Question 1

1. Let the A has won 'i' games and B has won 'j' games
2. Now Probability of A winning be defined by $P(i, j)$
3. There are two possibilites for the next game either A could win or B could win.
4. The Probability of A winning the entire game (N wins) given A has won the next match is $0.5 * P(i + 1, j)$
5. The Probability of A winning the entire game (N wins) given B has won the next match is $0.5 * P(i, j + 1)$
6. Either of the above cases 4 or 5 could happen hence the probability is $0.5 * P(i + 1, j) + 0.5 * P(i, j + 1)$
7. We solve this recursively as below and store the probability in a dictionary with (i,j) as key so that we do not recalculate the combination again

```
probDictForWins = {}
probAWin = 0.5
probBWin = 0.5
def prob(AWinsTillNow,BWinsTillNow,NForMatchWin):

    if AWinsTillNow == NForMatchWin:
        return 1
    elif BWinsTillNow == NForMatchWin:
        return 0

    else:
        if (AWinsTillNow,BWinsTillNow) not in probDictForWins: ## if not already 
calculated

            probAWinsGameGivenAWinsNextMatch = 
probAWin*prob(AWinsTillNow+1,BWinsTillNow,NForMatchWin)
            probAWinsGameGivenBWinsNextMatch = 
probBWin*prob(AWinsTillNow,BWinsTillNow+1,NForMatchWin)

            probDictForWins[(AWinsTillNow,BWinsTillNow)] = 
probAWinsGameGivenAWinsNextMatch + probAWinsGameGivenBWinsNextMatch
            return probDictForWins[(AWinsTillNow,BWinsTillNow)]
        else:

            return probDictForWins[(AWinsTillNow,BWinsTillNow)]
```

In [1]:

```
probDictForWins = {}

def prob(AWinsTillNow,BWinsTillNow,NForMatchWin):
    probAWin = 0.5
    probBWin = 0.5
    if AWinsTillNow == NForMatchWin:
        return 1
    elif BWinsTillNow == NForMatchWin:
        return 0

    else:
        if (AWinsTillNow,BWinsTillNow) not in probDictForWins:
            probAWinsGameGivenAWinsNextMatch = probAWin*prob(AWinsTillNow+1,BWinsTillNow,NF
            probAWinsGameGivenBWinsNextMatch = probBWin*prob(AWinsTillNow,BWinsTillNow+1,NF

            probDictForWins[(AWinsTillNow,BWinsTillNow)] = probAWinsGameGivenAWinsNextMatch
            return probDictForWins[(AWinsTillNow,BWinsTillNow)]
        else:

            return probDictForWins[(AWinsTillNow,BWinsTillNow)]

prob(9,7,10)
```

Out[1]:

```
0.875
```

Question 2

```
Algorithm:

MaxSubset[i,sum] = MaxSubset[i-1, sum-A[i]]     if A[i] <= sum
                   MaxSubset[i-1, sum]          else


The algorithm is O(NT) because we keep storing the sub calculations and the same
combination of sum and index will not be calculated twice

i.e 't' the sum can in worst case be all values from 1 to T
    and each index combination with this is stored
    and when the sub problem leads to any of this combination, its never recacluated.
    Hence at Max there are N*T combinations
```

In [2]:

```
##code (pseudo code)
allElems = [2,3,4,5,7,2]
subsetDict = {}
indexPath = []

def findSubset(index,totalSum,allElems):

    if index < 0 : ## if index in recursion turns reduces to negative , there are no more e
        return False

    elif index >=0 and totalSum == 0  : ## if rest of totalSum reduces to zero and the inde

        return True

    if totalSum < 0 : ## when the element chosen is greater than the required sum itself
        return False


    if (index,totalSum) not in subsetDict: ##storing results in a dictinoary for caching ar
        subsetDict[(index,totalSum)] = findSubset(index-1,
                                                  totalSum-allElems[index],
                                                  allElems)

    if subsetDict[(index,totalSum)] == True :
        indexPath.append(index) ## tracking the solutions

    else:
        subsetDict[(index,totalSum)] = findSubset(index-1,totalSum,allElems)


    return subsetDict[(index,totalSum)]
findSubset(5,8,allElems)
```

Out[2]:

```
True
```

Question 3

Divide and conquer approach

Its a O(n logn) algorithm

```

def getMin(Arr,start,end):
    MinVal = Arr[start]
    MinIndex = start
    for i in range(start, end+1):
        if Arr[i] < MinVal:
            MinVal = Arr[i]
            MinIndex = i
    return MinVal, MinIndex

def getMax(Arr,start,end):
```

```
    MaxVal = Arr[start]
    MaxIndex = start
    for i in range(start, end+1):
        if Arr[i]> MaxVal:
            MaxVal = Arr[i]
            MaxIndex = i
    return MaxVal, MaxIndex



def getMaxDiff(start,end,arr):
    final = 0
    if start < end :

        ## return difference indices as max and min are both the same zero if there is
only one element
        mid = start +  int((end-start)/2)

        leftI,leftJ  = getMaxDiff(start,mid,arr) ##indices of i and j in the left
        rightI,rightJ = getMaxDiff(mid+1,end,arr) ## indices of i on j on the right

        leftDiff = arr[leftJ] - arr[leftI] ## leftMaxDifference
        rightDiff = arr[rightJ] - arr[rightI] ## rightMaxDifference

        minLeft,minIndexLeft = getMin(arr,start,mid) # index and min value
        maxRight,maxIndexRight = getMax(arr,mid+1, start) # index and max value

        centerDiff = maxRight - minLeft ## difference in between the left child and
right child

        final =  max(centerDiff, max(leftDiff,rightDiff))

        ## if final came from center diff
        if(centerDiff == final):
            return minIndexLeft,maxIndexRight

        ## if final came from right Difference
        elif (rightDiff == final):
            return rightI,rightJ

        ## if final came from left  deifference
        else:
            return leftI,leftJ

    return (0,0)
```

In [59]:

```
def getMin(Arr,start,end):
    MinVal = Arr[start]
    MinIndex = start
    for i in range(start, end+1):
        if Arr[i] < MinVal:
            MinVal = Arr[i]
            MinIndex = i
    return MinVal, MinIndex

def getMax(Arr,start,end):

    MaxVal = Arr[start]
    MaxIndex = start
    for i in range(start, end+1):
        if Arr[i]> MaxVal:
            MaxVal = Arr[i]
            MaxIndex = i
    return MaxVal, MaxIndex



def getMaxDiff(start,end,arr):
    final = 0
    if start < end :
        ## return difference indices as max and min are both the same zero if there is only
        mid = start +  int((end-start)/2)

        leftI,leftJ  = getMaxDiff(start,mid,arr)
        rightI,rightJ = getMaxDiff(mid+1,end,arr)

        leftDiff = arr[leftJ] - arr[leftI]
        rightDiff = arr[rightJ] - arr[rightI]

        minLeft,minIndexLeft = getMin(arr,start,mid)
        maxRight,maxIndexRight = getMax(arr,mid+1, start)

        centerDiff = maxRight - minLeft

        final =  max(centerDiff, max(leftDiff,rightDiff))

        if(centerDiff == final):
            return minIndexLeft,maxIndexRight

        elif (rightDiff == final):
            return rightI,rightJ

        else:
            return leftI,leftJ

    return (0,0)


getMaxDiff(0,2,[1,2,43,4,5,6])
```

Out[59]:

```
(0, 2)
```

Dynamic Approach

The complexity is O(N)

$$\max_{0 \le i \le n} \{A[i] - min(A[1:i-1])\}$$

```
In the above equation minimum from 1 to i-1 is prepared before hand
example if the list that is given is [2,4,5,1,0]
Then the min list will be [2,2,2,1,0] i.e the minimum element till that index

That way we do not run minimum over entire array at each i
The equation in the above cell reduces to the below
```

$$\max_{0 \le i \le n} \{A[i] - minList[i-1])\}$$

```
###pseudo code
import random
IsMaxMinInitialised = False
minList  = []
## denotes (indexOfMin, minimum Value) initialising all to 0,Infinity
minList = [[0,float("Inf")] for _ in numList]

minList[0][1] = numList[0] ## storeing the first number as minimum for the first min

numList = [1,23,5,6,7,8] ## say the list of numbers given

def maxDiffTillIndex(tillIndex,numList):

    ###prepare min list till tillIndex

    if numList[tillIndex] < minList [tillIndex-1][1]: ## next index value is lesser
        minList [tillIndex][1] = numList[tillIndex] ##update minValue
        minList [tillIndex][0] = tillIndex ##update index from


    else : ## if next index value is greater than the minimul till now
        minList [tillIndex][1] = minList [tillIndex-1][1]
        minList [tillIndex][0] = minList [tillIndex-1][0]


    maxDiff = numList[tillIndex]-minList[tillIndex-1][1]

    return maxDiff, minList[tillIndex-1] ## difference and the (indexOfMin, minimum
Value)



def getMaxDiff(numList):


    maxValue = -float("Inf")

    for i in range(1,len(numList)):
```

```
        ## notice that the minimum list is prepared till 'i-1'already
        (maxDiff,(frmIndex,minValue))  = maxDiffTillIndex(i,numList)

        ## find the max difference till now
        if maxDiff > maxValue:

            maxValue = maxDiff
            indexAt = frmIndex ## tracking the minimum index
            tillIndex = i
    return maxValue,indexAt,tillIndex ## max value , i, j
```

In [8]:

```
import random
IsMaxMinInitialised = False
minList  = []
def maxDiffTillIndex(tillIndex,numList):

    global IsMaxMinInitialised
    global minList
    if not IsMaxMinInitialised:
        ##index from where min came, minValue

        minList = [[0,float("Inf")] for _ in numList]
        minList[0][1] = numList[0]

        IsMaxMinInitialised = True

    ###prepare min list
    if numList[tillIndex] < minList [tillIndex-1][1]: ## next index value is lesser
        minList [tillIndex][1] = numList[tillIndex] ##update minValue
        minList [tillIndex][0] = tillIndex ##update index from


    else : ## if next index value is greater than the minimul till now
        minList [tillIndex][1] = minList [tillIndex-1][1]
        minList [tillIndex][0] = minList [tillIndex-1][0]


    maxDiff = numList[tillIndex]-minList[tillIndex-1][1]

    return maxDiff, minList[tillIndex-1]



def getMaxDiff(numList):


    maxValue = -float("Inf")
    indexAt = 0


    for i in range(1,len(numList)):

        (maxDiff,(frmIndex,minValue))  = maxDiffTillIndex(i,numList)


        ## find the max difference till now
        if maxDiff > maxValue:

            maxValue = maxDiff
            indexAt = frmIndex
            tillIndex = i
    return maxValue,indexAt,tillIndex ## max value , i, j


numList = [random.randint(0,100) for i in range(50000)]
solution = getMaxDiff(numList)
print("Maximum Difference is :" ,solution[0], "and i is :",solution[1], " and j is :", solu
```

```
Maximum Difference is : 100 and i is : 41  and j is : 101
```

Question 4

1. Let all the people be invited first 1, . . . n.
2. Build a Graph on all the invited people.
3. Let the Subset I be the invited people (initially all the people),
4. Pick each of member i from set I and check if he satisifies the condition that he knows 5 or more people and doesnt know 5 or more people from I
5. If i doesnt satisfy the above condition remove i from set I i.e $I = I - i$
6. Upadte the Graph on set I since the vertex i and its edges are to be removed
7. While there exists $i \in I$ such that condition 4 is violated, repeat 5 and 6
8. Return I

```
### Pseudo Code:
### building a graph has a function to delete a vertex , getChildren returns all the
nodes children
class graph():

    def __init__(self,dictOfNodesAndConnections = None):

        if dictOfNodesAndConnections is None:


            self.nodesAndEdges = {}

        else:
            self.nodesAndEdges = dict(dictOfNodesAndConnections)
        pass

    def getChildren(self,atNode):

        if atNode in self.nodesAndEdges:
            return self.nodesAndEdges[atNode]
        else:
            return []

    def delNode(self,node):
        ## removing a node
        if node in self.nodesAndEdges:
            del self.nodesAndEdges[node]

        ## refresh connections
            ## removinga all edges
            for vertex in self.nodesAndEdges:
                if node in self.nodesAndEdges[vertex]:
                    self.nodesAndEdges[vertex].remove(node)
        else :
            pass



myOrigGraph = graph(dictOfConnections) ## orginal graph of all teh memebrs are
considered
inviteGraph = graph(dictOfConnections) ### invitation set which is constantly updated
```

```
def removeInvites(inviteGraph):

    while True: ## while there exists a i violating the required condition

        listOfNodes = list(inviteGraph.nodesAndEdges.keys()) ## copying the memebers
present in the set I currently
        countOfRetainedInvites = 0
        for node in listOfNodes:

            ## known <5 or unknown < 5 invalid members
            if (len(inviteGraph.getChildren(node)) < 5 or
                len(inviteGraph.nodesAndEdges) - len(inviteGraph.getChildren(node)) < 5)
:
                inviteGraph.delNode(node)


            else: ## valid members
                countOfRetainedInvites +=1

        if countOfRetainedInvites == len(listOfNodes): ## if all of the memebrs in set I
are valid
            break;
```

In [20]:

```
## coding

## function for generating random connections
def genRandomConnections(totalMembers):
##generate random connections
    listOfConnections = [(random.randint(0,totalMembers-1),
                          random.randint(0,totalMembers-1)) for _ in range(totalMembers*5)]

    ## get unique conncetions
    listOfConnections = list(set(listOfConnections))

    ## remove self connections
    listOfConnections = [conn for conn in listOfConnections if conn[0] != conn[1]]
    return listOfConnections

listOfConnections = genRandomConnections(20)


## converting connections to dictionary of node and its connections as list
## key is node and value is a list of nodes its conencted to.
def convertToDictOfConnections(totalMembers,listOfConnections):

    dictOfConnections = {key:[] for key in range(totalMembers)}

    for frmNode,toNode in listOfConnections:
        if toNode not in dictOfConnections[frmNode]:
            dictOfConnections[frmNode].append(toNode)
        if frmNode not in dictOfConnections[toNode]:
            dictOfConnections[toNode].append(frmNode)

    return dictOfConnections


dictOfConnections = convertToDictOfConnections(20,listOfConnections)





class graph():

    def __init__(self,dictOfNodesAndConnections = None):

        if dictOfNodesAndConnections is None:


            self.nodesAndEdges = {}

        else:
            self.nodesAndEdges = dict(dictOfNodesAndConnections)
        pass

    def getChildren(self,atNode):

        if atNode in self.nodesAndEdges:
            return self.nodesAndEdges[atNode]
        else:
            return []
```

```
    def delNode(self,node):
        ## removing a node
        if node in self.nodesAndEdges:
            del self.nodesAndEdges[node]

        ## refresh connections
            ## removinga all edges
            for vertex in self.nodesAndEdges:
                if node in self.nodesAndEdges[vertex]:
                    self.nodesAndEdges[vertex].remove(node)
        else :
            pass



myOrigGraph = graph(dictOfConnections)
inviteGraph = graph(dictOfConnections)


def removeInvites(inviteGraph):

    while True: ## while there exists a i violating the required condition

        listOfNodes = list(inviteGraph.nodesAndEdges.keys())
        countOfRetainedInvites = 0
        for node in listOfNodes:

            ## known <5 or unknown < 5
            if (len(inviteGraph.getChildren(node)) < 5 or
            len(inviteGraph.nodesAndEdges) - len(inviteGraph.getChildren(node)) < 5) :
                inviteGraph.delNode(node)


            else:
                countOfRetainedInvites +=1

        if countOfRetainedInvites == len(listOfNodes): ## if all of the memebrs in I are vo
            break;
    pass


removeInvites(inviteGraph)
inviteGraph.nodesAndEdges
```

Out[20]:

```
{0: [7, 9, 6, 16, 13, 15, 1, 3, 12, 19, 2],
 1: [15, 6, 2, 5, 10, 0, 19, 3, 11, 8],
 2: [3, 14, 1, 19, 13, 16, 0],
 3: [19, 2, 9, 11, 14, 1, 0, 10, 16],
 5: [1, 10, 13, 16, 12],
 6: [1, 7, 0, 13, 10, 19],
 7: [12, 0, 6, 15, 19, 11, 16],
 8: [10, 13, 11, 16, 1],
 9: [0, 3, 10, 13, 11, 12],
 10: [8, 13, 14, 1, 9, 19, 15, 5, 3, 6, 12],
 11: [3, 9, 13, 7, 8, 1],
 12: [7, 15, 0, 9, 5, 16, 10],
 13: [19, 10, 6, 0, 15, 9, 8, 5, 2, 11, 14],
 14: [15, 2, 10, 3, 16, 13],
 15: [1, 14, 13, 7, 0, 10, 12],
```

```
 16: [0, 3, 5, 14, 8, 7, 12, 2],
 19: [3, 13, 7, 10, 1, 2, 0, 6]}
```

Question 5

This is a Greedy Approach

1. Convert the given times in 24 Hour format.
2. Sort the list of Jobs based on their finishing time in ascending order
3. For each job m that spans over midnight.
   a. Remove all of the jobs that are not comaptible with m in the set $listOfJobs$ say the set is $M_{noncomaptible}$
   b. now solve greedely the sub problem $listOfJobs - (m + M_{noncomaptible})$

Sub-problem solving:

1. Solving the sub problem involves first selecting the first job i in the sorted list
2. Remove all the jobs from sorted list that are incomatible with i
3. Repeat 1,2 till progressively on all the jobs in the sorted till all the jobs in the list are compatible.

```
## imagines the listOfJobs list has jobs [(start,end)] in 24 HR format
## main function to solve subproblem
def solveForMax(sortedList):


    while True:
        copySortedList = list(sortedList)
        if len(copySortedList) == 0:
            break;
        for job in copySortedList:
            ##pick a job remove its intersections if any
            removeIntersectionsOfJobChosen(job,sortedList)

        ## if all of the jobs are comaptinble in the list
        if len(sortedList) == len(copySortedList):
            return sortedList


## Main function choose each midnight job and solve the remainng subproblem withouth the
job
def checkMidnightJobs(intersectionDict,listOfJobTimes):

    maxSolutionLen = -float("Inf")
    for job in intersectionDict:
        newListOfAllJobs = list(listOfJobTimes)

        if job[0] > job[1]:## choose a job that spans over midnight
            removeIntersectionsOfJobChosen(job,newListOfAllJobs)

            ## sort the list according to the finishing time
            sortedJobs = sorted(newListOfAllJobs,key = lambda x: x[1])

            ## solve the sub problem
            solution   = solveForMax(sortedJobs)
            if len(solution) > maxSolutionLen:
                maxSolutionLen = len(solution)
                maxSolution = solution
```

```
    return maxSolution
```

```
##pesudo code
## auxilarry functions to support above

## not the main funtion only to check incomaptible jobs
def isIntersecting(timeInt1, timeInt2):

    if timeInt1[0] > timeInt1[1] and timeInt2[0] > timeInt2[1] :
    ## both crossing over night hence intersecting
        return True

    ##both proper spanning within


    ##one of it is over midnight

        ##if time1 is over midnight
        #other time2  is normal
    if timeInt1[0] > timeInt1[1] :

        if timeInt1[0] >=  timeInt2[1] and timeInt1[1] <=  timeInt2[0] :
            return False
        else:
            return True

    ##if time2 is over midnight
    #other time1 is normal
    if timeInt2[0] > timeInt2[1] :

        if timeInt2[0] >=  timeInt1[1] and timeInt2[1] <=  timeInt1[0] :
            return False
        else:
            return True


    ## if both are normal
    if timeInt1[0] < timeInt1[1] and timeInt2[0] < timeInt2[1]:
        ##start1 between time2
        if timeInt1[0] >= timeInt2[0] and timeInt1[0] <= timeInt2[1]:
            return True
        ##start2 between time1
        if timeInt2[0] >= timeInt1[0] and timeInt2[0] <= timeInt1[1]:
            return True

        else:
            return False

    else:

        return False

jobNumbersPicked = []

## building a graph like structure with all of key: job and value : incompatible jobs
def getIntersectionDict(listOfJobTimes):
    intersectionDict = {key: [] for key in listOfJobTimes}
    for i,jobI in enumerate( listOfJobTimes):
```

```
        for j,jobJ in enumerate( listOfJobTimes):

            if i != j and isIntersecting(jobI,jobJ):
                intersectionDict[jobI].append(jobJ)


    return intersectionDict

listOfJobTimes = [(18,6),(21,4),(3,14),(13,19)]


## build intersection or jobs that are not compatible i.e key: Job and Value : list of
non comaptible jobs
intersectionDict = getIntersectionDict(listOfJobTimes)

## function to remove all the jobs incomatible with the job chosen now
def removeIntersectionsOfJobChosen(jobChosen,listOfJobs):
    global intersectionDict
    if jobChosen in intersectionDict:
        for intJobs in intersectionDict[jobChosen]:
            if intJobs in listOfJobs: ##try except may be less expsensive
                listOfJobs.remove(intJobs)

    else:
        return False

    pass
```

In [21]:

```
def isIntersecting(timeInt1, timeInt2):

    if timeInt1[0] > timeInt1[1] and timeInt2[0] > timeInt2[1] :
    ## both crossing over night hence intersecting
        return True

    ##both proper spanning within


    ##one of it is over midnight
    ##if time1 is over midnight
    if timeInt1[0] > timeInt1[1] : #other time2 one is normal

        if timeInt1[0] >=  timeInt2[1] and timeInt1[1] <=  timeInt2[0] :
            return False
        else:
            return True

    ##if time2 is over midnight
    if timeInt2[0] > timeInt2[1] : #other one is normal

        if timeInt2[0] >=  timeInt1[1] and timeInt2[1] <=  timeInt1[0] :
            return False
        else:
            return True


    ## if both are normal
    if timeInt1[0] < timeInt1[1] and timeInt2[0] < timeInt2[1]:
        ##start1 between time2
        if timeInt1[0] >= timeInt2[0] and timeInt1[0] <= timeInt2[1]:
            return True
        ##start2 between time1
        if timeInt2[0] >= timeInt1[0] and timeInt2[0] <= timeInt1[1]:
            return True

        else:
            return False

    else:

        return False

jobNumbersPicked = []

def getIntersectionDict(listOfJobTimes):
    intersectionDict = {key: [] for key in listOfJobTimes}
    for i,jobI in enumerate( listOfJobTimes):

        for j,jobJ in enumerate( listOfJobTimes):

            if i != j and isIntersecting(jobI,jobJ):
                intersectionDict[jobI].append(jobJ)


    return intersectionDict

listOfJobTimes = [(18,6),(21,4),(3,14),(13,19)]
```

```
## build intersection or jobs that are not compatible i.e key: Job and Value : list of non
intersectionDict = getIntersectionDict(listOfJobTimes)

def removeIntersectionsOfJobChosen(jobChosen,listOfJobs):
    global intersectionDict
    if jobChosen in intersectionDict:
        for intJobs in intersectionDict[jobChosen]:
            if intJobs in listOfJobs: ##try except may be less expsensive
                listOfJobs.remove(intJobs)

    else:
        return False

    pass


## subproblem
def solveForMax(sortedList):


    while True:
        copySortedList = list(sortedList)
        if len(copySortedList) == 0:
            break;
        for job in copySortedList:
            ##pick a job remove its intersections if any
            removeIntersectionsOfJobChosen(job,sortedList)

        ## if all of the jobs are comaptinble in the list
        if len(sortedList) == len(copySortedList):
            return sortedList


def checkMidnightJobs(intersectionDict,listOfJobTimes):

    maxSolutionLen = -float("Inf")
    for job in intersectionDict:
        newListOfAllJobs = list(listOfJobTimes)
        if job[0] > job[1]:## choose a job that spans over midnight
            removeIntersectionsOfJobChosen(job,newListOfAllJobs)

            ## sort the list according to the finishing time
            sortedJobs = sorted(newListOfAllJobs,key = lambda x: x[1])
            solution    = solveForMax(sortedJobs)
            if len(solution) > maxSolutionLen:
                maxSolutionLen = len(solution)
                maxSolution = solution

    return maxSolution

checkMidnightJobs(intersectionDict,listOfJobTimes)
```

Out[21]:

```
[(21, 4), (13, 19)]
```