Dictionary inversion for d which has list as values
def dInv(d):
  d_ = {}
  if not isinstance(list(d.values())[0], list):
    for k, v in d.items():
      if v not in d_:
        d_[v] = []
      d_[v].append(k)
    return d_

  if isinstance(list(d.values())[0], list):
    for k, v in d.items():
      for v_ in v:
        if v_ not in d_:
          d_[v_] = []
        d_[v_].append(k)
    return d_

# Longest path function from the lecture
def longestpathlist(AList):
  indegree, lpath = {}, {}
  for u in AList:
    indegree[u], lpath[u] = 0, 0
  for u in AList:
    for v in AList[u]:
      indegree[v] = indegree[v] + 1

  zerodegreeq = Queue()
```

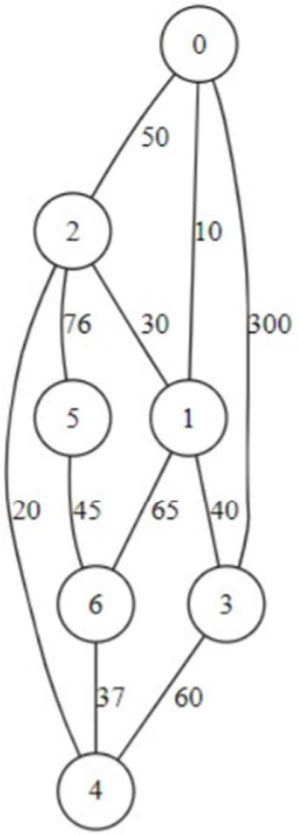
```
  for u in AList:
    if indegree[u] == 0:
      zerodegreeq.addq(u)

  while not zerodegreeq.isempty():
    j = zerodegreeq.delq()
    indegree[j] = indegree[j] - 1
    for k in AList[j]:
      indegree[k] = indegree[k] - 1
      lpath[k] = max(lpath[k], lpath[j] + 1)
      if indegree[k] == 0:
        zerodegreeq.addq(k)
  return lpath

def longJourney(AList):
  lpath = longestpathlist(AList) # longest path (dict)
  IAList = dInv(AList) # inverse adjacency list to get the reverse graph
  Ilj = dInv(lpath) # longest path as key and list of cities as values

  maxVal = max(lpath.values()) # value of longest path in which the city ends in the longest path
  prev = Ilj[maxVal][0] # last city
  path = [prev] # appending the last city
  for i in range(maxVal, -1, -1): # going backwards from last city to the first city in terms of longest path
    for p in Ilj[i]: # for every city p has the longest path i
      if p in IAList[prev]:
        path.append(p) # append to path if there is edge from p to prev in AList or edge from prev to p in IAList(reversed graph)
        prev = p
  return path[::-1] # reverse the path since it is computed from the last
```

An Internet service provider company wants to lay fiber lines to connect `n` cities labeled `0` to `n-1`. Write a function **FiberLink(distance_map)** that accepts a weighted adjacency list `distance_map` in the following format:-

```
distance_map = {
    source_index : [(destination_index,distance(km)),(destination_index,distance),..],
    ..
    ..
    source_index : [(destination_index,distance),(destination_index,distance),..]
}
```

The function returns the minimum length of fiber required to connect all `n` cities.

For the given graph



Sample Input

```
7 # number of vertices
[(0,1,10),(0,2,50),(0,3,300),(5,6,45),(2,1,30),(6,4,37),(1,6,65),(2,5,76),(1,3,40),(3,4,60),(2,4,20)] #edges
```

Output

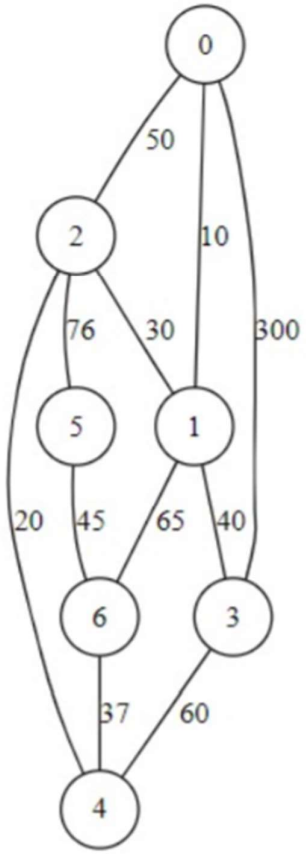
```
1 | 182
```

```
def kruskal(WList):
    (edges,component,TE)=([],{},[])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
        component[u] = u
    edges.sort()
    for (d,u,v) in edges:
        if component[u] != component[v]:
            TE.append((u,v))
            c = component[u]
        for w in WList.keys():
            if component[w] == c:
                component[w] = component[v]
    return(TE)


def FiberLink(distance_map):
    R = kruskal(distance_map)
    S = 0
    for e in R:
        for ed in distance_map[e[0]]:
            if ed[0]==e[1]:
                S += ed[1]
    return S
```

Write a Function **min_cost_walk(WList, S, D, V)** that accepts a weighted adjacency list `WList` for a undirected and connected graph. The function returns the minimum cost and walk route in the format `(minimum_cost, [walk_route])` from node **S** to node **D**, via node **V** (`S -> V -> D`), where the cost of a walk is the sum of weights of edges encountered on the route.

Note:- Node can be repeat in path.

For the given graph



Sample Input 1

```
7 # number of vertices
[(0,1,10),(0,2,50),(0,3,300),(5,6,45),(2,1,30),(6,4,37),(1,6,65),(2,5,76),
(1,3,40),(3,4,60),(2,4,20)] # edges
0 #S
4 #D
5 #V
```

Output

```
(198, [0, 1, 2, 5, 6, 4])
```

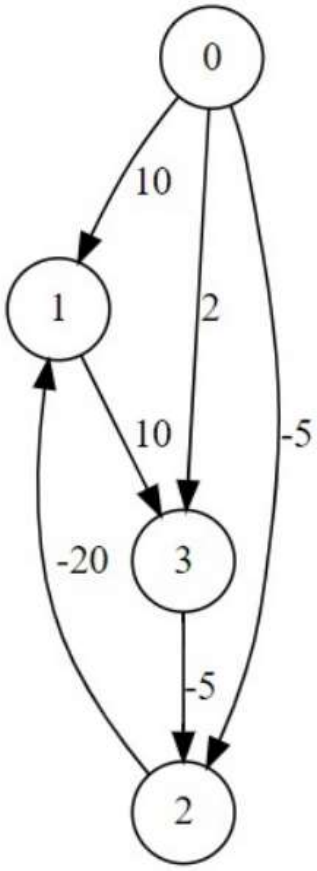
def dijkstra(WList,s):

```
    infinity = 1 + len(WList.keys())*max([d for u in WList.keys()for (v,d) in WList[u]])
    (visited,distance,prev) = ({},{},{})
    for v in WList.keys():
        (visited[v],distance[v],prev[v]) = (False,infinity,None)
    distance[s] = 0
    for u in WList.keys():
        nextd = min([distance[v] for v in WList.keys() if not visited[v]])
        nextvlist = [v for v in WList.keys() if (not visited[v]) and distance[v] == nextd]
        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
        for (v,d) in WList[nextv]:
            if not visited[v]:
                if distance[v] > distance[nextv]+d:
                    distance[v] = distance[nextv]+d
                    prev[v] = nextv
    return(distance,prev)


def min_cost_walk(WList,S, D, V):
    distance1,path1 = dijkstra(WList, S)
    distance2,path2 = dijkstra(WList, V)
    tot_dist = distance1[V] + distance2[D]
    Route_S_V = []
    Route_V_D = []
            # shortest route for S to V
    if distance1[V] != 0:
        dest = V
        while dest != S:
            Route_S_V = [dest] + Route_S_V
```

```
        for i,j in path1.items():
            if dest == i:
                dest = j
                break
    Route_S_V = [dest] + Route_S_V
        # shortest route for V to D
if distance2[D] != 0:
    dest = D
    while dest != V:
        Route_V_D = [dest] + Route_V_D
        for i,j in path2.items():
            if dest == i:
                dest = j
                break
    Route_V_D = [dest] + Route_V_D
Route_S_D = Route_S_V[:-1]+ Route_V_D
return (tot_dist,Route_S_D)
```

Write a function **IsNegativeWeightCyclePresent(WList)** that accepts a weighted adjacency list `WList` for a directed and connected graph and returns `True` if the graph has a negative weight cycle. Otherwise, return `False`.

For Given Graph



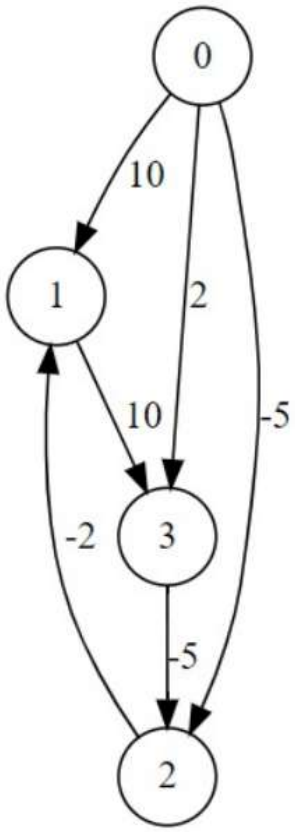
Sample Input

```
4 # number of vertices
[(0,1,10),(0,2,-5),(0,3,2),(3,2,-5),(2,1,-20),(1,3,10)] # edges
```

Output

```
True
```

For Given Graph



Sample Input

```
4
[(0,1,10),(0,2,-5),(0,3,2),(3,2,-5),(2,1,-2),(1,3,10)]
```

Output

```
False
```

```
def IsNegativeWeightCyclePresent(WList):
    s = 0
    infinity = 1 + len(WList.keys())*max([d for u in WList.keys() for (v,d) in WList[u]])
    distance = {}
    for v in WList.keys():
        distance[v] = infinity
    distance[s] = 0
    for i in WList.keys():
        for u in WList.keys():
```

```
        for (v,d) in WList[u]:
            distance[v] = min(distance[v], distance[u] + d)

    for u in WList.keys():
        for (v,d) in WList[u]:
            if (distance[u] + d < distance[v]):
                return True
    return False
```

Let `L` be a list with k elements, where each element is a sorted list of integers. Write a Python function **mergeKLists(L)** that merges all the lists into a single sorted list and returns it. Try to write a solution that runs in $O(n \log k)$ time complexity, where `n` is the total number of integers in all `k` lists combined.

```
def mergeKLists(L):
  #Complete this function
```

Sample Input

```
L = [ [2, 5, 9],
      [5, 23, 67, 212],
      [1, 10, 22]]
```

Sample Output

```
Return: [1, 2, 5, 5, 9, 10, 22, 23, 67, 212]
```

```
class min_heap:
  def __init__(self):
    self.size=0
    # The below is a list of tuples. In every tuple one value
    # is the element from list and the second value identifies the list.
    self.arr = []

  def isempty(self):
    return True if (self.size == 0) else False

  # Heapify element at index i
```

```
def heapify_down(self, i):
  n = self.size
  a = self.arr
  while (i<n):
    ss = l = 2 * i + 1
    r = 2 * i + 2
    # ss is smaller of left and right child
    if (l<n and r<n and a[l][0] > a[r][0]):
      ss = r
    if (ss<n and a[ss][0]<a[i][0]):
      a[i], a[ss] = a[ss], a[i]
      i = ss
    else:
      break

def delete_min(self):
  min = self.arr[0]
  last = self.arr.pop()
  #size of heap after pop operation will reduce by 1
  self.size -= 1
  if (self.size > 0):
    self.arr[0] = last
    self.heapify_down(0)
  return min

# Heapify last element in the heap
def heapify_up(self):
  i = self.size - 1
  while (i>0):
    parent = (i-1)//2
    if (self.arr[i][0] < self.arr[parent][0]):
```

```
            self.arr[i], self.arr[parent] = self.arr[parent], self.arr[i]
            i = parent
        else:
            break

    # Insert to min heap
    def insert_min_heap(self, x):
        self.arr.append(x)
        self.size += 1
        self.heapify_up()

def mergeKLists(L):
    k = len(L)
    h = min_heap()
    finalList = []

    # Insert first element from each k lists to heap.
    for i in range(k):
        tup = (L[i][0], i)
        h.insert_min_heap(tup)

    kPointers = [1 for i in range(k)] # As all 0th elements will be inseted into the heap of size k
    while (not h.isempty()):
        tup = h.delete_min()
        finalList.append(tup[0])
        listNumber = tup[1]
        if (kPointers[listNumber] < len(L[listNumber])):
            tup = (L[listNumber][kPointers[listNumber]], listNumber)
            h.insert_min_heap(tup)
            kPointers[listNumber] += 1
```

Write a Python function maxLessThan(root, K), that accepts the root node of the binary search tree and a number *K* and returns the maximum number less than or equal to *K* in the tree. If *K* is the less than every number in the tree return *None* . Each node in the tree is an object of class *Tree* , and the tree will have at least one node.

```
class Tree:
  #constructor
  def __init__(self, initval=None):
    self.value = initval
    if self.value:
      self.left = self.right = Tree()
    else:
      self.left = self.right = None
    return

  #Only empty node has value None
  def isempty(self):
    return(self.value == None)

  #Leaf nodes have both children empty
  def isleaf(self):
    return(self.value != None and self.left.isempty() and
self.right.isempty())

# Not a member of the class Tree.
def maxLessThan(root, K):
  # Complete this funtion
```

Sample Input: (Below elements are inserted in the same order in a BST)

```
45 22 57 12 33 73 55 2 80 62
58
```

Sample Output

```
57
```

Sample Input: (Below elements are inserted in the same order in a BST)

```
45 22 57 12 33 73 55 2 80 62
1
```

Sample Output

```
None
```

```
# Method 1
# Insert K in the binary search tree {O(log n)}, then do inorder traversal of the tree in a List L {O(n)}.
# Then search for K in the List L {O(n)} and print the elementleft to K. If K is found at index 0 print None.

# Method 2 (Iterative O(log n))
def maxLessThan(root, K):
  max = root.value
  temp = root
  while (not temp.isempty()):
    if (temp.value and K<temp.value):
      temp = temp.left
    elif (temp.value and K>=temp.value):
      max = temp.value
      temp = temp.right

  if (K >= max):
    return max
  else:
    return None
```

Given an array based min Heap `arr`, write a Python function **min_max(arr)** that to convert `arr` to a max Heap. The function should change the original `arr` to max heap. The expected time complexity is $O(n)$.

For e.g for the below min heap

```
3 5 9 6 8 20 10 12 18 9
```

the function should return a max heap like(or any other max heap with the same elements)

```
20 18 10 12 9 9 3 5 6 8
```

```
def heapify(A, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and A[largest] < A[l]:
        largest = l
    if r < n and A[largest] < A[r]:
        largest = r
    if largest != i:
        A[i], A[largest] = A[largest], A[i]
        heapify(A, n, largest)


def min_max(A):
    n = len(arr)
    for i in range(n//2,-1,-1):
        heapify(A,n,i)
```

For the given class, `AVLTree`, write a method `insert(self,v)` to create a height-balanced tree after insertion of the element `v`.

Sample Input

```
[1,2,3,4,5,6,7] #order of insertion
```

Output

```
[1, 2, 3, 4, 5, 6, 7] #inorder traversal
[4, 2, 1, 3, 6, 5, 7] #preorder traversal
[1, 3, 2, 5, 7, 6, 4] #postorder traveral
```

```
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = AVLTree()
            self.right = AVLTree()
```

```
            self.height = 1
            return
        if self.value == v:
            return
        if v < self.value:
            self.left.insert(v)
            self.rebalance()
            self.height = 1 + max(self.left.height, self.right.height)
        if v > self.value:
            self.right.insert(v)
            self.rebalance()
            self.height = 1 + max(self.left.height, self.right.height)

    def rebalance(self):
        if self.left == None:
            hl = 0
        else:
            hl = self.left.height
        if self.right == None:
            hr = 0
        else:
            hr = self.right.height
        if  hl - hr > 1:
            if self.left.left.height > self.left.right.height:
                self.rightrotate()
            if self.left.left.height < self.left.right.height:
                self.left.leftrotate()
                self.rightrotate()
            self.updateheight()
        if  hl - hr < -1:
            if self.right.left.height < self.right.right.height:
                self.leftrotate()
            if self.right.left.height > self.left.right.height:
                self.right.rightrotate()
                self.leftrotate()
            self.updateheight()

    def updateheight(self):
        if self.isempty():
            return
        else:
            self.left.updateheight()
            self.right.updateheight()
            self.height = 1 + max(self.left.height, self.right.height)
```

The government decides to construct a new railway station. Railway authorities provide a list of scheduled trains in a day with arrival time and departure time. Find the minimum number of platforms required for the railway station so that no train is kept waiting.

Write a function **minimum_platform(train_schedule)** that accepts a list of tuples `train_schedule`, where each tuple stores three values in the format `(train_no(int),Arrival_time('HH:MM'),departure_time('HH:MM'))`. The function should return the minimum number of platforms required for the railway station so that no train is kept waiting.

Sample Input

```
[(1,'09:00','09:10'),(2,'09:40','12:00'),(3,'09:50','11:20'),
(4,'11:00','11:30'),(5,'11:40','12:10'),(6,'12:05','19:00'),
(7,'12:06','13:00'),(8,'13:05','14:00'),(9,'14:05','15:00'),
(10,'18:00','20:00')]
```

Output

```
3
```

```
def minimum_platform(train_schedule):
    count = 1
    train_list = []
    for (i,j,k) in train_schedule:
        train_list.append((int(j.replace(':','')),int(k.replace(':','')),i))
    train_list.sort()
    train_at_plateform = []
    for train in train_list:

        t = len(train_at_plateform)-1
        while t >= 0:
            if train[0] > train_at_plateform[t][1]:
                train_at_plateform.pop(t)
            t = t-1
        t = len(train_at_plateform)-1
        while t >= 0:
            if train[0] < train_at_plateform[t][1]:
                t = t - 1
            elif train[1] > train_at_plateform[t][1]:
                train_at_plateform.pop(t)
                t = t - 1
        train_at_plateform.append(train)
        if len(train_at_plateform) > count:
            count = len(train_at_plateform)
    return count
```

A popular meeting hall in a city receives many overlapping applications to hold meetings. The manager wishes to satisfy as many customers as possible. Each application is a tuple `(id, start_day, end_day)` where `id`, `start_day` and `end_day` are the unique id assigned to the application, starting day of the meeting and ending day of meeting ends inclusive respectively. Write a function `no_overlap(L)` to return the list of customer ids whose applications are accepted that ensures optimal scheduling. Let `L` be a list tuples with `(id, start_day, end_day)`.

Sample Input

```
L = [
    (0, 1, 2),
    (1, 1, 3),
    (2, 1, 5),
    (3, 3, 4),
    (4, 4, 5),
    (5, 5, 8),
    (6, 7, 9),
    (7, 10, 13),
    (8, 11, 12)
]
```

Sample output

```
[0, 3, 6, 8]
```

```
def tuplesort(L, index):
    L_ = []
    for t in L:
        L_.append(t[index:index+1] + t[:index] + t[index+1:])
    L_.sort()

    L__ = []
    for t in L_:
        L__.append(t[1:index+1] + t[0:1] + t[index+1:])
    return L__

def no_overlap(L):
    sortedL = tuplesort(L, 2)
    accepted = [sortedL[0][0]]
    for i, s, f in sortedL[1:]:
        if s > L[accepted[-1]][2]:
            accepted.append(i)
    return accepted
```

Consider a list *L* of *n* integers sorted in ascending order. Write a Python

function findOccOf(L, x) to find the first and last occurrences of a number *x* in list *L*. Function should return the index of first and last occurrence of *x* as a tuple, for e.g. if *x* appears from index *3* to index *7* in list *L*, then return the tuple *(3, 7)*. If *x* is not in list *L* return *(None, None)*. The function should run in O(\log{n})*O*(log*n*) time.
Sample Input

3 3 5 5 5 5 6 6 6 6 6 6 9 9 9 9 10 10 11 13 14 14 14 14 14 14
5

Function should return tuple

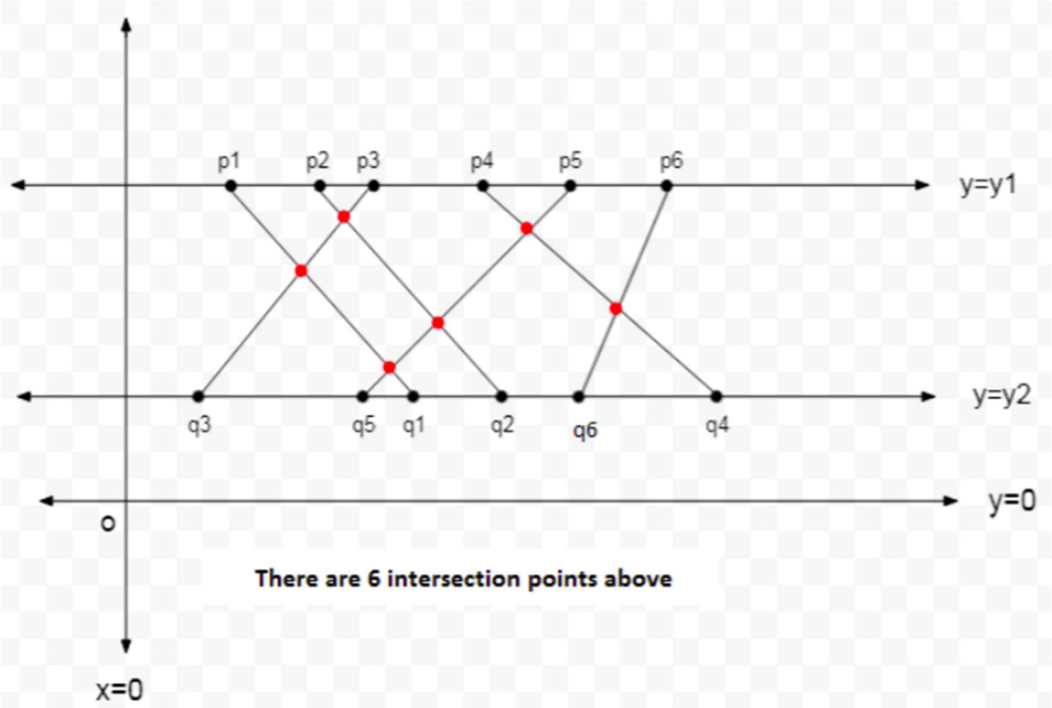
(2, 5)

```
import math
def findLeft(A, x, l, r):
  if (l>r):
    return None

  mid = (l+r)//2
  if (A[mid] == x):
    if (mid == 0 or A[mid] != A[mid-1]):
      return mid
    else:
      return findLeft(A, x, l, mid)
  elif (x > A[mid]):
    return findLeft(A, x, mid+1, r)
  else:
    return findLeft(A, x, l, mid-1)
```

```
def findRight(A, x, s, l, r):
  if (l>r):
    return None

  mid = math.ceil((l+r)/2)
  if (A[mid] == x):
    if (mid == s-1 or A[mid] != A[mid+1]):
      return mid
    else:
      return findRight(A, x, s, mid, r)
  elif (x > A[mid]):
    return findRight(A, x, s, mid+1, r)
  else:
    return findRight(A, x, s, l, mid-1)


def findOccOf(L, x):
  s = len(L)
  if (s<1):
    return None

  start = findLeft(L, x, 0, s-1)
  end = findRight(L, x, s, 0, s-1)
  return (start, end)
```

In a 2-dimensional space, given two sets of `n` points `S1 = [p1, p2, ..,pn]` and `S2=[q1, q2, ...,qn]` on two parallel lines `y=y1` and `y=y2` respectively. Every point `pi` in `S1` is connected to every corresponding point `qi` in `S2` through a line segment. `X1` and `X2` are two lists that contains x coordinates of the points in set `S1` and `S2` respectively.

Write a Python function **countIntersection(X1, X2)** that accepts the two Lists `X1` and `X2` as described above and returns the number of intersection points where the line segments intersect. The function should run in $O(n \log n)$ time.



Sample Input

```
8 22 10 4 17 14  #x coordinates of points on line 1
15 18 3 12 10 23  #x coordinates of points on line 2
```

Sample Output

```
6
```

```
def mergeAndCount(A,B):
  (m,n) = len(A), len(B)
  (C, i, j, k, count) = ([], 0, 0, 0, 0)
  while k< m+n:
    if i == m:
      C.append(B[j])
      (j,k) = (j+1, k+1)
    elif j == n:
      C.append(A[i])
      (i,k) = (i+1, k+1)
    elif A[i] < B[j]:
```

```
    C.append(A[i])
    (i,k) = (i+1, k+1)
   else:
    C.append(B[j])
    (j, k, count) = (j+1, k+1, count+(m-i))
  return (C,count)

def sortAndCount(A):
  n = len(A)

  if n <= 1:
   return(A,0)

  (L,countL) = sortAndCount(A[:n//2])
  (R,countR) = sortAndCount(A[n//2:])

  (B,countB) = mergeAndCount(L,R)

  return(B,countL+countR+countB)

def countIntersection(X1, X2):
  # Sort according to one points while keeping the matching of points in X1 to X2
  combined = [(X1[i], X2[i]) for i in range(0, len(X1))]
  combined.sort()
  X1, X2 = zip(*combined)

  # Now we just need to count the inversions in X2 for number of intersection points.
  return sortAndCount(X2)[1]
```

Consider a set of `n` points `Points` given as a list of tuples, where each tuple represents a point in a 2 dimensional space with first element of tuple as x-coordinate and second element as y-coordinate. Write a Python function **minDistance(Points)** that returns the distance (rounded to 2 decimal places) between the closest pair of points `(pi, pj)`, in the set of points `Points`.

$n >= 2$

No two points have same x coordinate.

No two points have same y coordinate.

Solution should run in $O(n \log n)$ time.

Sample Input

```
[(2, 15), (40, 5), (20, 1), (21, 14), (1,4), (3, 11)]
```

Sample Output

```
4.12
```

```
import math
```

```
# Returns eucledian disatnce between points p and q
def distance(p, q):
  return math.sqrt(math.pow(p[0] - q[0],2) + math.pow(p[1] - q[1],2))

def minDistanceRec(Px, Py):
  s = len(Px)
  # Given number of points cannot be less than 2.
  # If only 2 or 3 points are left return the minimum distance accordingly.
  if (s == 2):
    return distance(Px[0],Px[1])
  elif (s == 3):
    return min(distance(Px[0],Px[1]), distance(Px[1],Px[2]), distance(Px[2],Px[0]))

  # For more than 3 points divide the poitns by point around median of x coordinates
  m = s//2
  Qx = Px[:m]
  Rx = Px[m:]
  xR = Rx[0][0]    # minimum x value in Rx

  # Construct Qy and Ry in O(n) rather from Py
  Qy=[]
  Ry=[]
  for p in Py:
    if(p[0] < xR):
      Qy.append(p)
    else:
      Ry.append(p)

  # Extract Sy using delta
  delta = min(minDistanceRec(Qx, Qy), minDistanceRec(Rx, Ry))
  Sy = []
  for p in Py:
    if abs(p[0]-xR) <= delta:
      Sy.append(p)

  #print(xR,delta,Sy)
  sizeS = len(Sy)
  if sizeS > 1:
      minS = distance(Sy[0], Sy[1])
      for i in range(1, sizeS-1):
          for j in range(i, min(i+15, sizeS)-1):
              minS = min(minS, distance(Sy[i], Sy[j+1]))
      return min(delta, minS)
  else:
      return delta

def minDistance(Points):
  Px = sorted(Points)
  Py = Points
  Py.sort(key=lambda x: x[-1])
  #print(Px,Py)
```

```
return round(minDistanceRec(Px, Py), 2)
```

Write a Python function **MoM7Pos(arr)** that accepts a list `arr` of integers(not necessarily distinct) and computes the median of medians(of blocks) `M` dividing the list `arr` into blocks of 7 and returns the position of `M` in `arr` if it were sorted. If `M` is repeated more than once in the list `arr` return the index of first occurrence of `M` in `arr` if it were sorted.

For simplicity the size of `arr` will be a multiple of 7. Your solution should run in $O(n)$ time.

Sample Input

```
44 9 31 12 15 98 48 45 13 75 23 6 35 74
```

Sample Output

```
7
```

```
def partitionPos(arr, pivot):
  arr[pivot], arr[0] = arr[0], arr[pivot]
  l = 1
  r = len(arr)-1
  while (l<r):
    while(arr[l] < arr[0]):
      l+=1
    while(arr[r]>=arr[0]):
      r-=1
    arr[l], arr[r] = arr[r], arr[l]

  return l-1

def MoM7(arr):
  if len(arr) <= 7:
    arr.sort()
    return(arr[len(arr)//2])

  # Construct list of block medians
  M = []

  for i in range(0, len(arr), 7):
    X = arr[i:i+7]
    X.sort()
    M.append(X[len(X)//2])

  return(MoM7(M))

def MoM7Pos(arr):
  mom = MoM7(arr)
  return partitionPos(arr, arr.index(mom))
```

A thief robbing a store and can carry a maximum weight of `W` in his bag. There are `n` items in the store with weights $\{w_1, w_2, w_3, \ldots, w_n\}$ and corresponding values $\{v_1, v_2, v_3, \ldots, v_n\}$. What items should he select to get maximum value? He cannot break an item, either he picks the complete item or doesn't pick it.

Write a function **MaxValue(Items, W)** that accepts a dictionary `Items`, where the key of the dictionary represents the item number (1 to n) and the corresponding value is a tuple `(weight of item, value of item)`. The function accepts one more integer `W`, which represents the maximum weight capacity of the bag. The function returns the total value of all selected items, which is maximum in all possible selection.

Example Input

```
8 # W - Maximum weight capacity of bag
{1: (2,10),2: (3,20),3: (4,40)} # Items
```

Output

```
60 #total value is 60 which is maximum.
```

Explanation

The thief can pick items 2 and 3 where the total weight of picked items is 3 + 4 = 7 which is less than the maximum capacity of the bag, but he gets maximum value (60) for this selection.

Sample Input

```
30
{1:(10,2), 2:(15,5), 3:(6,8), 4:(9,1)}
```

Output

```
14
```

```
import numpy as np
def MaxValue(Items,W):
    m,n=len(Items),W
    c=np.zeros((m+1,n+1))
    for i in range(1,m+1):
        for w in range(1,n+1):
            if (Items[i][0]>w):
                c[i][w]=c[i-1][w]
            else:
                c[i][w]=max(c[i-1][w] , Items[i][1]+c[i-1][w-Items[i][0]])
    return int(c[m][n])
```

You're given a list of tuples `Activity` for `n` activities, where in each tuple `(activity_name, S, F, P)`, activity `activity_name` is scheduled to be done from start time `S` to finish time `F` and obtains a profit of `P` after the finish.

Find out the maximum profit you can obtained by scheduled activities, but no two activities should be in the subset with overlapping of time frame. If you choose an activity that finishes at time `x` then another activity can be started at time `x`, not before that.

Write a function **MaxProfit(Activity)** that accepts a list of tuples `Activity` for `n` activities and returns the value of maximum profit that can be obtained by scheduled activities.

Sample Input

```
[('A',1,2,40),('B',3,4,5),('C',0,7,6),('D',1,2,3),('E',5,6,8),('F',5,9,2),
('G',10,11,9),('H',0,11,35)]
```

Output

```
62
```

Explanation

Activity schedule `[A, B, E, G]` gives a profit of 62 which is the maximum in all possible schedules

```
def tupsort(Activity, index):
    L1=[]
    for t in Activity:
        L1.append(t[index:index+1] + t[:index]+t[index+1:])
    L1.sort()
    L2=[]
    for t in L1:
        L2.append(t[1:index+1]+t[0:1]+t[index+1:])
    return L2

def MaxProfit(Activity):
    n=len(Activity)
    act=tupsort(Activity,2)
    Maxprofit=[]
    for i in act:
        Maxprofit.append(i[3])
    for i in range(1,n):
        for j in range(0,i):
            if (act[i][1]>=act[j][2] and Maxprofit[i]<=Maxprofit[j]+act[i][3]):
                Maxprofit[i]=Maxprofit[j]+act[i][3]
    return max(Maxprofit)
```

Write a recursive function `constructWord(word, wordList)` which returns a list of lists in which all the ways `word` can be constructed from the elements of `wordList`. Return an empty list if there are no possible ways to construct `word` with `wordList`. Reusing elements of `wordList` is allowed.

Note: Write an efficient solution to pass all test cases.

Sample Input

```
constructWord('apple', ['ap', 'pple', 'app', 'apl', 'appl', 'le', 'ple'])
```

Sample Output

```
[['ap', 'ple'], ['app', 'le']]
```

```
memo = {}
def constructWord(word, wordList):
    if word == '':
        return [[]]
    if word in memo.keys():
        return memo[word]
    totalwordlist = []
    for subword in wordList:
        if subword == word[:len(subword)]:
            subwordList = constructWord(word[len(subword):], wordList)
            totalwordlist.extend([[subword] + lst for lst in subwordList])
    memo[word] = totalwordlist
    return totalwordlist
```

Given a 2-D matrix `M` where each cell `M[i][j]` contains some number of coins, find a path to collect maximum coins from cell `(x1, y1)` to cell `(x2, y2)` in matrix `M` where `x2 >= x1` and `y2 >= y1`. You can only travel one step right or one step down in each move.

Write a function **MaxCoinsPath(M,x1,y1,x1,y1)** that accepts a matrix `M` , index (x1,y1) of the source cell and index (x2,y2) of the destination cell in matrix `M`. The function should return the maximum number of coins collected across all paths from source to destination.

Sample Input

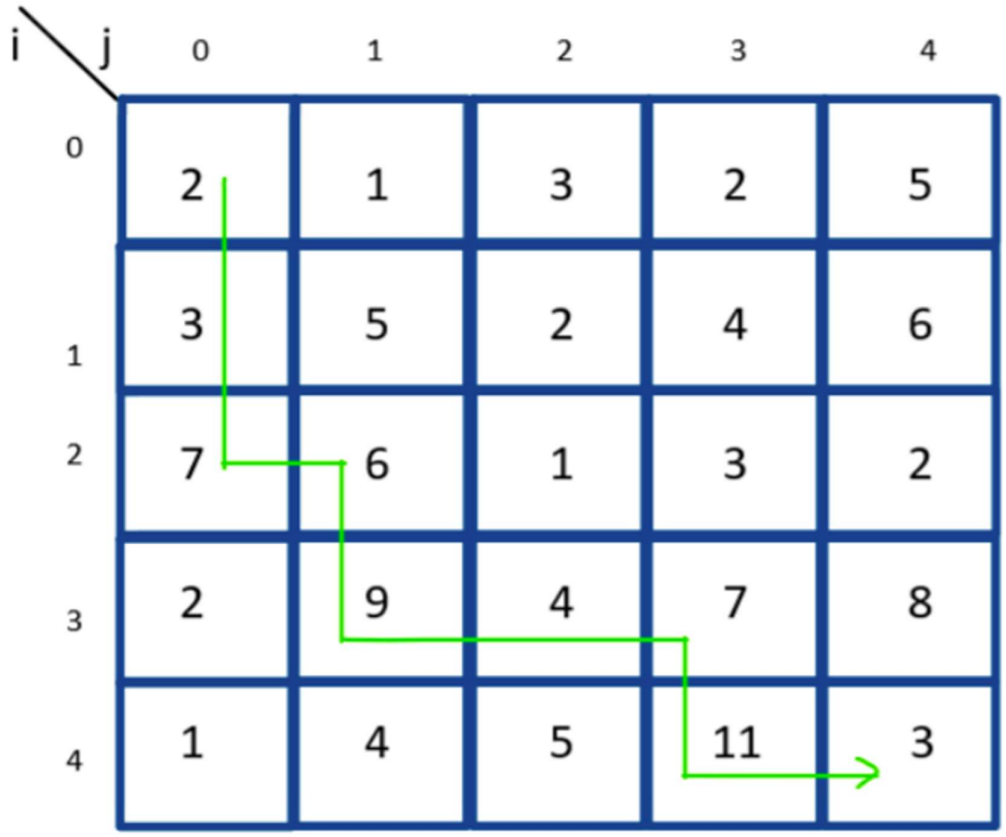
```
[[2,1,3,2,5],[3,5,2,4,6],[7,6,1,3,2],[2,9,4,7,8],[1,4,5,11,3]] # Matrix M
(0,0) #Source (x1,y1)
(4,4) #Destination (x2,y2)
```

Output

```
52
```

Explanation

2 + 3 + 7 + 6 + 9 + 4 + 7 + 11 +3 = 52, which is maximum in all paths from (1,1) to (4,4).




```
def MaxCoinPath(M,x1,y1,x2,y2):
    MCP=[]
    # Create matrix same size of M and initialized with 0
```

```
    for i in range(len(M)):
        L = []
        for j in range(len(M[0])):
            L.append(0)
        MCP.append(L)
    # Initialize x1 row and y1 coloumn
    MCP[x1][y1] = M[x1][y1]
    for i in range(y1+1,len(M[0])):
        MCP[x1][i]= MCP[x1][i-1] + M[x1][i]
    for i in range(x1+1,len(M)):
        MCP[i][y1]= MCP[i-1][y1] + M[i][y1]
            # calculate value for each cell
    for i in range(x1+1,len(M)):
        for j in range(y1+1,len(M[0])):
            MCP[i][j] = max(MCP[i-1][j],MCP[i][j-1]) + M[i][j]
    # return max coin value at x2,y2
    return MCP[x2][y2]
```

Longest Decreasing Sequence

Longest Decreasing Sequence (LDS) is in which the value gets strictly decreasing over the sequence. For example, in `[5, 4, 7, 1]`, `[5, 4, 1]` is a longest decreasing sequence.

Write a function `LDS(L)` to return a list of longest decreasing sequence. If more than one LDS is present in the list `L` then return any one of LDS.

Sample Input

```
1 L = [0, 2, 7, 3, 7, 3, 8, 2, 1, 0]
```

Sample Output

```
1 [7, 3, 2, 1, 0]
```

```
def LDS(L):
    n = len(L)
    LDSCount = [1]*n # LDS with respect to the index
    prev = [None]*n # previous value with respect to the index
    for i in range(n):
        preMax = L[0]
        for j in range(i):
            if L[j] > L[i] and LDSCount[j] > preMax:
                preMax, prev[i] = LDSCount[j], j
        LDSCount[i] = 1 + preMax # updating LDSCount
    mx = max(LDSCount) # count of LDS
    mxi = LDSCount.index(mx) # index of LDS

    # backtracking to get the sequence
    seq = []
    while mxi != None:
        seq.append(L[mxi])
```

```
    mxi = prev[mxi]
  return seq[::-1]
```

For a pattern `p` some positive integer `x`, p^x refers to `x` repetitions of `p`, for example ab^3 is `ababab` and abc^4 is `abcabcabcabc`.

Write a Python function **findContinuousRepetitions(t, p)** which accepts a text `t` and a pattern `p` and returns the maximum value of `x` such that p^x is a substring of `t`.

Sample Input

```
abcdababababcdabab # text t
ab                 # pattern p
```

Sample Output

```
3                  #as ababab is a substring of t
```

Sample Input

```
abaaabaaaaaacdabab # text t
aa                 # pattern p
```

```
def findContinuousRepetitions(t, p):
  last = {} # Preprocess
  m = len(p)
  for i in range(m):
    last[p[i]] = i
  poslist, i, count, maxR = [], 0, 0, 0 # Loop

  while i <= (len(t)-m):
    matched,j = True,len(p)-1
    poslist.append(i)
    while j > 0 and matched:
      if t[i+j] != p[j]:
        matched = False
        count = 0
      j = j - 1
    if matched:
      count += 1
      if (count > maxR): maxR = count
      i = i + m
    else:
      j = j + 1
      if t[i+j] in last.keys():
        i = i + max(j-last[t[i+j]],1)
      else:
        i = i + j + 1
  return maxR
```

Write a Function `FindPattern(T,P)` which accepts a string `T` and a string `P` where `length(P) <= length(T)`. In pattern `P`, `$` is a special symbol that represents a single character, that will match any character. The function returns a list of tuples in the format `[(start matched index of pattern in T, matched pattern),...]`.

Sample Input 1

```
abcabedsgababcjigdabjhtadce #T
$ab$ #P
```

Output 1

```
[(2, 'cabe'), (8, 'gaba'), (10, 'babc'), (17, 'dabj')] #matched pattern
```

Sample Input 2

```
abcabedsgababcjigdabjhtadce
ab$$$
```

Output 2

```
[(0, 'abcab'), (3, 'abeds'), (9, 'ababc'), (11, 'abcji'), (18, 'abjht')]
```

```
def FindPattern(T,P):
    last = {} # Preprocess
    for i in range(len(P)):
        if P[i] != '$':
            last[P[i]] = i
    poslist=[]
    i = 0 # Loop
    while i <= (len(T)-len(P)):
        matched,j = True,len(P)-1
        while j >= 0 and matched:
            if P[j] != '$':
                if T[i+j] != P[j]:
                    matched = False
            j = j - 1
        if matched:
            poslist.append((i,T[i:i+len(P)]))
            i = i + 1
        else:
            j = j + 1
            if T[i+j] in last.keys():
                i = i + max(j-last[T[i+j]],1)
            else:
                i = i + 1
    return(poslist)
```

Write a function `MakePalindrome(s)` that accepts a non-empty string `s` of letters in lower case and without spaces. The function returns the smallest string that converts `s` into a palindrome by adding it on the front of `s`. If input string `s` is already a palindrome, then return `None`. Try to give an $O(n)$ solution.

Sample Input 1

```
abcdc #s
```

Output

```
cdcb # cdcb + abcdc = cdcbabcdc which is palindrome
```

Sample Input 2

```
abcdcba
```

Output

```
None
```

Sample Input 3

```
abcdefghijk
```

Output

```
kjihgfedcb
```

```
def kmp_fail(p):
# Initialize
   m = len(p)
   fail = [0 for i in range(m)]
# Update
   j,k = 1,0
   while j < m:
      if p[j] == p[k]: #k+1 chars match
         fail[j] = k+1
         j,k = j+1,k+1
      elif k > 0: #find shorter prefix
         k = fail[k-1]
      else: #no match found at j
         j = j+1
   return(fail[-1])

def MakePalindrome(s):
   if len(s)==0:
      return s
   m = kmp_fail(s + '#' + s[::-1])
```

```
    if m == len(s):
        return None
    else:
        return s[m:][::-1]
```

MOCK:
Mock PP1:

Swapping Min-max

You are given two lists `a` and `b` of `n` positive integers each. You can apply the following **swap** operation to them any number of times:

Select an index i $(1 \leq i \leq n)$ and swap a_i with b_i (i.e. a_i becomes b_i and vice versa).

Write a function **minmax(a,b)** which takes two lists a and b of size `n` as inputs and returns an integer, which is the minimum possible value of $max(a_1, a_2, \ldots, a_n) \times max(b_1, b_2, \ldots, b_n)$ you can get after applying the **swap** operation any number of times (possibly zero).

Example

Consider the lists - `a = [1,2,6,5,1,2]` and `b = [3,4,3,2,2,5]`
In this case, you can apply the **swap** operation at indices `1` and `5`, then `a = [1,4,6,5,1,5]` and `b = [3,2,3,2,2,2]` and
$max(1, 4, 6, 5, 1, 5) \times max(3, 2, 3, 2, 2, 2) = 6 \times 3 = 18$

Sample Input 1

```
[1,2,6,5,1,2]
[3,4,3,2,2,5]
```

Output

```
18
```

Sample Input 2

```
def minmax(a,b):
    maxa,maxb = a[0],b[0]
    for i in range(1,len(a)):
        cmaxa = max(maxa,a[i])
        cmaxb = max(maxb,b[i])
        p = cmaxa*cmaxb
        swapcmaxa = max(b[i],maxa)
        swapcmaxb = max(a[i],maxb)
        pswap = swapcmaxa*swapcmaxb
        if p < pswap:
            maxa = cmaxa
            maxb = cmaxb
        else:
```

```
        maxa = swapcmaxa
        maxb = swapcmaxb
    return min(p,pswap)



a = eval(input())
b = eval(input())
print(minmax(a,b))
```

Mock PP2:

Peak in Unimodal

Consider an list `A` with n entries, with each entry holding a distinct number. The sequence of values `A[0], A[1], . . . , A[n-1]` is unimodal: For some index `p` between 0 and $n-1$, the values in the array entries increase up to position `p` in `A` and then decrease the remainder of the way until position `n-1`. (Assume length of `A` is at least 3)

Write a function `peak_unimodal`, that takes a list `A` as input and returns the index of the peak in `A` in $O(logn)$ time

Sample Input 1

```
[2, 3, 4, 21, 43, 52, 51, 18, 11, 9, 6, 5, 1]
```

Output

```
5
```

Sample Input 2

```
[8,9,3]
```

Output

```
1
```

```
def peak_unimodal(A):
    l = 1
    h = len(A) -2
    while l <= h:
        m = (l+ h )//2
        if A[m-1]<=A[m]>=A[m+1]:
            return m
        elif A[m+1]> A[m]:
            l = m+1
        else:
            h = m-1
```

Mock PPA4:

Domino Solitaire

(Indian National Olympiad in Informatics, 2008)

In Domino Solitaire, you have a grid with two rows and many columns. Each square in the grid contains an integer. You are given a supply of rectangular 2 × 1 tiles, each of which exactly covers two adjacent squares of the grid. You have to place tiles to cover all the squares in the grid such that each tile covers two squares and no pair of tiles overlap. The score for a tile is the difference between the bigger and the smaller number that are covered by the tile. The aim of the game is to maximize the sum of the scores of all the tiles. Here is an example of a grid, along with two different tilings and their scores.

8	6	2	3
9	7	1	2

Tiling 1

8	6	2	3
9	7	1	2

Score 12

Tiling 2

8	6	2	3
9	7	1	2

Score 6

The score for Tiling 1 is 12 = (9−8)+(6−2)+(7−1)+(3−2) while the score for Tiling 2 is 6 = (8−6)+(9−7)+(3−2)+(2−1). There are other tilings possible for this grid, but you can check that Tiling 1 has the maximum score among all tilings. Your task is to read the grid of numbers and compute the maximum score that can be achieved by any tiling of the grid.

Solution hint

Recursively find the best tiling, from left to right. You can start the tiling with one vertical tile or two horizontal tiles. Use dynamic programming to evaluate the recursive expression efficiently.

Input format

The first line contains one integer N, the number of columns in the grid. This is followed by 2 lines describing the grid. Each of these lines consists of N integers, separated by blanks.

Output format

A single integer indicating the maximum score that can be achieved by any tiling of the given grid.

Test Data: For all inputs, $1 \leq N \leq 10^5$. Each integer in the grid is in the range $\{0, 1, \ldots, 10\}$.

Sample Input:

```
4
8 6 2 3
9 7 1 2
```

Sample Output:

```
12
```

```
N = int(input())
R1 = list(map(int,input().split()))
```

```
R2 = list(map(int,input().split()))
dp = [0]*(N-1)+[abs(R1[-1]-R2[-1])] + [0]
k = N-2
for i in range(N-2,-1,-1):
   dp[k] = max(abs(R1[i]-R2[i])+dp[i+1],dp[i+2]+abs(R1[i]-R1[i+1])+abs(R2[i]-R2[i+1]))
   k -= 1
print(dp[0])
```

Shortest Circular Route

A traveler made a travel plan which starts from city `S`. Due to time limitations, he decided to choose the shortest circular route that returns to the starting city `S` without using any road twice in the route. The route need not visit all cities.

Write a Function **shortestCircularRoute(WList, S)** that accepts a weighted adjacency list `WList` for the connected two-way road network of `n` cities, labeled `0` to `n-1` and another parameter `S` which represents the start city. The function returns the total distance of the shortest circular route from city `S`.
Note: You are guaranteed that there is always at least one circular route.

Hint: Observe that a circular route from S consists of a road from S to a neighbour U followed by a path back from U to S that does not use the road (U, S).

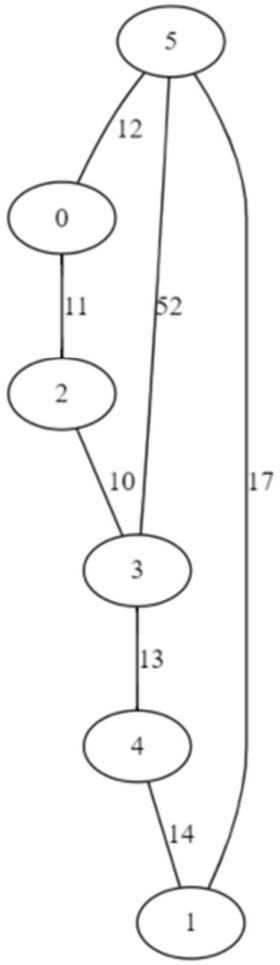
Format of WList

```
WList = {
    source_index : [(destination_index,distance),
(destination_index,distance),..],
    ..
    ..
    source_index : [(destination_index,distance),
(destination_index,distance),..]
}
```

Sample Input 1

```
6 #n- number of nodes(cities) labeled 0 to 5
[(2,0,11),(5,0,12),(5,3,52),(5,1,17),(4,1,14),(3,4,13),(2,3,10)] # edges (roads between city with distance)
0 #S source index (start city)
```

Graph representation of input 1



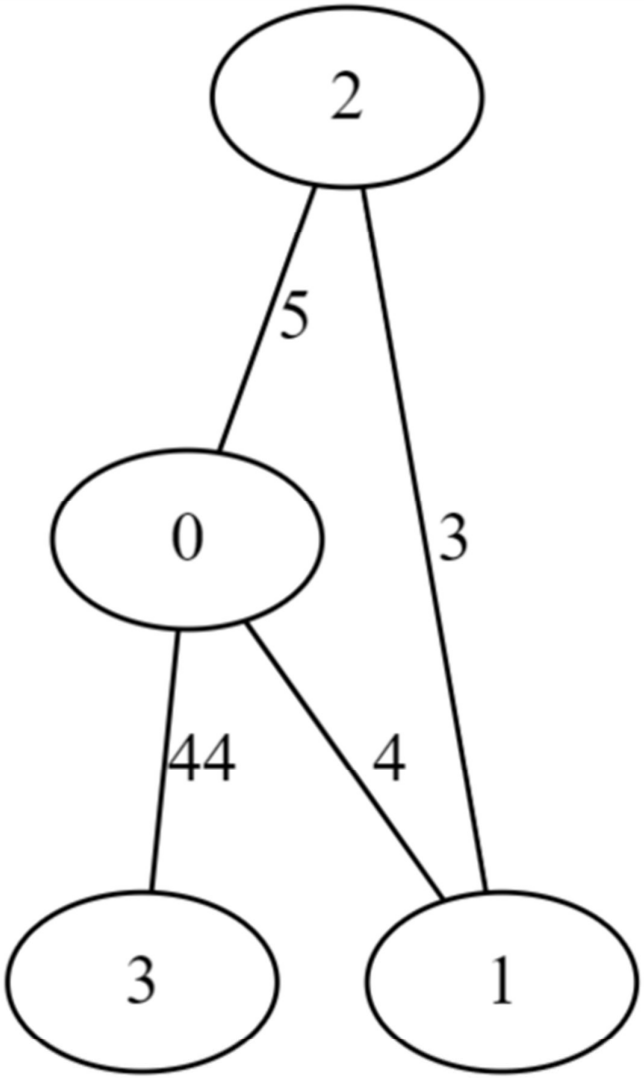
Output

```
77 #total distance of Shortest circular route 0 - 2 - 3 - 4 - 1 - 5 - 0
```

Sample Input 2

```
4 #n - number of nodes(cities) labeled from 0 to 3
[(2,0,5),(0,1,4),(1,2,3),(0,3,44)] # edges (roads between city with distance)
0 #S - source index (start city)
```

Graph representation of input 2



Output

```
12 #total distance of shortest circular route 0 - 1 - 2 - 0
```

```
def shortestCircularRoute(Wlist, s):
    edgesofs = Wlist[s]
    #mind = 10000
    sums = []
    for (a,b) in edgesofs:
        d = b
        source = a
```

```
        for (c,e) in Wlist[source]:
            if c == s:
                Wlist[source].remove((c,e))
        distance = dijkstra(Wlist, source)
        sum = distance[s] + d
        sums.append(sum)
        sum = 0
    return min(sums)
```

LCW:

```
def LCW(s1,s2):
    import numpy as np
    (m,n) = (len(s1),len(s2))
    lcw = np.zeros((m+1,n+1))
    maxw = 0
    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if s1[r] == s2[c]:
                lcw[r,c] = 1 + lcw[r+1,c+1]
            else:
                lcw[r,c] = 0
            if lcw[r,c] > maxw:
                maxw = lcw[r,c]
    return maxw
```

LCS:

```
def LCS(s1,s2):
    import numpy as np
    (m,n) = (len(s1),len(s2))
    lcs = np.zeros((m+1,n+1))
```

```
  for c in range(n-1,-1,-1):
    for r in range(m-1,-1,-1):
      if s1[r] == s2[c]:
        lcs[r,c] = 1 + lcs[r+1,c+1]
      else:
        lcs[r,c] = max(lcs[r+1,c], lcs[r,c+1])
  return lcs[0,0]
```

ED

```
def ED(u,v):
  import numpy as np
  (m,n) = (len(u),len(v))
  ed = np.zeros((m+1,n+1))
  for i in range(m-1,-1,-1):
    ed[i,n] = m-i
  for j in range(n-1,-1,-1):
    ed[m,j] = n-j
  for j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        ed[i,j] = ed[i+1,j+1]
      else:
        ed[i,j] = 1 + min(ed[i+1,j+1], ed[i,j+1], ed[i+1,j])
  return(ed[0,0])
```

MULTIPLICATION: (dp)

```
def MM(dim):
  n = dim.shape[0]
  C = np.zeros((n,n))
  for i in range(n):
    C[i,i] = 0
  for diff in range(1,n):
```

```
    for i in range(0,n-diff):
        j = i + diff
        C[i,j] = C[i,i] + C[i+1,j] + dim[i][0] * dim[i+1][0] * dim[j][1]
        print(C)
        for k in range(i+1, j+1):
            C[i,j] = min(C[i,j],C[i,k-1] + C[k,j] + dim[i][0] * dim[k][0] * dim[j][1])
        print(C)
return(C[0,n-1])
```