19CSE302 - Design and Analysis of Algorithms

Name : Sriram S and Suryanarayan B

Roll : CB.EN.U4CSE19055 and CB.EN.U4CSE19056

Date : 10/11/2021

Greedy Algorithm

* Code

# function to return the maximum possible denomination for a given amount

def getFloor(coins, amount):

start = 0

end = len(coins) - 1

while(start <= end):

mid = (start + end)//2

if coins[mid] == amount:

return mid

elif amount < coins[mid]:

end = mid - 1

else:

start = mid + 1

return end

def minimum\_coins\_greedy(coins, amount):

count = 0

coins.sort()

while amount > 0:

x = getFloor(coins, amount)

amount -= coins[x]

count += 1

return count

def main():

amount = int(input("Enter the Total Amount : "))

print('Enter denominations (space separated input)')

coins = list(map(int,input().split()))

result = minimum\_coins\_greedy(coins, amount)

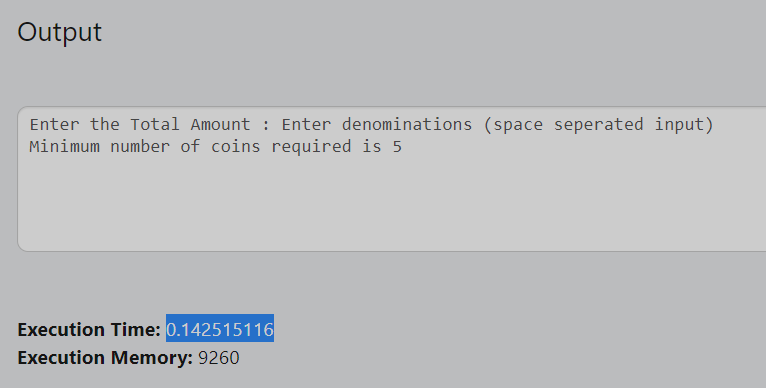
print("Minimum number of coins required is %s" % result)

main()

* Time Complexity

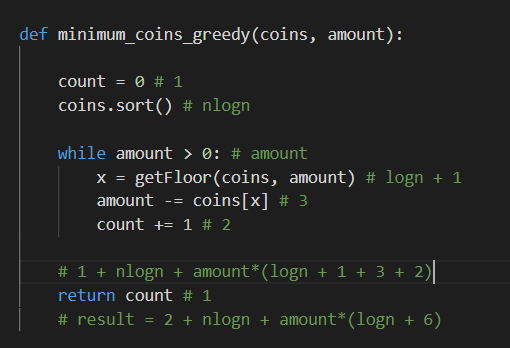
O(Amount) : We perform sorting on the coins array, which is O(NlogN). But since the amount >> len(coins), Amount being the most dominating term, time complexity is O(Amount)

* Time Taken in s = 0.1425 s



* Number of steps required to get the output

No of steps = 2 + nlogn + amount\*(logn + 6)



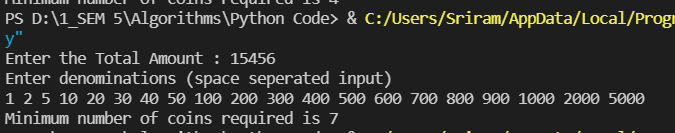
* Correctness of solution

This greedy algorithm may work for some smaller/larger inputs. But it is not a completely correct solution as we can find multiple counter examples for this approach

* Working example

Denomination of size = 20

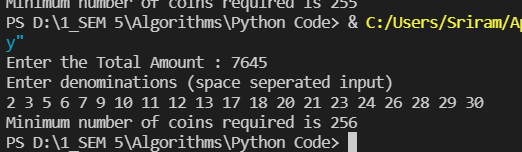
Result arrived = expected result. Hence it works for this large input



* Counter Examples

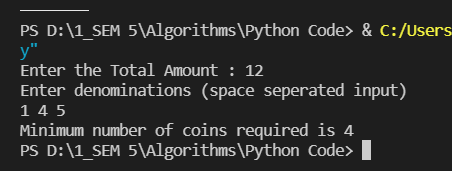
Denomination of size = 20

Expected result = 255. But the result through the greedy approach = 256. Hence for large inputs it fails



Denomination of size = 3

Expected result = 3. But the result through the greedy approach = 4. Hence for small inputs it fails as well



Dynamic Programming

* Code

from sys import maxsize as INF

def minimum\_coins\_dp(coins,amount):

n = len(coins)

dp = [[0 for i in range(amount+1)] for i in range(n+1)]

for i in range(n+1):

for j in range(amount+1):

if j == 0:

dp[i][j] = 0

elif i == 0:

dp[i][j] = INF

elif (coins[i-1]>j):

dp[i][j] = dp[i-1][j]

else:

dp[i][j] = min(1+dp[i][j-coins[i-1]],dp[i-1][j])

if dp[n][amount] > INF:

return -1

else:

return dp[n][amount]

def main():

amount = int(input("Enter the Total Amount : "))

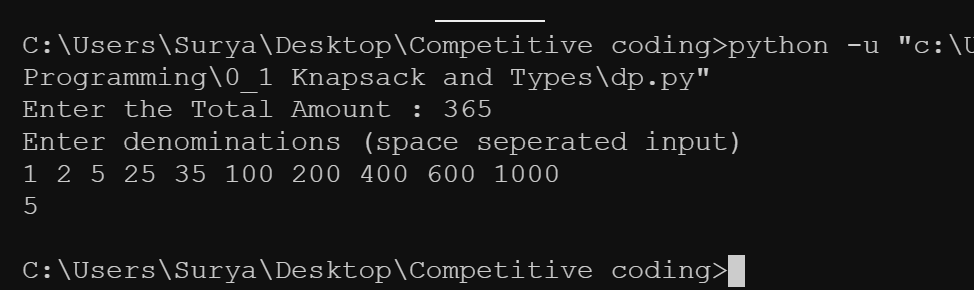
print('Enter denominations (space seperated input)')

coins = list(map(int,input().split()))

result = minimum\_coins\_dp(coins,amount)

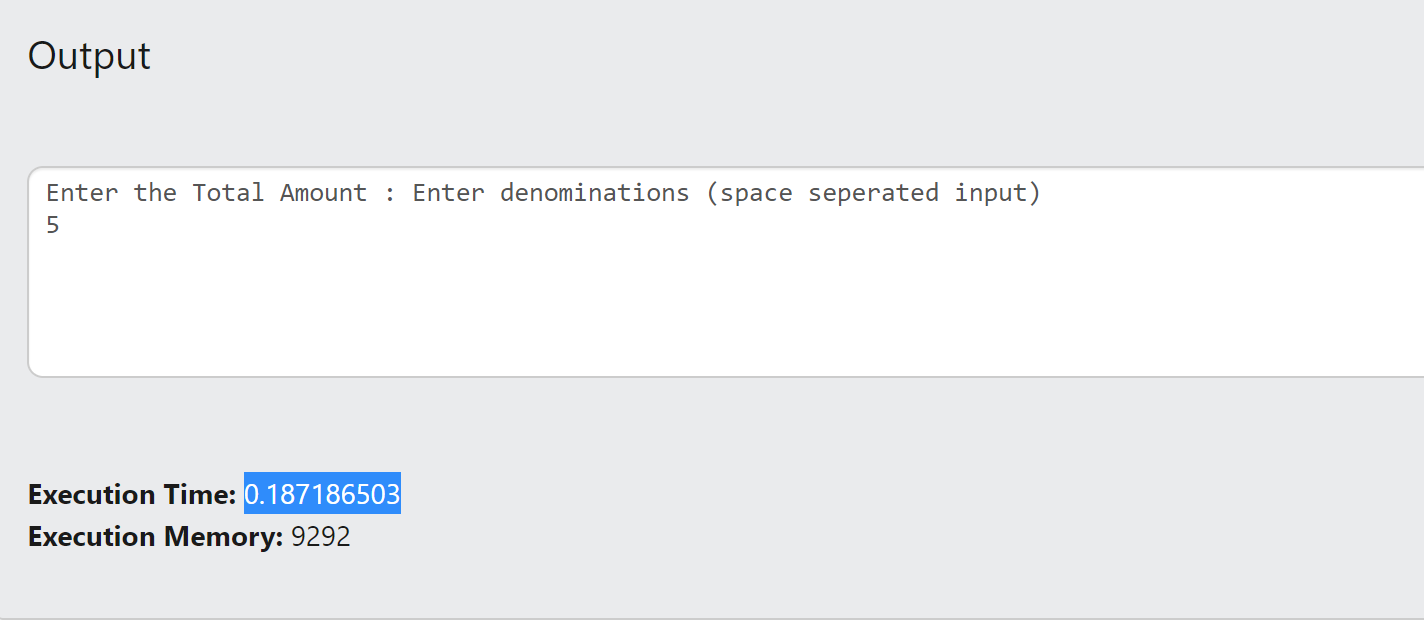
print(result)

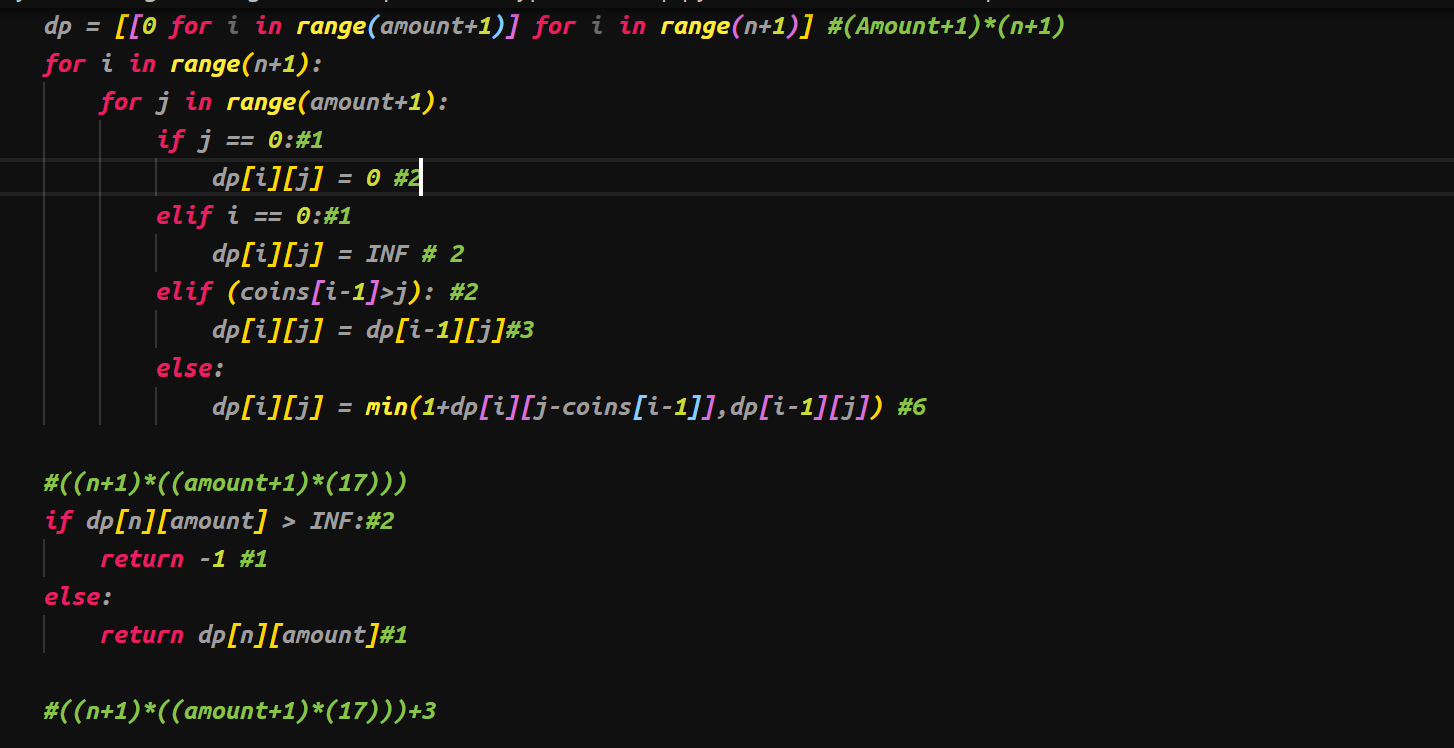
main()



* Time Complexity

O(amount\*Number of coins) : The subproblem here is to find the min number of coins for a small amount.Since we use a Dp table which has Amounts as the column and No. of coins as a row we find out the min coin for each and every subproblem and so the Time Complexity is O(amount\*Number of coins)

* Time Taken in ms = 0.1871s
* Number of steps required to get the output



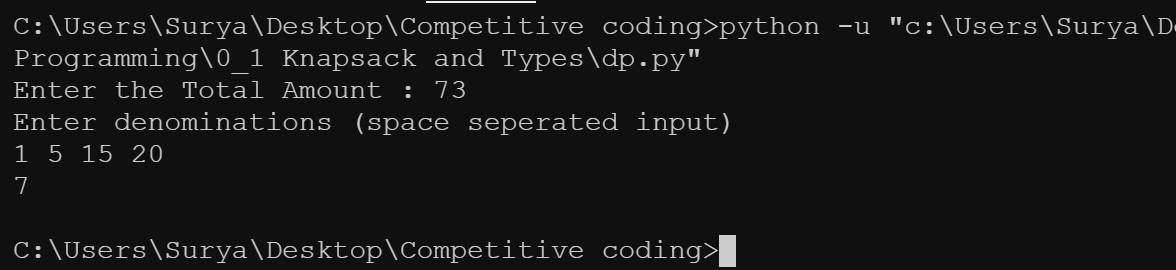
The number of steps is **((n+1)\*((amount+1)\*(17)))**

* Correctness of solution

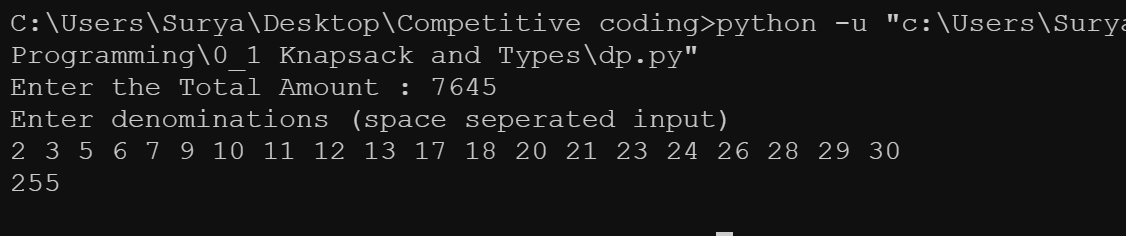
This algorithm works for all types of input

Example:

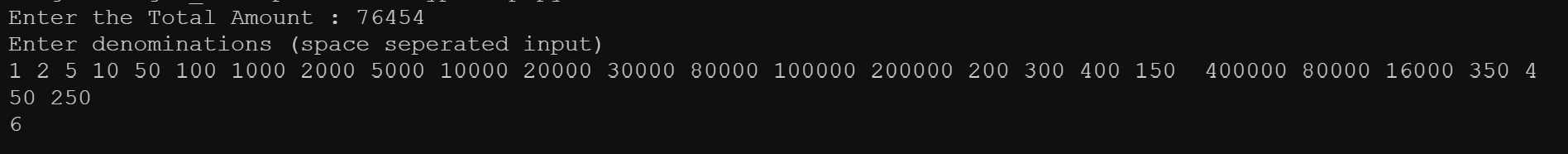
1. The denomination size is 4



1. Denomination size of 20



1. Denominations of size 25



Backtracking

* Code

from sys import maxsize as INF

import sys

sys.setrecursionlimit(10\*\*6)

def backtrack(i,coins,amount):

if amount == 0:

return 0

elif amount < 0 or i < 0:

return INF

else:

return min(1+backtrack(i,coins,amount-coins[i]),backtrack(i-1,coins,amount))

def main():

amount = int(input("Enter the Total Amount : "))

print('Enter denominations (space separated input)')

coins = list(map(int,input().split()))

result = backtrack(len(coins)-1,coins,amount)

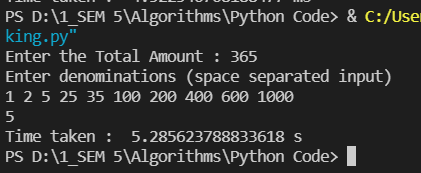
print(result)

main()

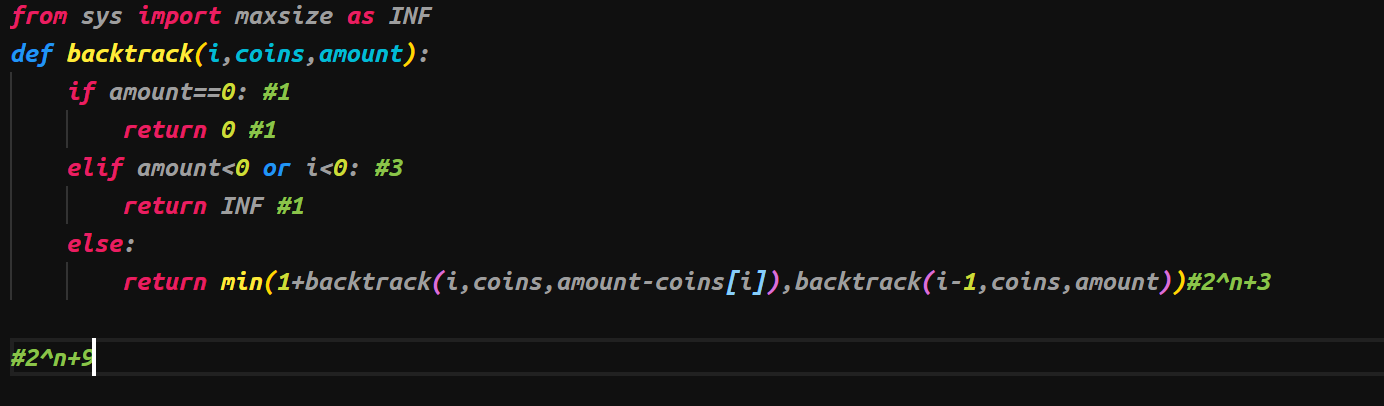
* Time Complexity

O(2^n) : The time complexity of the approach is similar to that of the basic recursion approach. Hence the backtracking approach has an exponential time complexity.

* Time Taken = 5.28 s



* Number of steps required to get the output



Total Number of steps is 2^n+9

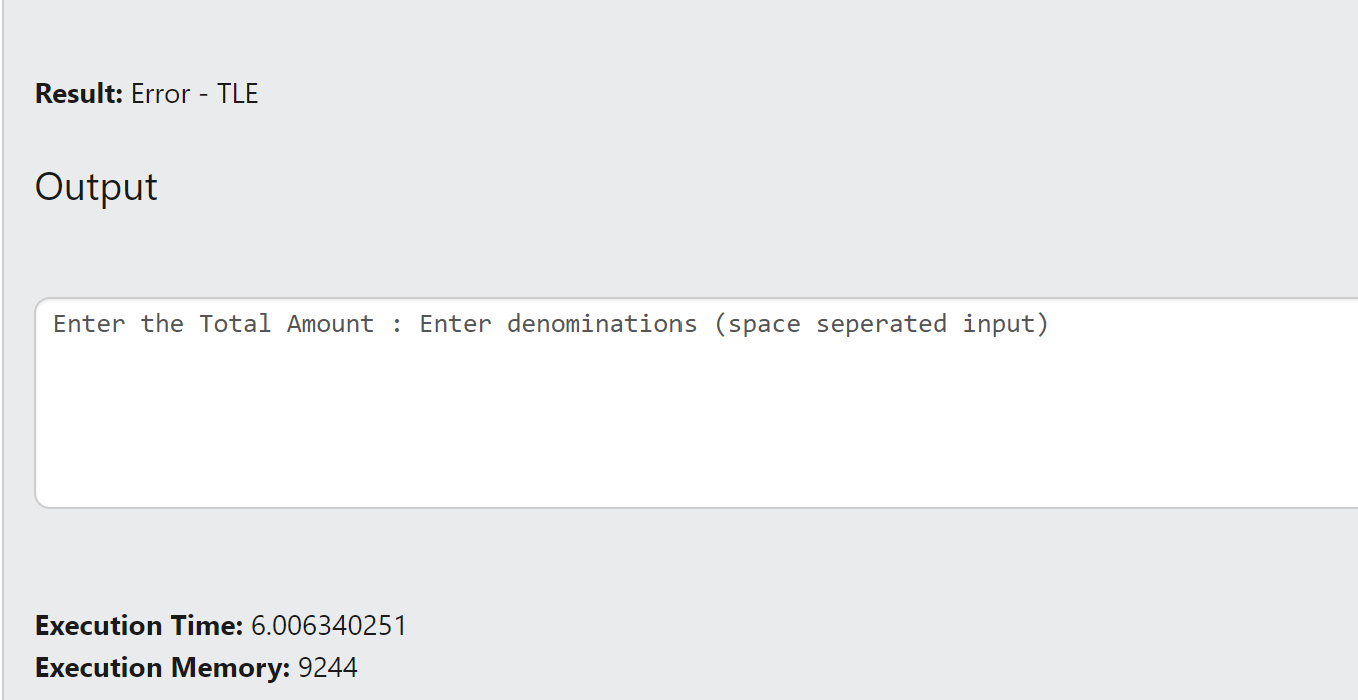
* Correctness of solution:

This algorithm exceeds time limit for large inputs but works for smaller inputs

**TLE example:**

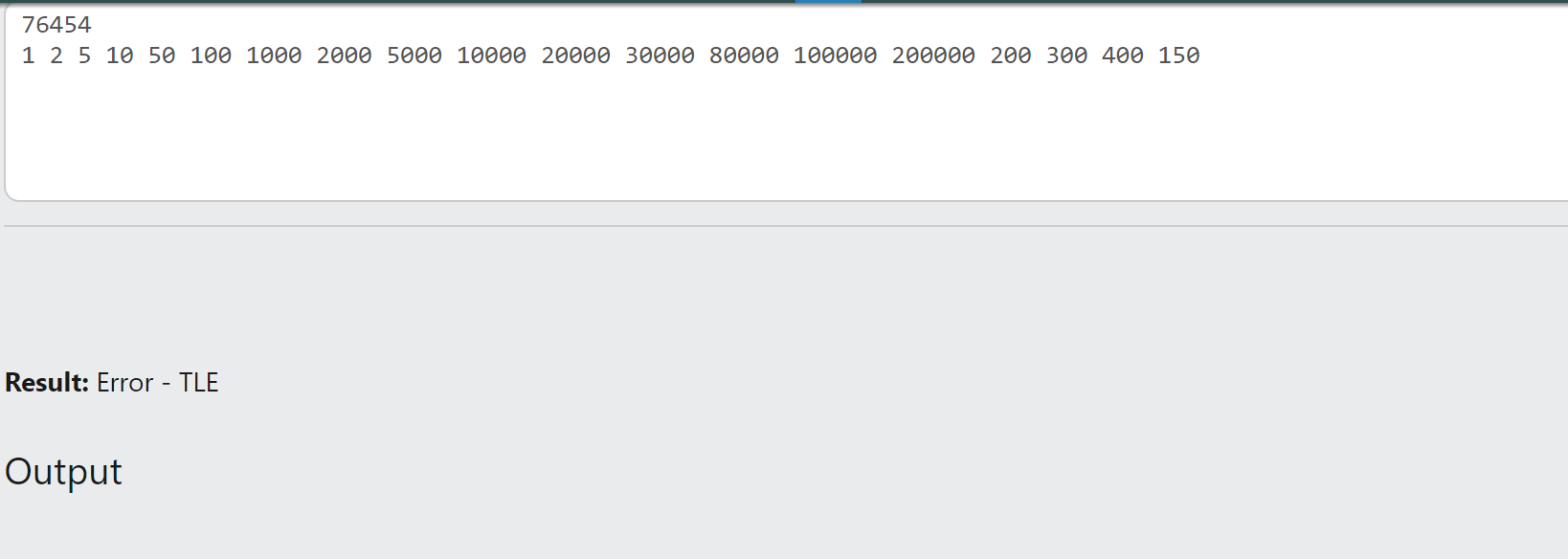
1. 7645

2 3 5 6 7 9 10 11 12 13 17 18 20 21 23 24 26 28 29 30



1. 76454

1 2 5 10 50 100 1000 2000 5000 10000 20000 30000 80000 100000 200000 200 300 400 150



Branch and Bound

* Code
* Time Complexity
* Time Taken in ms
* Number of steps required to get the output
* Correctness of solution