**VIETNAM GENERAL CONFEDERATION OF LABOUR**

**TON DUC THANG UNIVERSITY**

**INFORMATION TECHNOLOGY FALCULTY**



**FINAL REPORT**

**ALGORITHM AND DESIGN STRAGTEGY**

**FINAL PROJECT REPORT**

*Instructor*: **Nguyen Chi Thien**

*Author*: **Nguyen Dinh Minh Khoi – 519K0013**

Class**: 19K50201**

**HO CHI MINH CITY 2021**

APPRECIATION

Thank you Mr. Thien in class was teaching us all of useful knowledge and also encourage us doing exercise by giving plus points. Therefore, I was enthusiastic and did all of exercise. With the knowledge that you gave me and the practice I do daily. I was able to finish this subject easily and also I can do this final project report easier than ever. I just want to give all my respect and appreciation to Mr Thien for everything that I have been receiving all the time.

THE REPORT WAS COMPLETED AT TON DUC THANG UNIVERSITY

We assure that this project was completed by ourselves with the instruction of Mr.Nguyen Chi Thien. Research contents and results in this topic are honest and had not ever been published in any form.

In addition, the project also uses a number of comments, evaluations as well as data of other authors, other organizations and organizations with citations and origin notes.

If any fraud is detected, we will take full responsibility for the content of our project. Ton Duc Thang University is not related to any copyright or copyright infringement caused by us during the implementation process (if any).

Ho Chi Minh city, date…month…year…

Author

(sign with full name)

Nguyen Dinh Minh Khoi

LECTURER AND EXAMINER SCORING SECTION

**Comment and scoring for instructor**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh city, date…month…year

(sign with full name)

**Comment and scoring for examiner**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh city, date…month…year

(sign with full name)

OVERVIEW

This report will give reader about 10 design strategy and also 2 algorithms as examples. This report also have the analyze of each algorithm, the test case and also running time function to demo the algorithm.

TABLE OF CONTENT

[APPRECIATION 2](#_Toc91537303)

[THE REPORT WAS COMPLETED AT TON DUC THANG UNIVERSITY 3](#_Toc91537304)

[LECTURER AND EXAMINER SCORING SECTION 4](#_Toc91537305)

[OVERVIEW 5](#_Toc91537306)

[TABLE OF CONTENT 6](#_Toc91537307)

[I. BRUTE-FORCE-STRATEGY 10](#_Toc91537308)

[1. Explanation: 10](#_Toc91537309)

[2. Brute-force sorting: 10](#_Toc91537310)

[a. Algorithm idea: 10](#_Toc91537311)

[b. Time complexity: 10](#_Toc91537312)

[c. Code implementation and Demo: 10](#_Toc91537313)

[3. Brute-force searching: 14](#_Toc91537314)

[a. Algorithm idea: 14](#_Toc91537315)

[b. Time Complexity: 14](#_Toc91537316)

[c. Code implementation and demo: 14](#_Toc91537317)

[II. DIVIDE AND CONQUER STRATEGY: 17](#_Toc91537318)

[1. Explanation: 17](#_Toc91537319)

[2. Mergesort divide and conquer: 18](#_Toc91537320)

[a. Algorithm idea: 18](#_Toc91537321)

[b. Time complexity: 18](#_Toc91537322)

[c. Code implementation and demo: 19](#_Toc91537323)

[3. Find Tree height divide and conquer: 22](#_Toc91537324)

[a. Algorithm idea: 22](#_Toc91537325)

[b. Time complexity: 22](#_Toc91537326)

[c. Code implementation and demo: 22](#_Toc91537327)

[III. DECREASE AND CONQUER STRATEGY 25](#_Toc91537328)

[1. Explanation: 26](#_Toc91537329)

[2. find greatest common divisor decrease and conquer: 26](#_Toc91537330)

[a. Algorithm idea: 26](#_Toc91537331)

[b. Time complexity: 26](#_Toc91537332)

[c. Code implementation and demo: 26](#_Toc91537333)

[3. Insertion sort decrease and conquer: 30](#_Toc91537334)

[a. Algorithm idea: 30](#_Toc91537335)

[b. Time complexity: 30](#_Toc91537336)

[c. Code implementation and demo: 30](#_Toc91537337)

[IV. GREEDY APPROACH STRATEGY 33](#_Toc91537338)

[1. Explanation: 33](#_Toc91537339)

[2. Job Sequence with deadlines greedy approach: 33](#_Toc91537340)

[a. Algorithm idea: 33](#_Toc91537341)

[b. Time complexity: 33](#_Toc91537342)

[c. Code implementation and demo: 33](#_Toc91537343)

[3. Kruskal’s Algorithm greedy approach: 37](#_Toc91537344)

[a. Algorithm idea: 37](#_Toc91537345)

[b. Time complexity: 37](#_Toc91537346)

[c. Code implementation and demo: 38](#_Toc91537347)

[V. DYNAMIC PROGRAMMING STRATEGY 42](#_Toc91537348)

[1. Explanation: 42](#_Toc91537349)

[2. Change-Making problem dynamic programming: 43](#_Toc91537350)

[a. Algorithm idea: 43](#_Toc91537351)

[b. Time complexity: 43](#_Toc91537352)

[c. Code implementaion and demo: 43](#_Toc91537353)

[3.Longest common sequences dynamic programming: 46](#_Toc91537354)

[a. Algorithm idea: 46](#_Toc91537355)

[b. Time complexity: 46](#_Toc91537356)

[c. Code implementation and demo: 47](#_Toc91537357)

[VI. TRANSFORM AND CONQUER STRATEGY 50](#_Toc91537358)

[1. Explanation: 50](#_Toc91537359)

[2. Check uniqueness transform and conquer: 50](#_Toc91537360)

[a. Algorithm idea: 50](#_Toc91537361)

[b. Time complexity: 51](#_Toc91537362)

[c. Code implementation and demo: 51](#_Toc91537363)

[3. Heapsort transform and conquer: 55](#_Toc91537364)

[a. Algorithm idea: 55](#_Toc91537365)

[b. Time complexity: 55](#_Toc91537366)

[c. Code implementation and demo: 55](#_Toc91537367)

[VII. ITERATIVE IMPROVEMENT STRATEGY 58](#_Toc91537368)

[1. Explanation: 58](#_Toc91537369)

[2. Ford-fulkerson for maxflow Iterative improvement: 58](#_Toc91537370)

[a. Algorithm idea: 58](#_Toc91537371)

[b. Time complexity: 58](#_Toc91537372)

[c. Code implementation and demo: 59](#_Toc91537373)

[3. N-queen using Iterative improvement method: 62](#_Toc91537374)

[a. Algorithm idea: 62](#_Toc91537375)

[b. Time complexity: 62](#_Toc91537376)

[c. Code implementation and demo: 62](#_Toc91537377)

[VIII. BACKTRACKING STRATEGY 66](#_Toc91537378)

[1. Explanation: 67](#_Toc91537379)

[2. N-queen problem using backtracking method: 67](#_Toc91537380)

[a. Algorithm idea: 67](#_Toc91537381)

[b. Time complexity: 67](#_Toc91537382)

[c. Code implementation and demo: 67](#_Toc91537383)

[3. Subset of sum using backtracking method: 71](#_Toc91537384)

[a. Algorithm idea: 71](#_Toc91537385)

[b. Time complexity: 71](#_Toc91537386)

[c. Code implementation and demo: 72](#_Toc91537387)

[IX. BRANCH AND BOUND STRATEGY 75](#_Toc91537388)

[1. Explanation: 75](#_Toc91537389)

[2. N-queen using brand and bound method: 75](#_Toc91537390)

[a. Algorithm idea: 75](#_Toc91537391)

[b. Time complexity: 76](#_Toc91537392)

[c. Code implementation and demo: 77](#_Toc91537393)

[3. Traveling sale man problem using brand and bound method: 81](#_Toc91537394)

[a. Algorithm idea: 81](#_Toc91537395)

[b. Time complexity: 82](#_Toc91537396)

[c. Code implementation and demo: 82](#_Toc91537397)

[X. APPROXIMATION ALGORITHM STRATEGY 87](#_Toc91537398)

[1. Explanation: 87](#_Toc91537399)

[2. Subset of sum approximate method: 88](#_Toc91537400)

[a. Algorithm idea: 88](#_Toc91537401)

[b. Time complexity: 88](#_Toc91537402)

[c. Code implementation and demo: 88](#_Toc91537403)

[3. Travelling sale man approximate method: 91](#_Toc91537404)

[a. Algorithm idea: 91](#_Toc91537405)

[b. Time complexity: 91](#_Toc91537406)

[c. Code implementation and demo: 91](#_Toc91537407)

I. BRUTE-FORCE-STRATEGY

# 1. Explanation:

It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.

Brute force algorithm is a technique that guarantees solutions for problems of any domain helps in solving the simpler problems and also provides a solution that can serve as a benchmark for evaluating other design techniques, but takes a lot of run time and inefficient.

# 2. Brute-force sorting:

One of sorting method that is brute-force strategy is selection sort.

## a. Algorithm idea:

Selection sort: Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i (0 <= i <= n-2), find the smallest element in A[i..n-1] and swap it with A[i].

## b. Time complexity:

we are using two for loop that running from 0. The first loop run from 0 to n-1: and each time the second loop run from (i+1) to n with i is the ith time first loop run

In the worst case: this algorithm worst case is average case, all the time it will run through all loop.

Time Complexity = n-1 + n-2 + n-3 +…+1 = n(1+n)/2 ϵ Θ(n2)

## c. Code implementation and Demo:

Code implementation:

def BFsorting(lst):

    '''

    this brute force sorting function will sort a list in ascending order

    (this algorithm called selection sort)

    '''

    for i in range(len(lst)-1):

        min = lst[i]

        for j in range(i+1,len(lst)):

            if min > lst[j]:

                min = lst[j]

                lst[j],lst[i] = lst[i],lst[j]

    return lst

#test case

newList = [6,1,2,9,4,7,3]

output = [1,2,3,4,6,7,9]

print(BFsorting(newList))

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

def measure(n):

    start = time.time()

    BFsorting(n)

    stop = time.time()

    return stop - start

n2 = list(range(0,10000,500))

print(n2)

n3 = [i\*\*2 for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n^2'])

plt.show()

iput = []

for i in n2:

    temp = [0]\*i

    iput.append(temp)

running\_time = list(map(measure,iput))

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

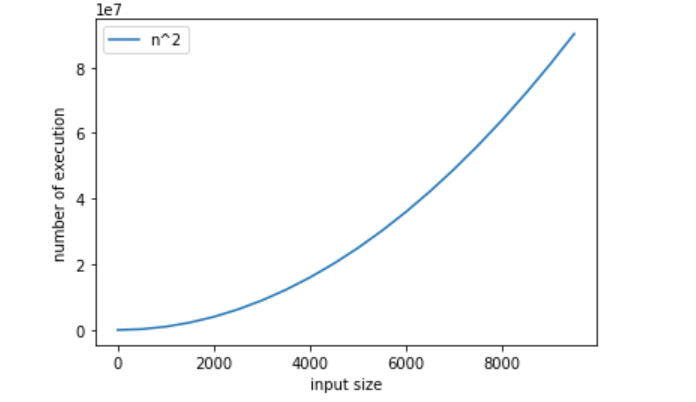
plt2.ylabel('time running in second')

plt2.legend(['buublesort'])

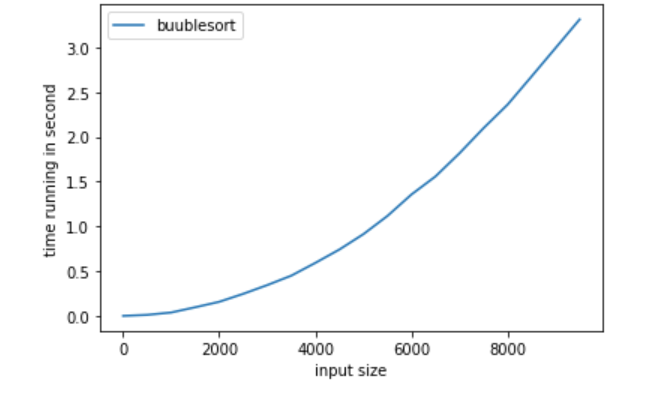
plt2.show()

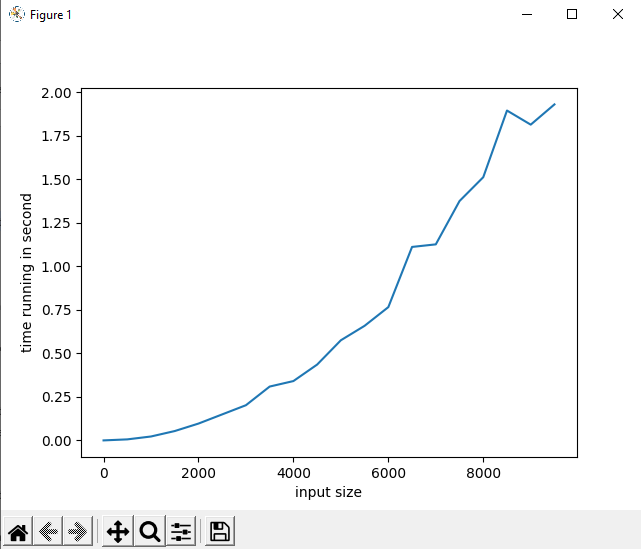
Running time graph Demo

Therotical running time:



Function running time:





# 3. Brute-force searching:

## a. Algorithm idea:

To find a key in a list we can use this brute-force search by Using a loop to compare element with key, if it is element equal key return the index. If not we go next element until end loop return None.

## b. Time Complexity:

This algorithm has 1 for loop run from 0 to n-1 time: in the worst case the loop will run through all element in this loop:

Time complexity: T(n) = n-1 ϵ Θ(n)

## c. Code implementation and demo:

def BFsearch(lst,key):

    '''

    This brute force search function is to search

    a key in a list and return its index

    '''

    for i in range(len(lst)):

        if lst[i] == key:

            return i

    return None

#test case

lst = [1,5,2,8,3,7,9]

key = 8

output = 3

print(BFsearch(lst,key))

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import copy

def measureBF(n):

    start = time.time()

    A = BFsearch(n,-1)

    stop = time.time()

    return stop - start

n2 = list(range(0,10000,200))

plt2 = plt

plt.plot(n2,n2)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n'])

plt.show()

iput = []

for i in n2:

    temp = [0]

    for j in range(i):

        temp.append(j)

    iput.append(temp)

running\_time = [measureBF(x) for x in iput]

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

plt2.ylabel('time running in second')

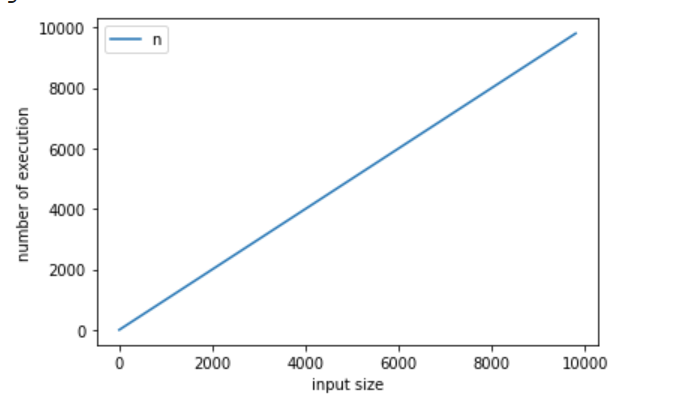
plt2.legend(['brute-force search'])

plt2.show()

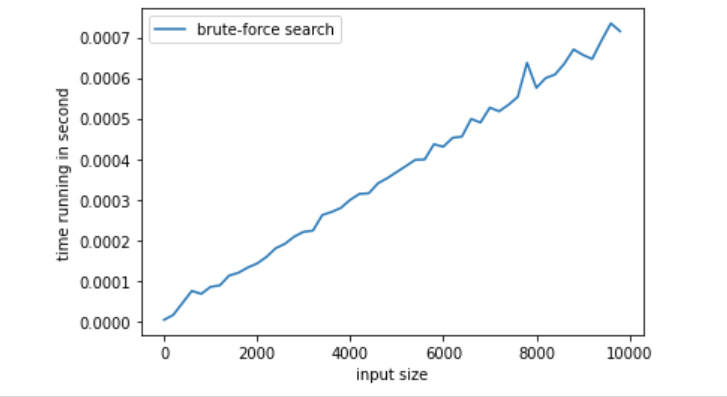
This running time function will create input list (with element is list with size increasing)

Running time graph:

Therotical grahp:



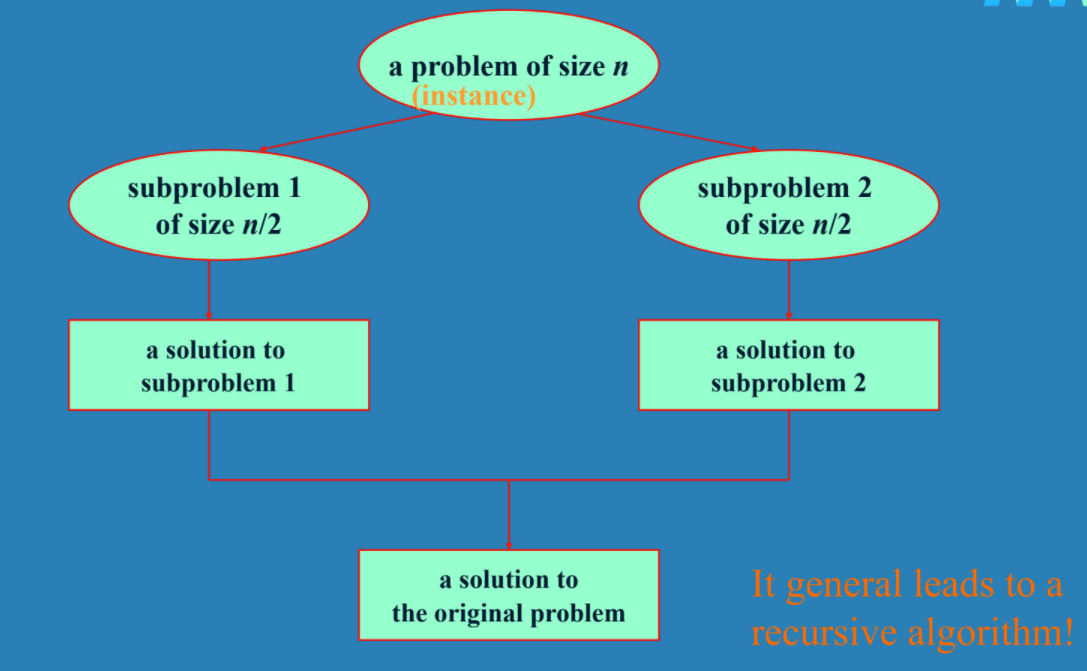
Function running graph:



II. DIVIDE AND CONQUER STRATEGY:

# 1. Explanation:

This strategy will divide the problem in to smaller instances then solve these small instances to get solutions and then those solutions them together to get the bigger instance.



# 2. Mergesort divide and conquer:

## a. Algorithm idea:

Split array A[0…n-1] in to 2 equals array B and C, then sort B and C resursively, then merge array B and array C to get sorted array A as follow:

+ repeat until no element remain in one of array:

- compare 1st element two array

- merge the smaller one to A while increasing the index indicate the unprocessed portion of that array

+ Once all element are processed copy the rest unprocessed element form other array to A.

## b. Time complexity:

The comparison we take is equal to number of elements in first array + number of elements in second array (Θ(sub1) + Θ(sub2)) because we compare each element in each array. But sub1 + sub2 = n (we divide n to 2 equal array sub1 and sub2)

Therefore comparison time complexity: Θ(n)

We divide problem to 2 sub array:

Time complexity: T(n) = 2T(n/2) + Θ(n)

According to master theorem:

T(n) = aT(n/b) + f(n) where f(n) ϵ Θ(nd) with d>=0

If a = b => T(n) ϵ Θ(ndlog n)

We have: a = 2, b = 2, d = 1

=> T(n) = 2T(n/2) + Θ(n1) ϵ Θ(nlogn)

## c. Code implementation and demo:

Code implementation:

def mergesort(A, i, j):

    '''

    mergesort is sorting list with given starting index i and ending index j

    in merge we divide by 2 smaller and sort again

    '''

    if i == j: return None

    k = (i + j)//2

    mergesort(A,i,k)

    mergesort(A,k+1,j)

    merge(A, i, k, j)

def merge(A,i,k,j):

    '''This function is to merge 2 list after sorting

    '''

    B = [0 for \_ in range(len(A))]

    p1, p2, p3 = i, k+1, i

    while p1 <= k and p2 <= j:

        if A[p1] < A[p2]:

            B[p3] = A[p1]

            p1 += 1

        else:

            B[p3] = A[p2]

            p2 += 1

        p3 += 1

    while p1 <= k:

        B[p3] = A[p1]

        p3 += 1

        p1 += 1

    while p2 <= j:

        B[p3] = A[p2]

        p3 += 1

        p2 += 1

    for r in range(i, j+1):

        A[r] = B[r]

#test case

newList = [2,5,1,7,8,3,8,8,6]

output = [1,2,3,5,6,7,8,8,8]

mergesort(newList,0,len(newList)-1)

print(newList)

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import copy

import math

def measure(n):

    start = time.time()

    A = mergesort(n,0,len(n)-1)

    stop = time.time()

    return stop - start

n2 = list(range(1,10000,200))

plt2 = plt

plt.plot(n2,[i\*math.log(i) for i in n2])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['nlogn'])

plt.show()

iput = []

for i in n2:

    temp = [0]

    for j in range(i):

        temp.append(j)

    iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

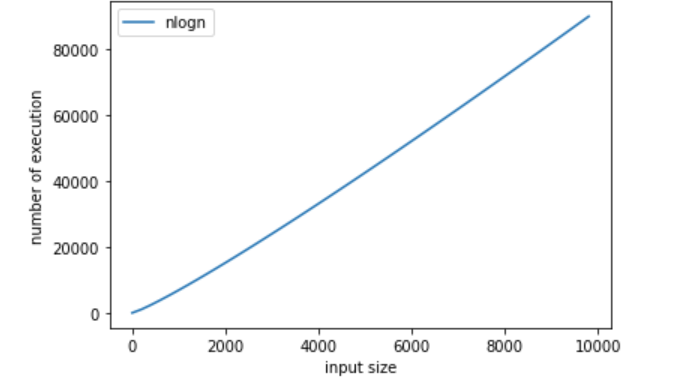
plt2.ylabel('time running in second')

plt2.legend(['mergesort divide and conquer'])

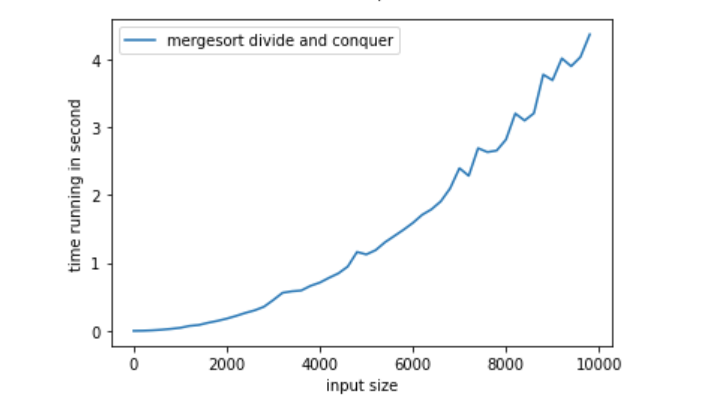
plt2.show()

This time function given input N with list element from 0 to 100

Running time graph therotical for nlogn:



Running time graph function for mergesort:



# 3. Find Tree height divide and conquer:

## a. Algorithm idea:

A tree has maximum n node with n is number of height. We divide tree into 2 part with n/2 child node in left and n/2 child right and keep continue doing so to get the height. If the node only has left => return left + 1 (because height in right stop) and the same if the node only has right. If the node has no children we return 1.

## b. Time complexity:

We return when node has no children => we go every single node at once no matter the tree balance or not until all leaf node is checked => Time complexity is T(n) ϵ Θ(n) with n is number of node.

## c. Code implementation and demo:

Code implementation:

First I make a class Node and building tree function to build a tree

class Node:

    def \_\_init\_\_(self,value,left,right) -> None:

        '''

        value is the value of the node

        left is the left children node

        right is the right children node

        '''

        self.value = value

        self.left = left

        self.right = right

def insertToTree(node,child):

    '''

    node is the root of tree

    children is the node we want to insert

    '''

    if(child.value<node.value):

        if(node.left == None):

            node.left = child

        else:

            insertToTree(node.left,child)

    else:

        if(node.right == None):

            node.right = child

        else:

            insertToTree(node.right,child)

def buildTree(newNode,lst):

    '''

    build a tree from the begining

    lst is input list of values for node

    ex: [3,5,1,6,7]

    '''

    for i in range(1,len(lst)):

        insertToTree(newNode,Node(lst[i],None,None))

Here is the example tree

listofvalue = [10,6,7,5,9,12,20,15,3]

'''

               10

            6       12

          5   7         20

        3       9      15

height = 4

if add 2 and 1 => height is 6

'''

newNode = Node(listofvalue[0],None,None)

buildTree(newNode,listofvalue)

now is find height function

def findtreeHeight(node):

    if(node.left == None and node.right == None):

        return 1

    if(node.left == None):

        return 1 + findtreeHeight(node.right)

    elif(node.right == None):

        return 1 + findtreeHeight(node.left)

    else: #if the node has 2 children then find which way is longer

        leftheight = 1 + findtreeHeight(node.left)

        rightheight = 1 + findtreeHeight(node.right)

        if(leftheight<rightheight):

            return 1 + findtreeHeight(node.right)

        else:

            return 1 + findtreeHeight(node.left)

print(findtreeHeight(newNode))

output: 

Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    newNode = Node(n[0],None,None)

    buildTree(newNode,n)

    start = time.time()

    findtreeHeight(newNode)

    stop = time.time()

    return stop - start

n2 = list(range(0,1000,20))

plt2 = plt

plt.plot(n2,n2)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n'])

plt.show()

iput = []

for i in n2:

    temp = [0]

    for j in range(i):

        temp.append(j)

    iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

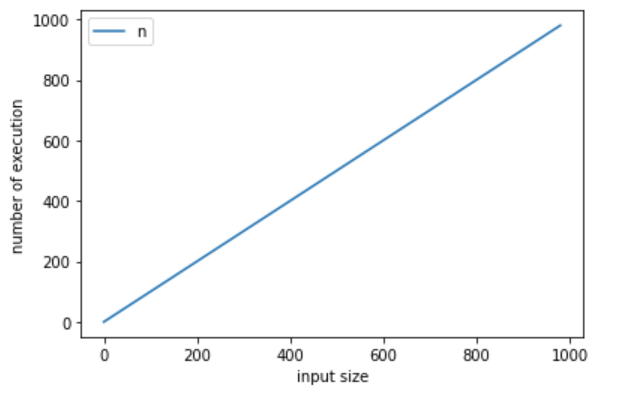
plt2.ylabel('time running in second')

plt2.legend(['find height tree divide and conquer'])

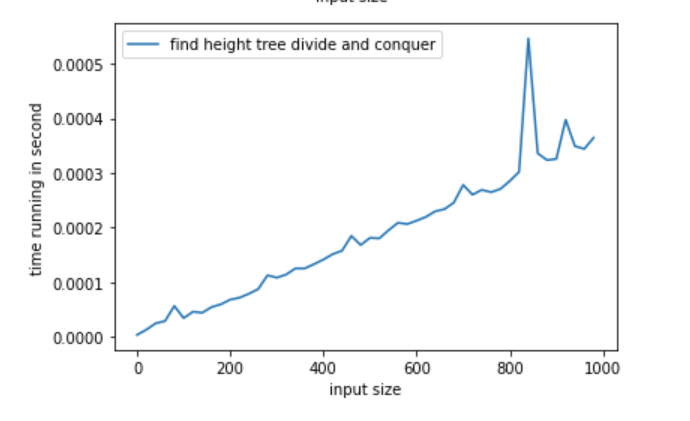
plt2.show()

This time function is creating input list with element is list with size increasing (element inside is random number) remember to set recursion limit

Running time graph theoretical:



Running time graph function Find height:



III. DECREASE AND CONQUER STRATEGY

# 1. Explanation:

Decrease and conquer strategy we can reduce problem instance to smaller instance with the same problem and then solve the smaller instances then we extend the solution for bigger instances.

# 2. Find greatest common divisor decrease and conquer:

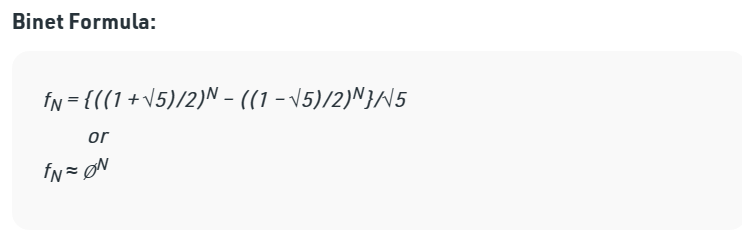
## a. Algorithm idea:

Problem: we find the greatest common divisor between 2 number

For 2 number a and b (a>=b) we divide a to b and get remain then we divide b to remain and repeat this process until remain is 0 (which mean it is divisible) then we stop.

## b. Time complexity:

Let’s assume, the number of steps required to reduce b to 0 using this algorithm is N. Now, if the Euclidean Algorithm for two numbers a and b reduces in N steps then, a should be at least f(N + 2) and b should be at least f(N + 1). where, f(N) is the Nth term in the Fibonacci series(0, 1, 1, 2, 3, …) and N >= 0.  
 We have



Where *∅* is around 1.618

So, to prove the time complexity, it is known that:

*fN ≈ ∅N  
N ≈ log∅(fN)*

Thus, the time complexity is logarithmic based on the sum of a and b —

O(log(a + b)).

=> T(n) ϵ Θlog(a+b)

## c. Code implementation and demo:

code implementation

def findGCD(a,b):

    """find greatest common divisor of 2 number a b"""

    if(b==0):

        return a

    else:

        return findGCD(b,a%b)

a = 68

b = 16

result = findGCD(a,b)

output = 4

print(result)

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import copy

import math

def measure(n):

    start = time.time()

    A = findGCD(n[0],n[1])

    stop = time.time()

    return stop - start

n2 = list(range(5,30)) # list of fibonacii number

plt2 = plt

def Fibonacci(n):

    if n<= 0:

        print("Incorrect input")

    # First Fibonacci number is 0

    elif n == 1:

        return 0

    # Second Fibonacci number is 1

    elif n == 2:

        return 1

    else:

        return Fibonacci(n-1)+Fibonacci(n-2)

listofFibon = []

for i in range(1,n2[-1]+2):

  listofFibon.append(Fibonacci(i))

print(listofFibon)

print(len(listofFibon))

iput = []

realN= []

for i in n2:

  a = listofFibon[i]

  b = listofFibon[i-1]

  realN.append(a+b)

  iput.append([a,b])

plt.plot(realN,[math.log(i) for i in realN])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['log(a+b)'])

plt.show()

running\_time = [measure(x) for x in iput]

plt2.plot(realN,running\_time)

plt2.xlabel('input size')

plt2.ylabel('time running in second')

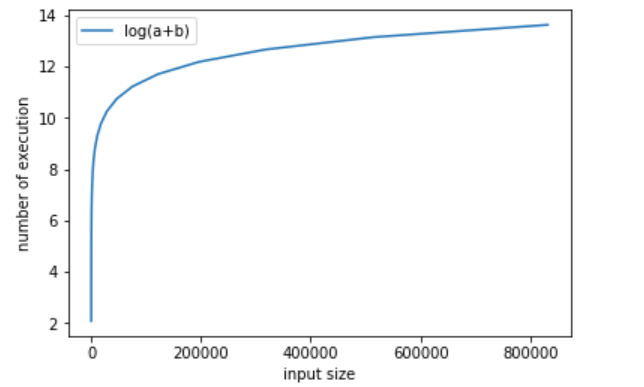
plt2.legend(['GCD divide and conquer'])

plt2.show()

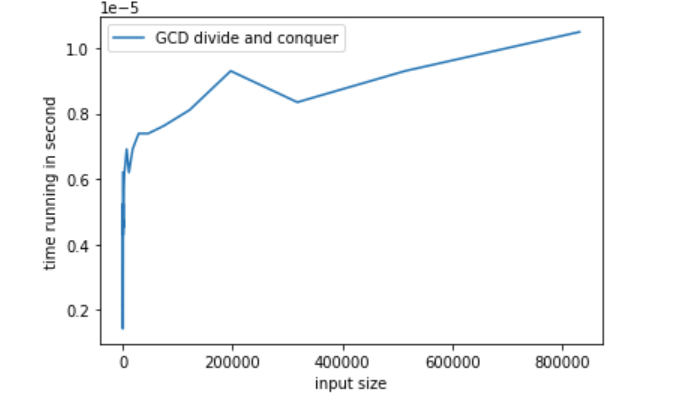
Running time graph Theoretical:

This graph has iput is a and b with a and b is nth Fibonacci number.

Where A is nth number and B is from 3 to nth number. Where n is input size



Running time graph GCD:



# 3. Insertion sort decrease and conquer:

## a. Algorithm idea:

Problem: sorting array

In array A[0…n-1] we divide array to A[0 | 1…n-1] we sort each of the right to the left by compare the element with all element in the left until we can find a perfect slot A [0 1 2 | 3… n-1]. => A[0 1 2 3 … | n-1]

## b. Time complexity:

In the worst case we have to go through all element on the left to find slot for element get in so it will be 1 + 2 + 3 +… + n-1

T(n) = 1 + 2 + 3 + … + n – 1 = n(n-1)/2 = n2 /2 ϵ Θ(n2)

This worst case appeared when array is sorted reversely.

## c. Code implementation and demo:

Code implementation:

def insertSortRecursive(A,n):

    '''

    A is input array and n is len(A)

    this function is to sort an array

    '''

    if n <= 1:

        return

    insertSortRecursive(A,n-1)

    v = A[n-1] #get the last index

    j = n-2

    while(j>=0 and A[j] > v):

        A[j+1] = A[j]

        j = j - 1

    A[j+1] = v

#test case

A = [5,2,8,3,9,6,8,4,2,1]

output = [1,2,2,3,4,5,6,8,8,9]

insertSortRecursive(A,len(A))

print(A)

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    insertSortRecursive(n,len(n))

    stop = time.time()

    return stop - start

n2 = list(range(0,10000,500))

print(n2)

n3 = [i\*\*2 for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n^2'])

plt.show()

iput = []

for i in n2:

    temp = []

    for j in range(i,-1,-1):

      temp.append(j)

    iput.append(temp)

running\_time = list(map(measure,iput))

plt2.plot(n2,running\_time)

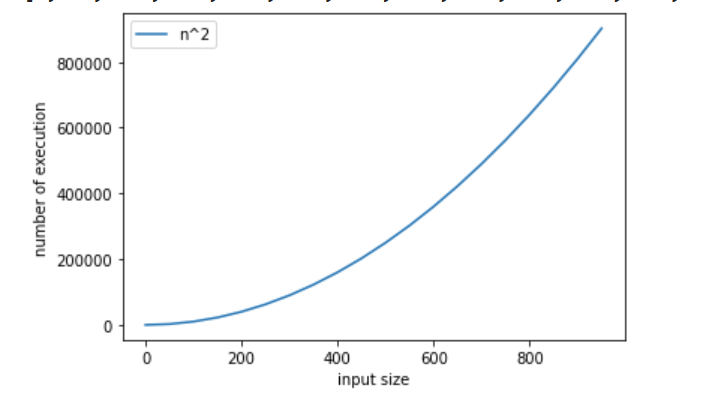
plt2.xlabel('input size')

plt2.ylabel('time running in second')

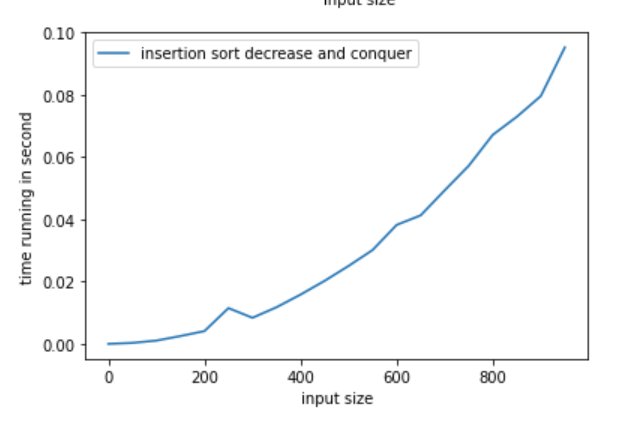
plt2.legend(['Insertion sort decrease an conquer'])

plt2.show()

Running time graph Theoretical:



Running time graph Inserttion sort decrease and conquer:



IV. GREEDY APPROACH STRATEGY

# 1. Explanation:

Greedy technique is that you will create a solution to an optimization problem piece by piece through a sequences of choice that are:

+ feasible and meet the demand of constraints

+ locally optimal

+ greedy (like you get the biggest one remain), and irrevocable

For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations. We are mostly interested in the former case in this class.

# 2. Job Sequence with deadlines greedy approach:

## a. Algorithm idea:

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

You are given a list of jobs, each job has a defined deadline with some profit, you earn profit when you complete the job in its deadline and Processor takes one unit of time to complete a job.

Idea: By applying greedy technique we do following like this:

+ First we sort job with descending profit, we check the value of deadline

+ Second we check the value of maximum deadline, and make a array list size of maximum deadline

+ Third we pick up job by job (job was sorted first step) and put the job as far as possible from 0 deadline base on there deadline. For example, if we have 15 deadline and first job has 8 we place at 8 then if the next job has same 8 deadline we move closer to 0 so place at 7

## b. Time complexity:

Assuming sorting using Mergesort: sorting time is Θ(n logn) in all cases

But I was using Bubble sort Θ(n2). Explain here:

We pick up job by job in the list: that is Θ(n) and each job we travel back from one deadline if slot has been fill up by another job if not then put there and go next job. In worst case that no deadline given so we have all jobs filled with the maximum deadline meaning the last one travel all list of job: that is Θ(n) => Θ(n2) for total

Time complexity: T(n) = Θ(n2) (sorting) + Θ(n2) (filling job)

=> T(n) = Θ(n2) (the same if you use Mergesort so bubble sort was use because it is easy to implemented)

## c. Code implementation and demo:

Code implementation:

def totalprofit(A):

    "sum all profit"

    total = 0

    for i in range(len(A)):

        total = total + A[i][2]

    return total

def sortJob(A):

    #A is the list of job with name,deadline,profit

    #ex: (A,3,6),(B,2,1),(C,2,8)

    for i in range(len(A)):

        for j in range(i,len(A)):

            if(A[i][2]<A[j][2]):

                A[i],A[j] = A[j],A[i]

    return A

    #time complexity 0(n^2)

def findMaxdeadline(A):

    "find the maximum deadline"

    max = A[0][1]

    for i in range(len(A)):

        if max < A[i][1]:

            max = A[i][1]

    return max

def jobSequencing(A):

    A = sortJob(A) #sort the job from highest profit

    result = [(0,0,0)]\*findMaxdeadline(A) #store chosen job

    for i in range(len(A)):

        j = A[i][1] - 1 #index for result base on deadline

        while(result[j]!=(0,0,0) and j>-1):

            #if result is occupied and still have index

            #we shift index until free space to store result

            j=j-1

        #store the reuslt in free space if all space is occupy we go next job

        if(j>=0 and result[j]==(0,0,0)):

            result[j] = A[i]

        else:

            continue

    return result #or return totalprofit(result)

Test case:

#test case

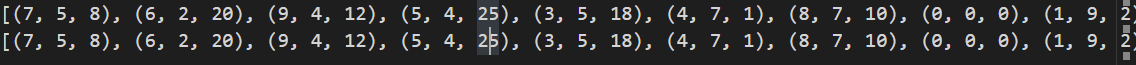
listofJob = [(1,9,2),(2,2,2),(3,5,18),(4,7,1),(5,4,25),(6,2,20),(7,5,8),(8,7,10),(9,4,12),(10,3,5)]

output = [(7,5,8),(6,2,20),(9,4,12),(5,4,25),(3,5,18),(4,7,1),(8,7,10),(0,0,0),(1,9,2)]

print(jobSequencing(listofJob))

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    A = jobSequencing(n)

    stop = time.time()

    return stop - start

n2 = list(range(2,1000,50))

print(n2)

n3 = [i\*\*2 for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.show()

iput = []

for k in n2:

    temp = []

    for i in range(1,k):

        profit = random.randint(1,i+5)

        deadline = random.randint(1,i+5)

        temp.append((i,deadline,profit))

    iput.append(temp)

running\_time = [measure(x) for x in iput]

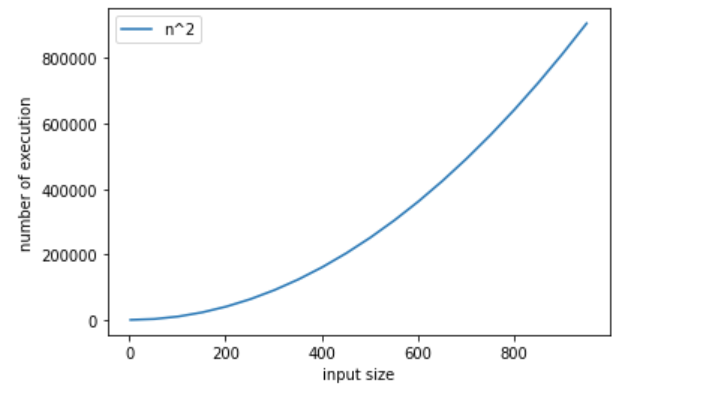
plt2.plot(n2,running\_time)

plt2.xlabel('input size')

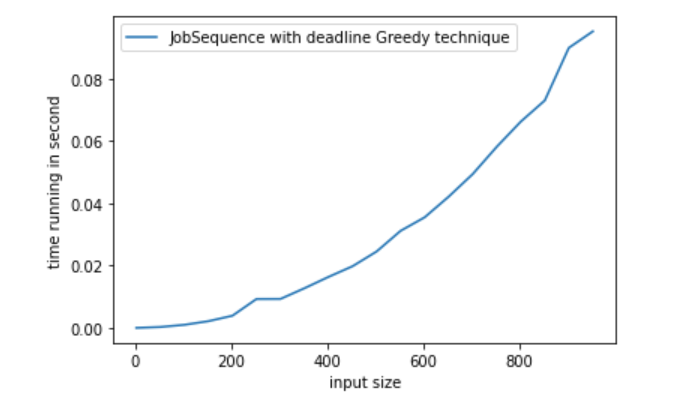
plt2.ylabel('time running in second')

plt2.show()

Running time graph theoretical:



Running time graph JobSequence greedy:



# 3. Kruskal’s Algorithm greedy approach:

## a. Algorithm idea:

Kruskal’s algorithm is an algorithm to find minimum spanning tree for a graph. In this graph we will consider undirected graph with weight. So to find minimum spanning we find all edges possible that not make cycle with the least total weight. In kruskal we will sort all edges in non-decreasing order of their weight. We check that edges will make the graph cycle or not. If not we put in, if it makes cycle we skip it and try the next edges until all edges are consider then we stop.

## b. Time complexity:

when we sorted the edges assuming we use sorting method Mergesort: time complexity: Θ(nlogn) in all case with n is number of edges so I call it Θ(ElogE). The number of edges base on V for example we have 4 Vertices so max E is 6

For each edges take non-decreasing weight Θ(E) and check if it make cycle. In worst case it take Θ(log V) with V is number of vertices. So in total we have:

Time complexity: T(n) = Θ(ElogE) + Θ(ElogV)

=> time complexity is Θ(ElogE) (sorting time) if we use other sorting method. => the complexity change

This time I was sorting using Brute-force method and it takes Θ(n2) to sort => Θ(E2) but we already been seen that sorting exceed the time of finding where it cycle or not so => T(n) in this case is T(E2)

## c. Code implementation and demo:

Code implementation:

def sortingEdges(graph):

    #sorting using bruteforce method

    for i in range(len(graph)):

        min = graph[i][2]

        index = i #initial index of min edge

        for j in range(i+1,len(graph)): # find max

            if min > graph[j][2]:

                min = graph[j][2]

                index = j

        #switch max to the front

        graph[i],graph[index] = graph[index],graph[i]

    return graph

def find(parent, i):

    '''function to find set of an element i'''

    if parent[i] == i:

        return i

    return find(parent, parent[i])

def union(parent, rank, x, y):

        xroot = find(parent, x)

        yroot = find(parent, y)

        # Attach smaller rank tree under root of high rank tree (Union by Rank)

        if rank[xroot] < rank[yroot]:

            parent[xroot] = yroot

        elif rank[xroot] > rank[yroot]:

            parent[yroot] = xroot

        # If ranks are same, then make one as rootand increment its rank by one

        else:

            parent[yroot] = xroot

            rank[xroot] += 1

def KruskalMST(graph):

    '''KrusalMST will find minimum spaning tree with input adjacency matrix

    '''

    #stop duplicate (1 -- 4, w: 6 the same with 4 -- 1, w: 6)

    #we stop by do u -- v with u < v (u is start vertice and v is end vertice)

    newGraph = []

    for i in range(len(graph)):

        for j in range(i,len(graph)):

            if graph[i][j] != 0:

                #add edge follow this format [u,v,w]

                newGraph.append([i,j,graph[i][j]])

    #sorting edges in non-decreasing of the weight

    newGraph = sortingEdges(newGraph)

    parent = []

    rank = []

    V = len(graph) #number of vertices

    # Create V subsets with single elements

    for node in range(V):

        parent.append(node)

        rank.append(0)

    result = [] #store the result

    #create 2 index variable for loop

    i = 0 #i is for sorting edge,

    e = 0 #e is for result[]

    # Number of edges to be taken is equal to V-1

    while e < V - 1:

        # Step 2: Pick the smallest edge and increment

        # the index for next iteration

        u, v, w = newGraph[i]

        i = i + 1

        x = find(parent, u)

        y = find(parent, v)

        # If including this edge does't

        #  cause cycle, include it in result

        #  and increment the indexof result

        # for next edge

        if x != y:

            e = e + 1

            result.append([u, v, w])

            union(parent, rank, x, y)

        # Else discard the edge

    #print the tree

    minimumCost = 0

    print ("Edges in the constructed MST")

    for u, v, weight in result:

        minimumCost += weight

        print("%d -- %d == %d" % (u, v, weight))

    print("Minimum Spanning Tree" , minimumCost)

Test case:

#testcase

myGraph = [[0,4,0,0,0,0,0,8,0],

            [4,0,8,0,0,0,0,11,0],

            [0,8,0,7,0,4,0,0,2],

            [0,0,7,0,9,14,0,0,0],

            [0,0,0,9,0,10,0,0,0],

            [0,0,7,14,10,0,2,0,0],

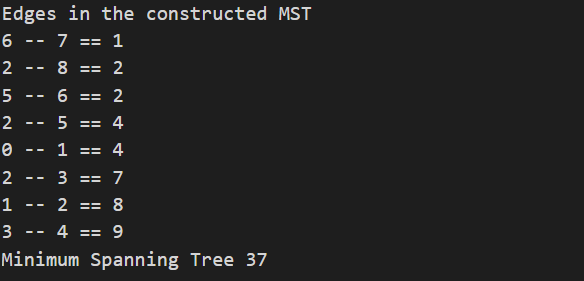
            [0,0,0,0,0,2,0,1,6],

            [8,11,0,0,0,0,1,0,7],

            [0,0,2,0,0,0,6,7,0]]

KruskalMST(myGraph)

Output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    KruskalMST(n)

    stop = time.time()

    return stop - start

n2 = list(range(5,20,1))

n3 = [i\*\*2 for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n^2'])

plt.show()

iput = []

for k in n2:

    temp = [[0]\* k for l in range(k)]

    for i in range(k):

      for j in range(i+1):

        m = random.randint(1,10)

        temp[i][i-j-1] = m

        temp[i-j-1][i] = m

      temp[i][i] = 0

    iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

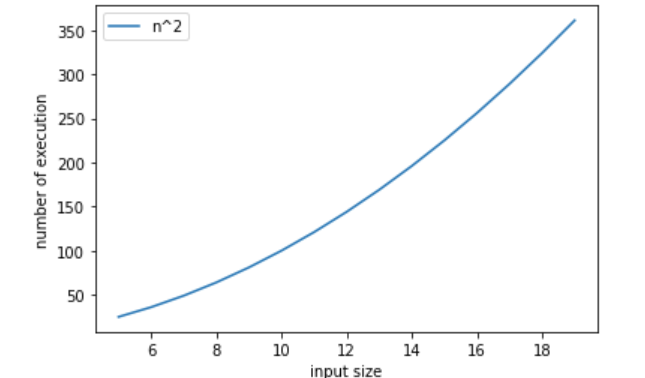
plt2.xlabel('input size')

plt2.ylabel('time running in second')

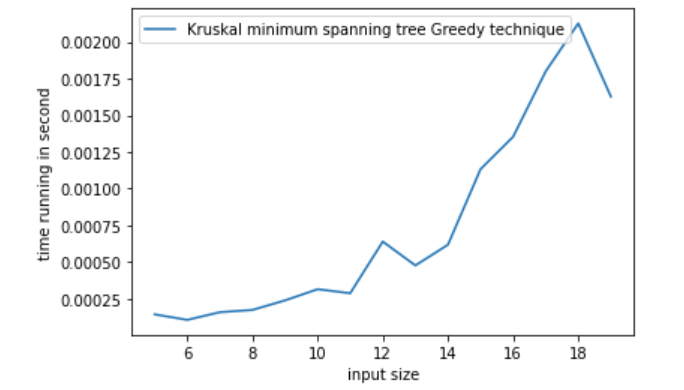
plt.legend(['Kruskal minimum spanning tree Greedy technique'])

plt2.show()

Running time graph theoretical:



Running time graph Kruskal:



V. DYNAMIC PROGRAMMING STRATEGY

# 1. Explanation:

Dynamic Programming is a general algorithm design technique

for solving problems defined by or formulated as recurrences

with overlapping subinstances.

Main idea:

- set up a recurrence relating a solution to a larger instance

to solutions of some smaller instances

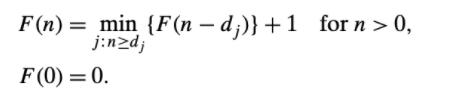
- solve smaller instances once

- record solutions in a table

- extract solution to the initial instance from that table

# 2. Change-Making problem dynamic programming:

## a. Algorithm idea:

Problem: Give change for amount n using the minimum number of coins of denominations d1 < d2 < . . . < dm. For the general case, assuming availability of unlimited quantities of coins for each of the m denominations d1 < d2 < . . . < dm where d1 = 1. Let F (n) be the minimum number of coins whose values add up to n; it is convenient to define F (0) = 0. The amount n can only be obtained by adding one coin of denomination dj to the amount n−dj for j =1,2,...,m such that n≥dj. Therefore, we can consider all such denominations and select the one minimizing F(n − dj) + 1. Since 1 is a constant, we can, of course, find the smallest F(n − dj) first and then add 1 to it. Hence, we have the following recurrence for F (n): 

## b. Time complexity:

In the worst case we might just add 1 in each step. F(0) = 0, F(1) = 1, F(2) = 2. And so on. So at most it its Θ(n) with n is the money. Each step we check if the biggest coin can go if not try the next coin so at most is Θ(D) with d is list of coin. So total the time complexity: T(n) = Θ(n) \* Θ(D) ϵ Θ(Dn)

## c. Code implementaion and demo:

Code implementation:

def change\_making(C,m):

    '''Chang making will comput the minimum coin needed to change

    C is input list of coins value

    m is the input money

    '''

    C.insert(0,0) #insert 0 at the first

    n = len(C) #len c now increase +1

    C.insert(-1,0) #insert -1 at the first

    # if C = [25,10,5,1] now it is C = [-1,0,25,10,5,1]

    F = [(m+1) for i in range(m+1)]

    F[0] = 0

    for i in range(1,m+1):

        temp = m+1

        j = 1

        while j<=n and C[j]<=i:

            #if put the bigger coin in success then go

            #if not go to next coins and start again

            temp = min(F[i-C[j]],temp)

            j+=1

        F[i] = temp + 1

    return F[m]

#test case

Coins = [25,10,5,1]

money = 49

#output (7: 25 + 10 + 10 + 1 + 1 + 1 + 1 = 49)

output = 7

print(change\_making(Coins,money))

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    A = change\_making(n[0],n[1])

    stop = time.time()

    return stop - start

n2 = list(range(0,1000,50))

plt2 = plt

iput = []

xline = []

for j in n2:

  #temp

  temp = [] #list of coin

  for k in range(j):

    temp.append(k)

  iput.append([temp,j+10])

  xline.append([j\*len(temp)])

plt.plot(n2,xline)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n\*m where n ~ m'])

plt.show()

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

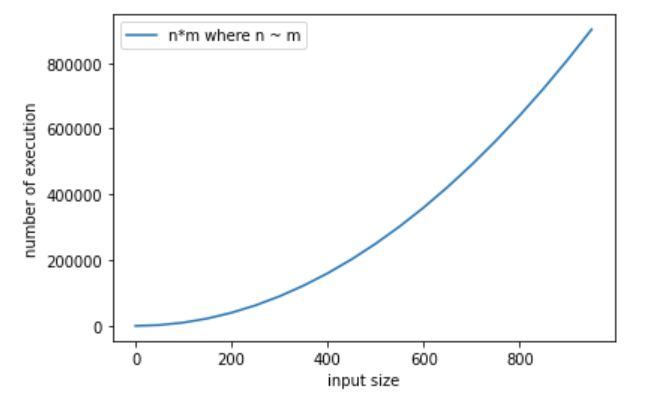
plt2.xlabel('input size')

plt2.ylabel('time running in second')

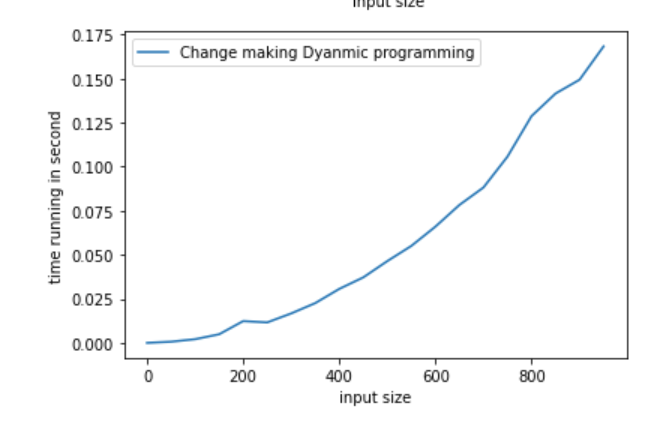
plt2.legend(['Change making Dyanmic programming'])

plt2.show()

Running time graph theoretical:



Running time graph change-making function:



# 3.Longest common sequences dynamic programming:

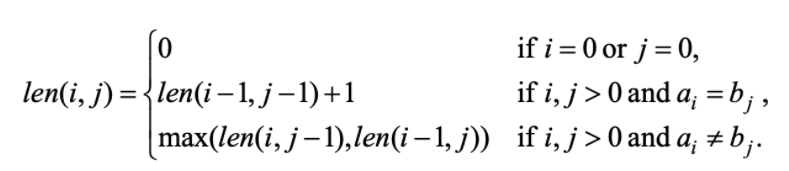
## a. Algorithm idea:

Problem: find a longest common sequence between 2 given string

Idea: Let A = a1, a2,… ai, B = b1,b2, …,bj

Making 2d array len with proper initialization

Then we got len(i ,j) as follow:



## b. Time complexity:

First make first row (i=0) and first column (j=0) of len(i,j) = 0: Θ(n) + Θ(m) where n is size of first string and m is size of second string

Go through the rest element in 2d array: to set len(i,j) base on formula =>

Θ(n\*m)

=> total time complexity: T(n) = Θ(n) + Θ(m) + Θ(n\*m) ϵ Θ(n\*m)

## c. Code implementation and demo:

Code implementation:

def LCS(A,B):

    '''LCS function is to find the Longest Common Subsequence

    with A is first input sentence

    B is second sentence

    '''

    m = len(A)

    n = len(B)

    Len = [[None]\*(n+1) for i in range(m+1)] #for computing common subsequnce

    prev = [[None]\*(n+1) for i in range(m+1)] #for backtracking

    for i in range(m+1):

        Len[i][0] = 0 #make first column 0 as initialize

    for j in range(n+1):

        Len[0][j] = 0 #make first row 0 as initialize

    #computing LCS

    for i in range(1,m+1):

        for j in range(1,n+1):

            if A[i-1] == B[j-1]:

                Len[i][j]=Len[i-1][j-1] + 1

                prev[i][j]="topleft"

            elif Len[i-1][j] >= Len[i][j-1]:

                Len[i][j]=Len[i-1][j]

                prev[i][j]="top"

            else:

                Len[i][j]= Len[i][j-1]

                prev[i][j]="left"

    print("LCS reuslt is: ")

    outputLSC(A,prev,i,j)

    print()

    return Len

#backtracking to print out the common subsequence

def outputLSC(A,prev,i,j):

    if i==0 or j==0:

        return

    if prev[i][j]=="topleft":

        outputLSC(A,prev,i-1,j-1)

        print(A[i-1],end = '')

    elif prev[i][j]=="top":

        outputLSC(A,prev,i-1,j)

    else:

        outputLSC(A,prev,i,j-1)

s1 = "president"

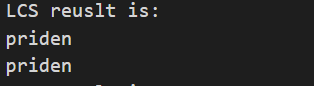
s2 = "providence"

output = "priden"

LCS(s1,s2)

print(output)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    A = LCS(n[0],n[1])

    stop = time.time()

    return stop - start

n2 = list(range(0,50))

plt2 = plt

iput = []

xline = []

yline = []

for i in n2:

  a = ''

  b = ''

  for j in range(i):

    a += 'B'

    b += 'C'

  iput.append([a,b])

  xline.append(len(a)+len(b))

  yline.append(len(a)\*len(b))

running\_time = [measure(x) for x in iput]

plt.plot(xline,yline)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n\*m where n = m'])

plt.show()

plt2.plot(xline,running\_time)

plt2.xlabel('input size')

plt2.ylabel('time running in second')

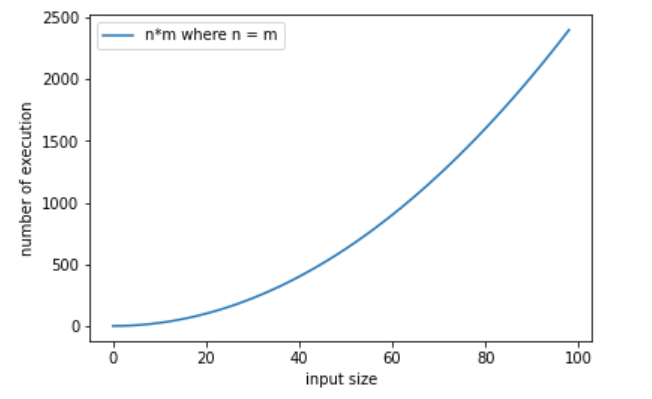
plt2.legend(['LCS of 2 string where len equal'])

plt2.show()

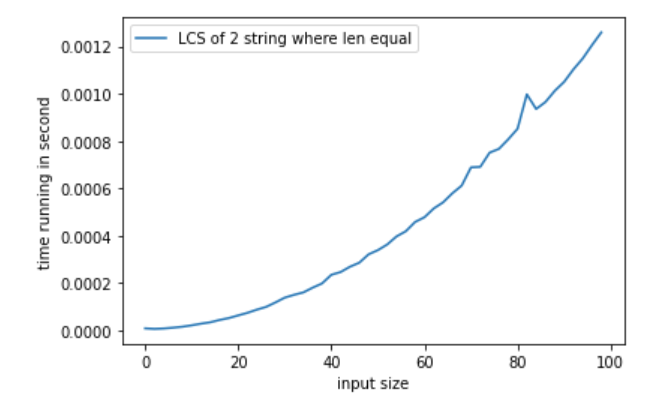
This running time function will put the different string size in to function

Running time graph theoretical:

Because we are having 2 input n and m so input will be 0 to n each time it go 0 to m



Running time graph LCS:



VI. TRANSFORM AND CONQUER STRATEGY

# 1. Explanation:

This method is a group of technique solves problem by transformation:

+ to a simpler/more convenient instance of the same problem (instance simplification)

+ to a different representation of the same instance (representation change)

+ to a different problem for which an algorithm is already available (problem reduction)

# 2. Check uniqueness transform and conquer:

## a. Algorithm idea:

Before transform and conquer we may have to use brute force method loop through each element and compare that element to all element left in list. This take about 2 loop run n-1 times => Θ(n2)

But before checking we already have sorting method. After sort the element that is duplicate will be next to each other. Forexample: [5,3,7,4,3,2] => [2,3,3,4,5,7]. So now we just need 1 more loop to check if the next element is same with current checking element.

## b. Time complexity:

Assumimng we are using Mergesort: time complexity = nlogn (in worst case). After sort we run loop to see if the element and the next element the same until the end. Time complexity is T(n) = Θ(nlogn) + Θ(n) ϵ Θ(nlogn)

## c. Code implementation and demo:

Code implementation: in this case i’m using MergseSort(although worst case is nlogn but still faster than bruteforce method)

#sortting mergesort code

def mergesort(A, i, j):

    '''

    mergesort is sorting list with given starting index i and ending index j

    in merge we divide by 2 smaller and sort again

    '''

    if i == j: return None

    k = (i + j)//2

    mergesort(A,i,k)

    mergesort(A,k+1,j)

    merge(A, i, k, j)

def merge(A,i,k,j):

    '''This function is to merge 2 list after sorting

    '''

    B = [0 for \_ in range(len(A))]

    p1, p2, p3 = i, k+1, i

    while p1 <= k and p2 <= j:

        if A[p1] < A[p2]:

            B[p3] = A[p1]

            p1 += 1

        else:

            B[p3] = A[p2]

            p2 += 1

        p3 += 1

    while p1 <= k:

        B[p3] = A[p1]

        p3 += 1

        p1 += 1

    while p2 <= j:

        B[p3] = A[p2]

        p3 += 1

        p2 += 1

    for r in range(i, j+1):

        A[r] = B[r]

Brute-force approach normally:

def checkUniqueness(A):

    '''

    This function is to check the unique ness of all element in given a list A

    in this case it is into brute-force method

    '''

    for i in range(len(A)-1):

        #check each element is it the same with the rest

        for j in range(i+1,len(A)):

            if(A[i]==A[j]):

                #return immediately if it is duplicate

                return False

    return True

Test case:

#testcase

newList = [3,1,6,8,3,9,0]

output = False #3 is duplicate

print(checkUniqueness(newList))

print(presortCheckUnique(newList))

print(output)

output:



Running time function:

This function also compare brute force and presort method:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import math

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    presortCheckUnique(n)

    stop = time.time()

    return stop - start

def measureBF(n):

    start = time.time()

    checkUniqueness(n)

    stop = time.time()

    return stop - start

n2 = list(range(1,10000,500))

n3 = [i\*math.log(i) for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['nlogn'])

plt.show()

iput = []

for i in n2:

    temp = []

    for j in range(i,-1,-1):

      temp.append(j)

    iput.append(temp)

running\_time = list(map(measure,iput))

running\_time\_BF = list(map(measureBF,iput))

plt2.plot(n2,running\_time,label="transform and conquer")

plt2.plot(n2,running\_time\_BF,label="brute-force")

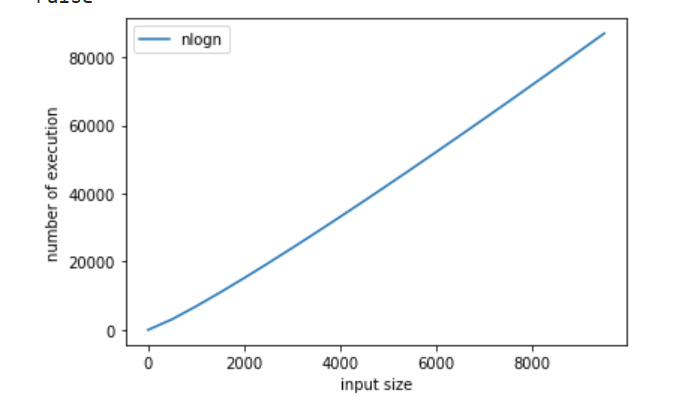
plt2.xlabel('input size')

plt2.ylabel('time running in second')

plt2.legend(['presort','no presort (bruteforce)'])

plt2.show()

Running time graph theoretical:

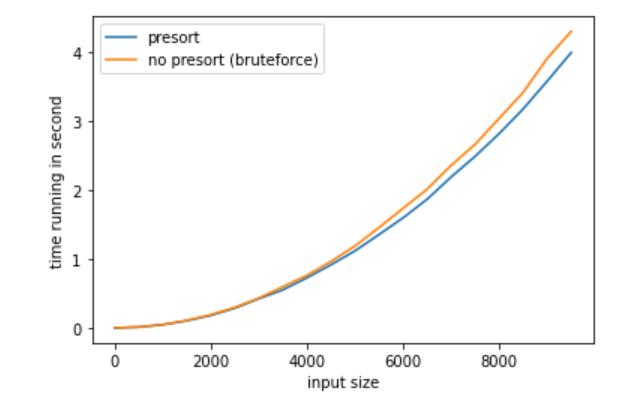


Running time graph using preSort and not using presort to check uniqueness.

The Blue line is with presort Θ(nlogn) + Θ(n) (Transform and conquer)

The Orange line is without presort Θ(n2) (Bruteforce)

We can see that presort has slightly faster



We can see that it is slightly faster for using Transform method instead of Brute force.

# 3. Heapsort transform and conquer:

## a. Algorithm idea:

A heap is a priority queue that is a complete tree. Where the parent node will larger than children node

In transform and conquer method. To sort a list we can use heapsort, which is reuse the heap function that already been done.

Heap-sort is one of the best sorting methods being in-place and with no

quadratic worst-case scenarios. Heap-sort algorithm is divided into two basic

parts.

1. Creating a heap from a (possibly) unsorted list, then

2. a sorted list is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the list. The heap is reconstructed after each removal.

Heap-sort is somehow slower in practice on most machines than a well-implemented quick-sort, but it has the advantage of a more favorable worst-case O(n \*log n) runtime. Heap-sort is an in-place algorithm, but it is not a stable sort.

## b. Time complexity:

To initially build the heap, the method is called for each parent node – backward, starting with the last node and ending at the tree root. A heap of size n has n/2 (rounded down) parent nodes. The heapify function is Θ(logn) and we do heapify for each node so Θ(n). And in heap sort we get the first element (the biggest element if use max heap, smallest if use min heap) then we swap it with last node. And call heapify to make tree heap again. It will take Θ(n) \* Θ(logn).

=> Time complexity T(n) = Θ(n) \* Θ(logn) + Θ(n) \* Θ(logn) ϵ Θ(nlogn).

## c. Code implementation and demo:

code implementation:

def max\_heapify(ls, heap\_size,i):

    """This operation is also sometimes called "push down", "shift\_down" or

    "bubble\_down".

    Time complexity: O(log(n))."""

    m = i

    left = 2 \* i + 1

    right = 2 \* i + 2

    if left < heap\_size and ls[left] > ls[m]:

        m = left

    if right < heap\_size and ls[right] > ls[m]:

        m = right

    if i != m:

        ls[i], ls[m] = ls[m], ls[i]

        max\_heapify(ls, heap\_size, m)

def build\_max\_heap(ls):

    """

    this function is create a heap

    Time complexity: O(n)

    """

    for i in range(len(ls) // 2, -1, -1):

        max\_heapify(ls, len(ls), i)

def heap\_sort(ls):

    build\_max\_heap(ls)

    for i in range(len(ls) - 1, 0, -1):

        ls[i], ls[0] = ls[0], ls[i]

        max\_heapify(ls, i, 0)

Test case:

#test case

newList = [3,2,5,7,4,7,9]

output = [2,3,4,5,7,7,9]

heap\_sort(newList)

print(newList)

print(output)

output:



Running time function:

#time running function

import time

from matplotlib import pyplot as plt

import math

def measure\_time(func, N):

    """

    ...

    """

    runtime = []

    for n in N:

        start = time.time()

        func(n)

        stop = time.time()

        runtime.append(stop-start)

    return runtime

N = list(range(1,20000,500))

plt2 = plt

n2 = [i\*math.log(i) for i in N]

plt.plot(N,n2)

plt.xlabel('input size')

plt.ylabel('time running in second')

plt2.legend(['nlogn'])

plt.show()

iput = []

for i in range(5,len(N)+5):

    iput.append(list(range(i)))

rtime = measure\_time(heap\_sort,iput)

plt2.plot(N,rtime)

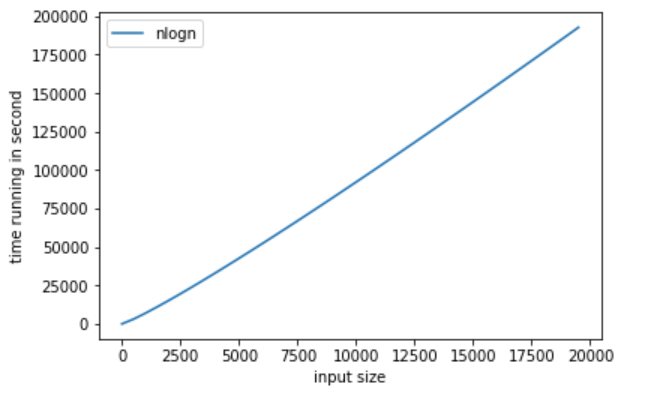
plt2.xlabel('input size')

plt2.ylabel('time running in second')

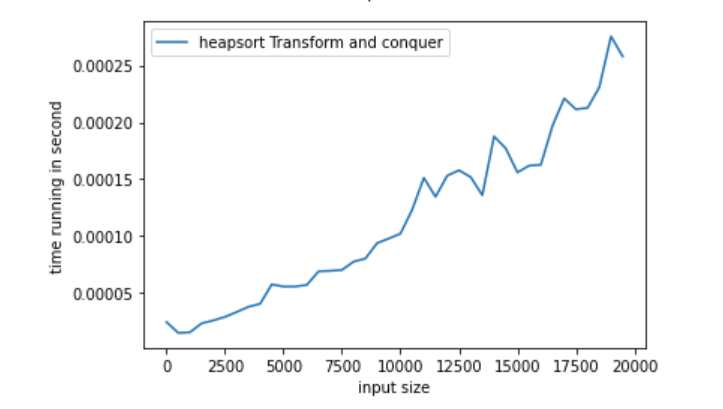
plt2.legend(['heapsort Transform and conquer'])

plt2.show()

Running time graph Theoretical:



Running time graph Heapsort: Θ(nlogn)



VII. ITERATIVE IMPROVEMENT STRATEGY

# 1. Explanation:

This method is that. We start with a feasible solution then we repeat: (change the current feasible solution to a feasible solution with a better) until we have no improvement. We then return the last feasible solution value of the objective function

# 2. Ford-fulkerson for maxflow Iterative improvement:

## a. Algorithm idea:

problem: we have a graph and we need to define the max flow can push in graph

We find an augmenting path in the graph by using DFS. If path is found we increase flow in that path base on bottle neck (the least amount edges) then we try again with other path that still can put flow in. until no path is found left we return the flow which is the max flow of the graph.

## b. Time complexity:

First we find a path using DFS which is Θ(V+E) where E is number of edges in graph and V is number of vertices. Usually E > V so it is Θ(E) Then when we have augment path we increase flow base on bottleneck for this augmenting path. In worst case we may add only 1 unit of flow on the path each time until we reach max flow => Θ(maxflow) time.

In total Time complexity: T(n) = Θ(E) \* Θ(maxflow) ϵ Θ(E\*maxflow)

## c. Code implementation and demo:

Code implementation:

#Ford-Fulkerson Algorithm

#find path by using BFS

def dfs(graph, F, s, t):

        stack = [s]

        paths={s:[]}

        if s == t:

                return paths[s]

        while(stack):

                u = stack.pop()

                for v in range(len(graph)):

                        if(graph[u][v]-F[u][v]>0) and v not in paths: #basic operation

                                paths[v] = paths[u]+[(u,v)]

                                if v == t:

                                        return paths[v]

                                stack.append(v)

        return None

def max\_flow(graph, s, t):

        n = len(graph)

        F = [[0] \* n for i in range(n)] #F is the flow

        path = dfs(graph, F, s, t)

        while path != None: #you can write like while dfs(graph,F,s,t) for short

            flow = min(graph[u][v] - F[u][v] for u,v in path)

            for u,v in path:

                F[u][v] += flow

                F[v][u] -= flow

            path = dfs(graph,F,s,t)

        return sum(F[s][i] for i in range(n))

Test case:

#test case

mygraph = [[0,3,2,0],

        [0,0,3,2],

        [0,0,0,3],

        [0,0,0,0]]

source = 0

sink = 3

outputflow = 5 #3 from 0-1 and 2 from 0-2

max\_flow\_value = max\_flow(mygraph, source, sink)

print("max\_flow\_value is: ", max\_flow\_value)

print(outputflow)

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    A = max\_flow(n,0,len(n)-1)

    stop = time.time()

    return stop - start

n2 = list(range(5,50))

plt2 = plt

plt.plot(n2,[i\*random.randint(i,i+20) for i in n2])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n\*random max flow'])

plt.show()

iput = []

for i in n2:

  newGraph = [[random.randint(1,20) for j in range(i)] for k in range(i)]

  #to make directed graph change some value to 0

  lenG = len(newGraph)

  for k in range(lenG):

    for l in range(k+1):

      newGraph[k][l] = 0

  iput.append(newGraph)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

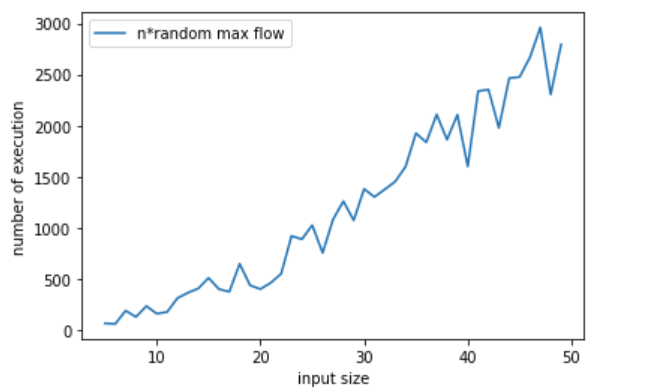
plt2.ylabel('time running in second')

plt2.legend(['Forfulker son with E increase and random maxflow'])

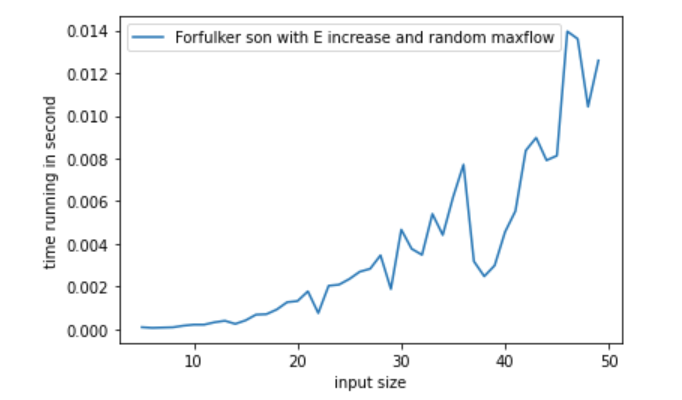
plt2.show()

Running time graph Theoretical:

Because ford-fulkersone require 2 paramenter N (number of edges) and max flow so the graph of Θ(n\*maxflow) should be like this where I set random maxflow



Running time graph Ford-Fulkerson:

Here in this graph I increase number of vertices and connect them with random weight (so number of edges will increase but with random maxflow)   


# 3. N-queen using Iterative improvement method:

## a. Algorithm idea:

N-queen problem is that we place queen in a nxn square board that no other queen is in same row, column or diagnose. In iterative improvement method we using for loop each time to get the queen placed. We check if queen placement is ok if yet we place if not we try other column or row or both until we can find other place. If no placement is found we try another loop to find method.

## b. Time complexity:

Check queen is safe take Θ(n) n is size of board because we need to check row and column and diagnose of queen

First we check column safe to place queen, there are n column => Θ(n). Each time we check if queen is safe to place => Θ(n) (explain above). If not safe we go next row and chek queen is safe again, there are n rows we may have to check safe n time each column => Θ(n) \* Θ(n) but if there is no queen can place in the row we have to undo queen in the last column and try there so worst case we may have to do n-1 more time to find another solution => later on it reduce only 1 case left => T(n) = n\*n-1 \*n-2…

1 => T(n) = Θ(n!)

## c. Code implementation and demo:

Code implementation:

This function is Θ(n!) \* possible configuration because we will print all possible solution

class Board:

    def \_\_init\_\_(self, size):

        self.N = size

        self.queens = [] # list of columns, where the index represents the row

    def is\_queen\_safe(self, row, col):

        for r, c in enumerate(self.queens):

            #check in same row/column/diagnose

            if r == row or c == col or abs(row - r) == abs(col - c):

                '''

                  1 2 3 4

                1 Q

                2

                3     Q

                4

                '''

                return False

        return True

    def print\_the\_board(self):

        print ("solution:")

        for row in range(self.N):

            line = ['.'] \* self.N

            if row < len(self.queens):

                line[self.queens[row]] = 'Q'

            print(''.join(line))

    def solution(self):

        self.queens = []

        col = row = 0

        while True:

            #check is the position good to place queen

            while col < self.N and not self.is\_queen\_safe(row, col):

                #if the full column is unable to place go next column

                col += 1

            #if column is till in size of board

            if col < self.N:

                self.queens.append(col)

                #if row is over than board size

                #place complete

                if row + 1 >= self.N:

                    self.print\_the\_board()

                    self.queens.pop()

                    col = self.N

                else:

                    row += 1

                    col = 0

            if col >= self.N:

                # not possible to place a queen in this row anymore

                if row == 0:

                    return # all combinations were tried

                col = self.queens.pop() + 1

                row -= 1

Testcase:

#test case

q = Board(4)

'''

output =

. Q . .

. . . Q

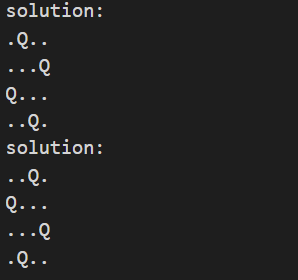
Q . . .

. . Q .

'''

q.solution()

Out put: print all possible solution



Running time function:

Base on n size:

#time running function

import time

import random

from matplotlib import pyplot as plt

def measure(N):

    """

    ...

    """

    start = time.time()

    q = Board(N)

    q.solution()

    stop = time.time()

    return stop - start

N = list(range(3,10))

plt2 = plt

def fac(n):

  fact = 1

  for i in range(1,n+1):

      fact = fact \* i

  return fact

plt.plot(N,[fac(i) for i in N])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n!'])

plt.show()

running\_time = [measure(x) for x in N]

plt2.plot(N,running\_time)

plt2.xlabel('input size')

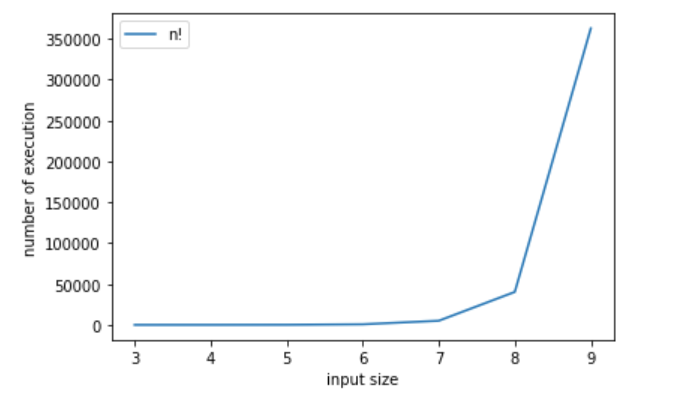
plt2.ylabel('time running in second')

plt2.legend(['N-Queen iterative'])

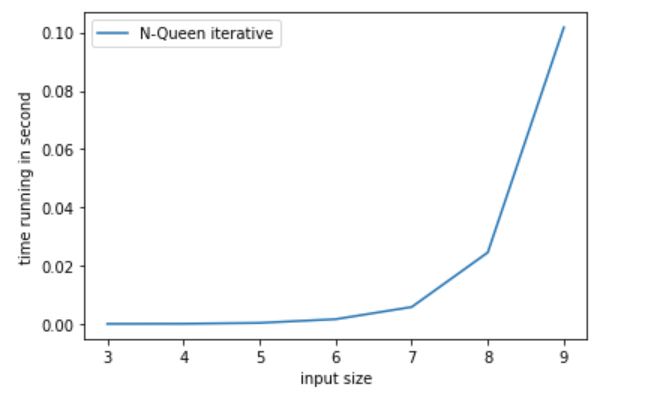
plt2.show()

Running time graph Theoretical:

Due to the fac that N-queen run take so much time so I put small input n



Running time graph N-queen iterative:



VIII. BACKTRACKING STRATEGY

# 1. Explanation:

Backtracking is a form of recursion. But it involves choosing only option out of any possibilities. We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specific option does not give the required solution. We repeat these steps by going across each available option until we get the desired solution.

# 2. N-queen problem using backtracking method:

## a. Algorithm idea:

Instead of solving using iterative improvement we can also use backtracking method to solve N-queen problem.

The idea of placement is the same, we place queen so that no queen in same row or column or diagnose.

The idea of backtracking is that we place each queen in each column then go next column check if this queen place in this column is crash (by row or diagnose then we find another row for this queen until we find solution and return it if no solution then return false.

## b. Time complexity:

A n run time loop to run through each row in colum. If column is placed success we repeat this with next column. But once column is place we don’t check that column anymore we move next and we don’t check that row ether so it is n \* n-1 \* n-2… 1 => n! => in worst case it takes Θ(n!) time to get solution.

=>time complexity: T(n) = Θ(n!)

## c. Code implementation and demo:

class Board:

    def \_\_init\_\_(self, size):

        self.N = size

        self.board = [[0]\*size for i in range(self.N)]

    def printSolution(self, board):

        for i in range(self.N):

            for j in range(self.N):

                print (board[i][j], end = " ")

            print()

    def isSafe(self, row, col):

        #checking column

        for i in range(self.N):

            if self.board[row][i] == 1 or self.board[i][col] == 1:

                return False

        #checking diagonals

        for k in range(self.N):

            for l in range(self.N):

                if (abs(k-l==row-col)):

                    if self.board[k][l]==1:

                        return False

        return True

    def solving(self, col):

        '''

        with backtracking queen problem

        we place each queen in each column then go next column

        check if this queen place in this column is crash (by row or diagnose)

        we find another row for this queen

        until we find solution and return it

        if no solution then return false

        '''

        #base case if column is bigger than size stop

        #col bigger than size when all queen has been place

        if col >= self.N:

            return True

        for i in range(self.N):

            #check the queen placement is good or not

            if self.isSafe(i, col):

                self.board[i][col] = 1

                #after place queen success, go try next column

                if self.solving(col + 1) == True:

                    return True

                #if next column is unable to place:

                #meaning it is not the solution so we undo the queen

                self.board[i][col] = 0

            #go next row try to place queen

        #if the queen cannot be place in anyrow

        #in this column => the solution is wrong

        return False

    def queenSolving(self):

        if self.solving(0) == False:

            print("no solution")

            return False

        self.printSolution(self.board)

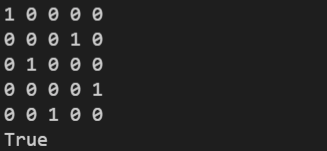
        return True

#test case

q = Board(5)

print(q.queenSolving())

output:



Running time function:

#time running function

import time

import random

from matplotlib import pyplot as plt

def measure(N):

    """

    ...

    """

    start = time.time()

    q = Board(N)

    q.queenSolving()

    stop = time.time()

    return stop - start

N = list(range(3,11))

plt2 = plt

def fac(n):

  fact = 1

  for i in range(1,n+1):

      fact = fact \* i

  return fact

plt.plot(N,[fac(i) for i in N])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['n!'])

plt.show()

running\_time = [measure(x) for x in N]

plt2.plot(N,running\_time)

plt2.xlabel('input size')

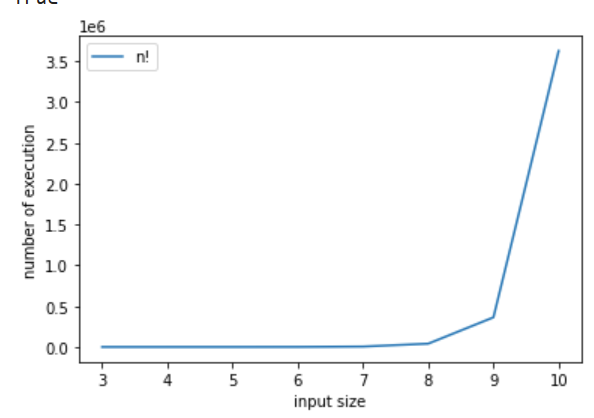
plt2.ylabel('time running in second')

plt2.legend(['N-queen backtracking'])

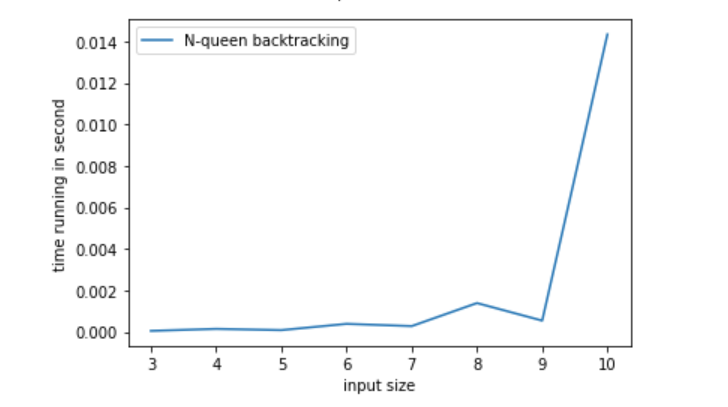
plt2.show()

Running time graph Theoretical:

Because N-queen run takes long time so I put small input



Running time graph N-queen backtracking:



# 3. Subset of sum using backtracking method:

## a. Algorithm idea:

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

We create node with parent. The children node will add the parent value and its value to become value. We check if the value is met d or not (d is the target value) If the value is bigger than d we do nothing. If met return if not continue to check children node, if the total of rest child node bigger than d and the node value is and we still have index of node. We continue backtracking until we got the met

## b. Time complexity:

We check the node value if not we recursive child node with value is its value and parent value until nothing happen we backtracking each time we get that child node out. The worst case is that we run everything but we don’t find the met solution or the solution is at the end. => each time we run 2 recursive method for 1 is get that other is not. Forexample: [1,3,5] => each node we choose to get, get 1 or no 1, then each of that get or not element and spread 2 more.

=> T(n) = 2T(n-1) = 4T(n-2) = … = 2n T(1). Where n is number of element in list.

=> T(n) ϵ Θ(2n)

## c. Code implementation and demo:

Code implementation:

class Node:

    def \_\_init\_\_(self, value, parent, select) -> None:

        """

        creates a node containing value(int), reference to parent node(Node), select(int)

        input:

        value(int): sum of partial subset

        parent(Node): reference to parent node

        select(int): index of the node in the original set (represented as a list), or

        -1 if the node is not selected

        output:

        a node (Node)

        """

        self.value = value

        self.parent = parent

        self.select = select

def print\_combination(node, S):

    if node.parent:

        if node.select >=0:

            print(f"{S[node.select]}", sep="-")

        print\_combination(node.parent,S)

def sum\_of\_subset(node, S, d, index):

    if node.value == d:

        print\_combination(node, S)

    elif(node.value < d) and (index < len(S)) and (node.value + sum(S[index:]) >= d):

        next\_element = S[index]

        child = Node(node.value + next\_element, node, index)

        #backtracking

        sum\_of\_subset(child, S, d, index + 1)

        child = Node(node.value, node, -1)

        sum\_of\_subset(child, S, d, index + 1)

Test case:

#test case

list1 = [3,5,6,7]

target = 15

'output = 3+5+7'

node0 = Node(0,None,0)

sum\_of\_subset(node0, list1, target, 0)

output:



Running time function:

#time running function

import time

import random

from matplotlib import pyplot as plt

def measure(N):

    """

    ...

    """

    start = time.time()

    newNode = Node(0,None,0)

    sum\_of\_subset(newNode,N,len(N)+5,0)

    stop = time.time()

    return stop - start

N = list(range(3,50,5))

plt2 = plt

plt.plot(N,[2\*\*i for i in N])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt2.legend(['2^n'])

plt.show()

iput = []

for i in N:

  temp = []

  for j in range(i):

    temp.append(j)

  iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(N,running\_time)

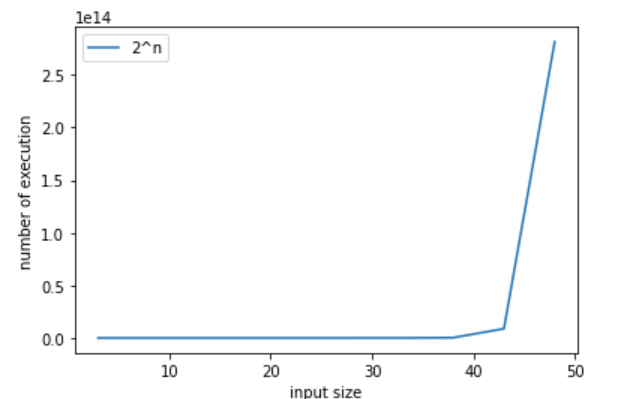
plt2.xlabel('input size')

plt2.ylabel('number of execution')

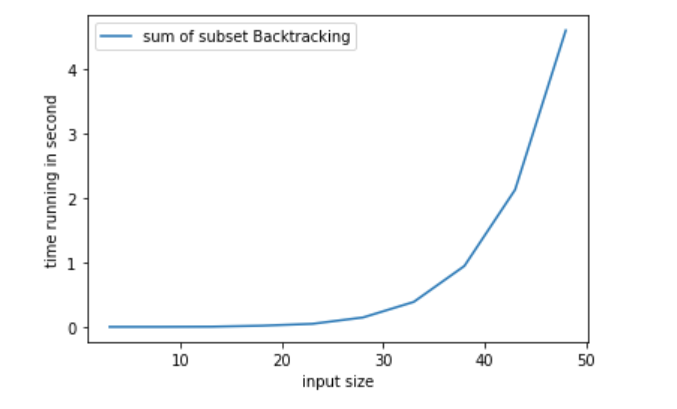
plt2.legend(['sum of sub set backtracking'])

plt2.show

Running time graph Theoretical 2n:



Running time graph subset of sum backtracking:



IX. BRANCH AND BOUND STRATEGY

# 1. Explanation:

Brand and bound strategy has 3 method which is fifo method (using stack), lifo method (using queue) and Least cost method (LC-BB).

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

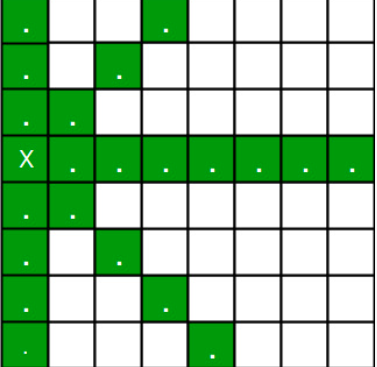
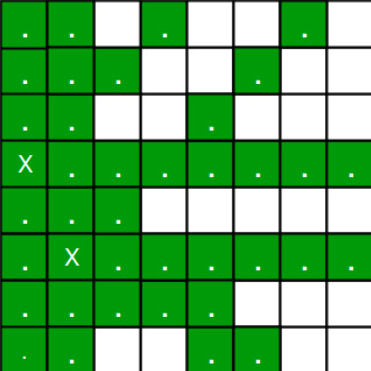
Brand and bound usually using BFS.

# 2. N-queen using brand and bound method:

## a. Algorithm idea:

In Backtracking method, we solve N-queen by placing queen in the leftmost column and next column if row is unsafe then next row until all queens placed.

But here is the problem:

We can see that when we place 1st queen there are 8 postion but the second queen will be place on second column but can’t same row or diagnose with 1st queen so in this case second queen only have 5 position left. And if we see that in third column we only have 3 postion left.

=> So we need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8­-by­-8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells.

Basically, we have to ensure 4 things:

1. No two queens share a column.

**2. No two queens share a row.**

3. No two queens share a top-right to left-bottom diagonal.

4. No two queens share a top-left to bottom-right diagonal.

The idea is that we make 3 boolean array as row, 2 diagnose way (top right to bottom left and bottom left to top rightto mark that it is not safe position.

First we make NxN array for / diagnose matrix and \ diagnose matrix.

Let’s call them slashCode and backslashCode respectively. The trick is to fill them in such a way that two queens sharing a same /­diagnose will have the same value in matrix slashCode, and if they share same \­diagnose, they will have the same value in backslashCode matrix. We can fill it through this formula:

slashCode[row][col] = row + col

backslashCode[row][col] = row – col + (N-1)

## b. Time complexity:

Because we change the check isSafe function so now, it takes Θ(n2) time to initialize the helper matrix.

Then we solving using backtracking algorithm. We check each colume is it good to place queen. Θ(n). If good we check safe function Θ(1) then recursion for next queen column. After that if no solution found we redo Q backtracking. So it is

T(n) = n2 + n\*n-1\*n-2\*n-3 = n2 + n! ϵ Θ(n!)

Although it is the same complexity with backtracking method. It perform faster.

## c. Code implementation and demo:

Code implementation:

class Board:

    def \_\_init\_\_(self, size):

        self.N = size

        self.board = [[0]\*self.N for i in range(self.N)]

    def printSolution(self, board):

        for i in range(self.N):

            for j in range(self.N):

                print (board[i][j], end = " ")

            print()

    def isSafe(self, row, col, slashCode, backslashCode,

        rowLookup, slashCodeLookup, backslashCodeLookup):

        #checking column

        if (slashCodeLookup[slashCode[row][col]]):

            return False

        elif backslashCodeLookup[backslashCode[row][col]]:

            return False

        elif rowLookup[row]:

            return False

        return True

    def solving(self, col, slashCode, backslashCode,

        rowLookup, slashCodeLookup, backslashCodeLookup):

        '''

        Solving N-queen problem by using Brand-and-bound method

        '''

        if col >= self.N: #when all queen has been placed

            return True

        for i in range(self.N):

            #check the queen placement is good or not

            if self.isSafe(i, col,slashCode, backslashCode,

        rowLookup, slashCodeLookup, backslashCodeLookup):

                self.board[i][col] = 1

                rowLookup[i] = True #check row is occupied

                slashCodeLookup[slashCode[i][col]] = True

                backslashCodeLookup[backslashCode[i][col]] = True

                #after place queen success, go try next column

                if self.solving(col + 1,slashCode, backslashCode,

                rowLookup, slashCodeLookup, backslashCodeLookup) == True:

                    return True

                #if next column is unable to place:

                #meaning it is not the solution so we undo the queen

                self.board[i][col] = 0

                rowLookup[i] = False

                slashCodeLookup[slashCode[i][col]] = False

                backslashCodeLookup[backslashCode[i][col]] = False

            #go next row try to place queen

        #if the queen cannot be place in anyrow

        #in this column => the solution is wrong

        return False

    def queenSolving(self):

        #create slashBoard and backslashboard

        slashCode = [[0 for j in range(self.N)] for i in range(self.N)]

        backslashCode = [[0 for j in range(self.N)] for i in range(self.N)]

        # keep two arrays to tell us

        # which diagonals are occupied

        x = 2 \* self.N - 1

        slashCodeLookup = [False] \* x

        backslashCodeLookup = [False] \* x

        # arrays to tell us which rows are occupied

        rowLookup = [False] \* self.N

        # initialize helper matrices

        for rr in range(self.N):

            for cc in range(self.N):

                slashCode[rr][cc] = rr + cc

                backslashCode[rr][cc] = rr - cc + self.N-1

        if(self.solving(0, slashCode, backslashCode,

                        rowLookup, slashCodeLookup,

                        backslashCodeLookup) == False):

            print("no solution")

            return False

        self.printSolution(self.board)

        return True

Test case:

#test case

q = Board(5)

'''

output:

Q . . . .

. . . Q .

. Q . . .

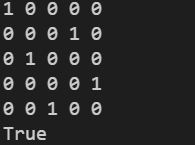
. . . . Q

. . Q . .

'''

print(q.queenSolving())

Output:



Running time function:

#time running function

import time

import random

from matplotlib import pyplot as plt

def measure(N):

    """

    ...

    """

    start = time.time()

    q = Board(N)

    q.queenSolving()

    stop = time.time()

    return stop - start

N = list(range(3,21))

plt2 = plt

def fac(n):

  fact = 1

  for i in range(1,n+1):

      fact = fact \* i

  return fact

plt.plot(N,[fac(i) for i in N])

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n!'])

plt.show()

running\_time = [measure(x) for x in N]

plt2.plot(N,running\_time)

plt2.xlabel('input size')

plt2.ylabel('time running in second')

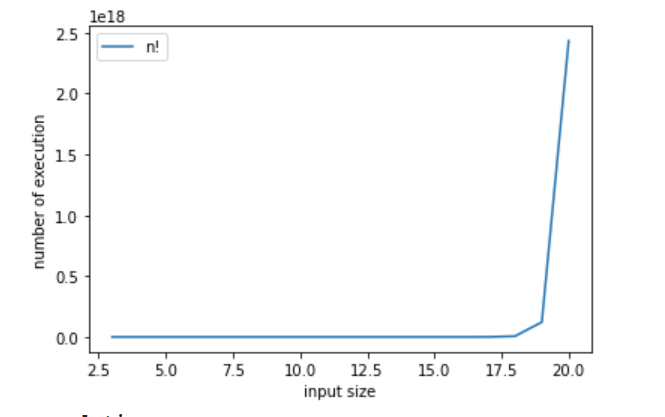
plt2.legend(['N-queen brand and bound'])

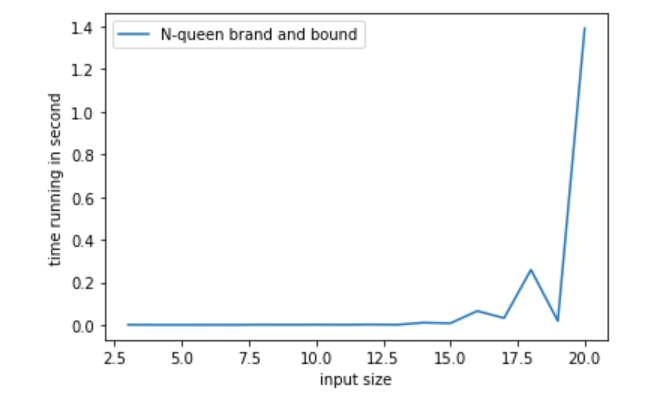
plt2.show()

Note: I tried 50x50 board so time run is very slow (if you want to test the code change the line N = list(range(n)) with n is the size nxn board for faster test

Running time graph theoretical:

Note this graph was running with input size from 0 to 21 coz N-queen take long time

  
Running time graph N-queen brand and bound:



# 3. Traveling sale man problem using brand and bound method:

## a. Algorithm idea:

in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)

2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

In traveling sale man problem. We consider each vertice is a node and the distance is a cost.

Cost of Tour T = ½ \* Sum of cost of two edges adjacent to u and in the tour T where u ∈ V. For every vertex u, if we consider two edges through it in T, and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

Cost of any tour >= (½ \* Sum of cost of two minimum weight edges adjacent to u) where u ∈ V

## b. Time complexity:

First we set bound base on number of vertices => Θ(V)

Second there are V level => Θ(V) (each level present a place to visit). Each level we if the lower bound is still not met we recursive but less than 1 level. Then set visit for each level => Θ(n). => Time complexity: Θ(V) + Θ(V\*V-1\*V-2\*…\*1)\*Θ(V) ϵ Θ(V\*V!) = Θ(V!).

## c. Code implementation and demo:

Code implementation:

import math

maxsize = float('inf')

def copyToFinal(curr\_path):

    '''

    Function to copy temporary solution to the final solution

    '''

    final\_path[:N + 1] = curr\_path[:]

    final\_path[N] = curr\_path[0]

def firstMin(adj, i):

    '''

    Function to find the minimum edge cost having an end at the vertex i

    proper saying finding min in row

    '''

    min = maxsize #set min to infinity

    for k in range(N):

        if adj[i][k] < min and i != k:

            min = adj[i][k]

    return min

def secondMin(adj, i):

    '''

    function to find the second minimum edge cost having an end at the vertex i

    '''

    temp, min = maxsize, maxsize

    for j in range(N):

        if i == j:

            continue

        if adj[i][j] <= temp:

            min = temp

            temp = adj[i][j]

        elif(adj[i][j] <= min):

            min = adj[i][j]

    return min

def TSPRec(adj, curr\_bound, curr\_weight, level, curr\_path, visited):

    '''function that takes as arguments:

    curr\_bound -> lower bound of the root node

    curr\_weight-> stores the weight of the path so far

    level-> current level while moving in the search space tree

    curr\_path[] -> where the solution is being stored which would later be copied to final\_path[]

    '''

    global final\_res

    # base case is when we have reached level N

    # which means we have covered all the nodes once

    if level == N:

        # check if there is an edge from

        # last vertex in path back to the first vertex

        if adj[curr\_path[level - 1]][curr\_path[0]] != 0:

            # curr\_res has the total weight

            # of the solution we got

            curr\_res = curr\_weight + adj[curr\_path[level - 1]]\

                                        [curr\_path[0]]

            if curr\_res < final\_res:

                copyToFinal(curr\_path)

                final\_res = curr\_res

        return

    # for any other level iterate for all vertices

    # to build the search space tree recursively

    for i in range(N):

        # Consider next vertex if it is not same

        # (diagonal entry in adjacency matrix and

        #  not visited already)

        if (adj[curr\_path[level-1]][i] != 0 and

                            visited[i] == False):

            temp = curr\_bound

            curr\_weight += adj[curr\_path[level - 1]][i]

            # different computation of curr\_bound

            # for level 2 from the other levels

            if level == 1:

                curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) +

                                firstMin(adj, i)) / 2)

            else:

                curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) +

                                 firstMin(adj, i)) / 2)

            # curr\_bound + curr\_weight is the actual lower bound

            # for the node that we have arrived on.

            # If current lower bound < final\_res,

            # we need to explore the node further

            if curr\_bound + curr\_weight < final\_res:

                curr\_path[level] = i

                visited[i] = True

                # call TSPRec for the next level

                TSPRec(adj, curr\_bound, curr\_weight,

                       level + 1, curr\_path, visited)

            # Else we have to prune the node by resetting

            # all changes to curr\_weight and curr\_bound

            curr\_weight -= adj[curr\_path[level - 1]][i]

            curr\_bound = temp

            # Also reset the visited array

            visited = [False] \* len(visited)

            for j in range(level):

                if curr\_path[j] != -1:

                    visited[curr\_path[j]] = True

# This function sets up final\_path

def TSP(adj):

    '''

    Calculate initial lower bound for the root node

    using the formula 1/2 \* (sum of first min + second min)

    for all edges. Also initialize the curr\_path and visited array

    '''

    curr\_bound = 0

    curr\_path = [-1] \* (N + 1)

    visited = [False] \* N

    # Compute initial bound

    for i in range(N):

        curr\_bound += (firstMin(adj, i) +

                       secondMin(adj, i))

    # Rounding off the lower bound to an integer

    curr\_bound = math.ceil(curr\_bound / 2)

    # We start at vertex 1 so the first vertex

    # in curr\_path[] is 0

    visited[0] = True

    curr\_path[0] = 0

    # Call to TSPRec for curr\_weight

    # equal to 0 and level 1

    TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited)

Test case:

# test case

adj = [[0, 10, 15, 20],

       [10, 0, 35, 25],

       [15, 35, 0, 30],

       [20, 25, 30, 0]]

'''

out put is 0 1 3 2 0 with total is 80

'''

N = 4

final\_path = [None] \* (N + 1)# final\_path[] stores the final solution

final\_res = maxsize# Stores the final minimum weight

TSP(adj)

print("Minimum cost :", final\_res)

print("Path Taken : ", end = ' ')

for i in range(N + 1):

    print(final\_path[i], end = ' ')

output:



Running function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    global N

    N = len(n)

    final\_path = [None] \* (N + 1)# final\_path[] stores the final solution

    final\_res = maxsize# Stores the final minimum weight

    TSP(n)

    stop = time.time()

    return stop - start

n2 = list(range(5,100,5))

n3 = [i\*\*2 for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n^2'])

plt.show()

iput = []

for k in n2:

    temp = [[1]\*k for i in range(k)]

    #set diagnose is zero

    for i in range(len(temp)):

      temp[i][i] = 0

    iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

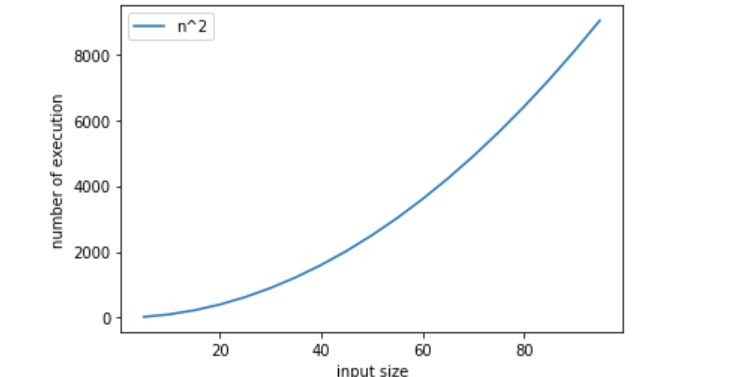
plt2.xlabel('input size')

plt2.ylabel('time running in second')

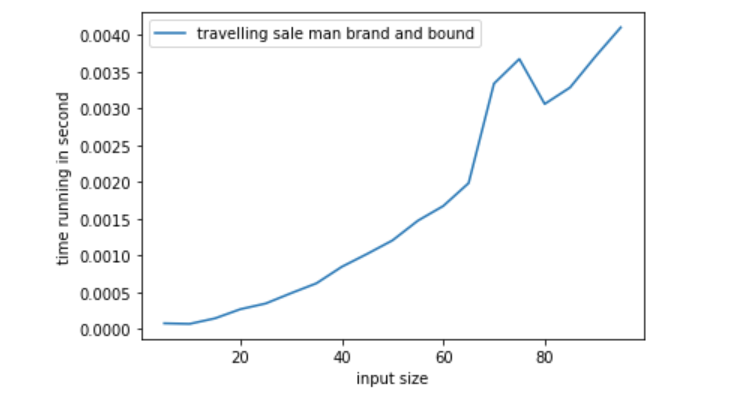
plt.legend(['travelling sale man brand and bound'])

plt2.show()

Running time graph theoretical:



Running time graph travelling sale man brand and bound:



X. APPROXIMATION ALGORITHM STRATEGY

# 1. Explanation:

An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. The goal of the approximation algorithm is to come close as much as possible to the optimal solution in polynomial time.

An approximation algorithm guarantees to run in polynomial time though it does not guarantee the most effective solution.

An approximation algorithm guarantees to seek out high accuracy and top quality solution(say within 1% of optimum)

Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time

# 2. Subset of sum approximate method:

## a. Algorithm idea:

In this method, we also follow the recursive approach but In this method, we use another 2-D matrix in we first initialize with -1 or any negative value.

In this method, we avoid the few of the recursive call which is repeated itself that’s why we use 2-D matrix. In this matrix we store the value of the previous call value.

## b. Time complexity:

Because we recursive back until sum == 0 or n == 0 with sum is the target sum and n is list size and each time we do 2 calls one where descrease size list by, other is decrease sum value by 1. In worst case sum value – 1 only. => Time complexity: T(n,sum) = Θ(n)\*Θ(sum) ϵ Θ(n\*sum).

## c. Code implementation and demo:

# Check if possible subset with

# given sum is possible or not

def subsetSum(a, n, sum):

    # If the sum is zero it means

    # we got our expected sum

    if (sum == 0):

        return 1

    if (n <= 0):

        return 0

    # If the value is not -1 it means it

    # already call the function

    # with the same value.

    # it will save our from the repetition.

    if (tab[n - 1][sum] != -1):

        return tab[n - 1][sum]

    # if the value of a[n-1] is

    # greater than the sum.

    # we call for the next value

    if (a[n - 1] > sum):

        tab[n - 1][sum] = subsetSum(a, n - 1, sum)

        return tab[n - 1][sum]

    else:

        # Here we do two calls because we

        # don't know which value is

        # full-fill our criteria

        # that's why we doing two calls

        tab[n - 1][sum] = subsetSum(a, n - 1, sum)

        return tab[n - 1][sum] or subsetSum(a, n - 1, sum - a[n - 1])

Test case:

n = 5

a = [1, 5, 3, 7, 4]

sum = 12

sum2 = 1 + 5 + 3 + 7 + 4 + 1 #more than sum of a => No

tab = [[-1 for i in range(sum\*n)] for j in range(sum\*n)]

if (subsetSum(a, n, sum)):

    print("YES")

else:

    print("NO")

if (subsetSum(a, n, sum2)):

    print("YES")

else:

    print("NO")

Output:



Running time:

#time running function

import time

from matplotlib import pyplot as plt

import random

def measure\_time(func,N):

    """

    ...

    """

    runtime = []

    for n in N:

        start = time.time()

        tempSum = 20 \* n[0]

        tab = [[-1 for k in range(2000)] for l in range(2000)]

        func(n,len(n),tempSum)

        stop = time.time()

        runtime.append(stop-start)

    return runtime

N = list(range(50))

iput = []

for i in range(10,len(N)+10):

    temp = []

    for j in range(i):

        temp.append(random.randint(i,i+1))

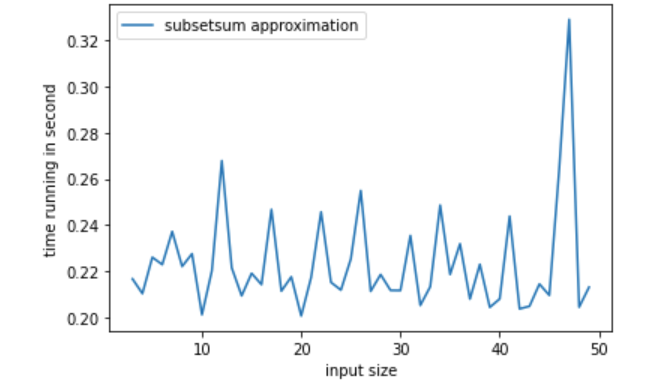
    iput.append(list(set(temp)))

rtime = measure\_time(subsetSum,iput)

plt.plot(N,rtime)

plt.show()

Running time graph sum of subset approximation:



# 3. Travelling sale man approximate method:

## a. Algorithm idea:

Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.

Generate all (n-1)! permutations of cities.

Calculate the cost of every permutation and keep track of the minimum cost permutation.

Return the permutation with minimum cost.

## b. Time complexity:

First we make permuataion base on n => Θ(n)

=> input n will have n! permutation

Then each permuation we loop through each element and each element we loop through each vertex in it.

=> T(n) = n + n!\*n = n + (n-1)! \* n2  = Θ(n!)

## c. Code implementation and demo:

Code Implementation:

from itertools import permutations

maxsize = float('inf')

V = 4

# implementation of traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex = []

    for i in range(V):

        if i != s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path = maxsize

    next\_permutation=permutations(vertex)

    for i in next\_permutation:

        # store current Path weight(cost)

        current\_pathweight = 0

        # compute current path weight

        k = s

        for j in i:

            current\_pathweight += graph[k][j]

            k = j

        current\_pathweight += graph[k][s]

        # update minimum

        min\_path = min(min\_path, current\_pathweight)

    return min\_path

Testcase:

# matrix representation of graph

graph = [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

start = 0

'''

out put way go: 0 - 1 - 3 - 2 - 0 total is 80

'''

print(travellingSalesmanProblem(graph, start))

output:



Running time function:

#time running demo:

import numpy as np

from matplotlib import pyplot as plt

import time

import sys

import random

sys.setrecursionlimit(2000000) #so that using recursion with large number is allow

def measure(n):

    start = time.time()

    global V

    V = len(n)

    travellingSalesmanProblem(n,0)

    stop = time.time()

    return stop - start

n2 = list(range(5,10,1))

def fac(n):

  if n==1:

    return 1

  return n\*fac(n-1)

n3 = [fac(i) for i in n2]

plt2 = plt

plt.plot(n2,n3)

plt.xlabel('input size')

plt.ylabel('number of execution')

plt.legend(['n^2'])

plt.show()

iput = []

for k in n2:

    temp = [[0]\* k for l in range(k)]

    for i in range(k):

      for j in range(i+1):

        m = random.randint(1,10)

        temp[i][i-j-1] = m

        temp[i-j-1][i] = m

      temp[i][i] = 0

    iput.append(temp)

running\_time = [measure(x) for x in iput]

plt2.plot(n2,running\_time)

plt2.xlabel('input size')

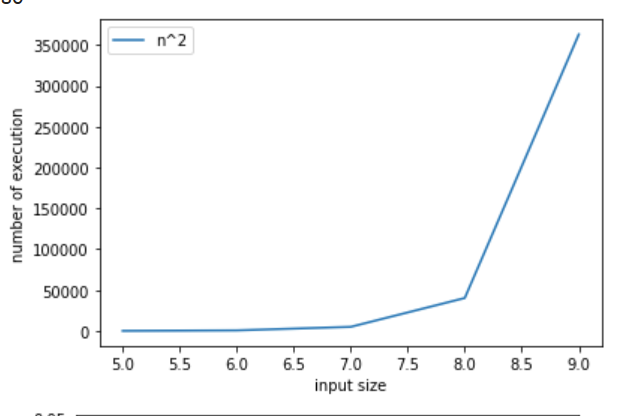
plt2.ylabel('time running in second')

plt.legend(['travelling sale man brand and bound'])

plt2.show()

This time I tried small size

Running time graph:



Running time graph Travelling sale man approximation

