**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**A lab Report 4**

**On**

**“Algorithm and Complexity”**

**[Course Code: COMP 314]**

**Submitted By:**

Sabil Shrestha (51)

**Submitted To:**

**Rajani Chulyadyo Mam**

**Submission Date: 22th November, 2020**

# **Knapsack Problem**

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. Knapsack Problem algorithm is a very helpful problem in combinatorics. It has several practical applications such as filling a lorry truck, fitting a cargo container, shopping bag in a supermarket and many logistical applications. The goal of the knapsack problem is to maximize the value we get from the corresponding weight given in the capacity of knapsack.

Knapsack algorithm can be divided into two types:

* **0/1 Knapsack**

The 0/1 Knapsack problem using dynamic programming. In this Knapsack algorithm type, each package can be taken or not taken. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by Dynamic Programming Approach.

* **Fractional Knapsack**

Fractional Knapsack problem algorithm. Here we even take the fraction of the least valuable weight to maximize the profit or value we get. This method usually produces higher value than 0/1 method because whole capacity of knapsack or bag is utilized in this method. Fractional knapsack problem is mainly solved by using Greedy Strategy, but can be solved using brute force algorithm also.



**Pseudocodes**

1. **0/1 knapsack using bruteforce**

Knapsack\_bruteforce\_zero(wt, pro, cap)

1. n = len(p)

2. bit = get\_combinations(n)

3. max\_pro = 0

4. for b in bit:

5. total\_pro = 0

6. for i in range(n):

7. total\_pro += int(b[i])\*pro[i]

8. total\_wei = 0

9. for i in range(n):

10. total\_wei += int(b[i])\*wt[i]

11. if total\_wei <= cap and total\_pro > max\_pro:

12. max\_pro = total\_pro

13. return max\_pro

def get\_combinations(n)

1. return [bin(x)[2:].rjust(n,'0') for x in range(2\*\*n)]

1. **Fractional Knapsack using brute force**

Knapsack\_bruteforce\_zero(wt, pro, cap)

1. n = len(p)

2. bit = get\_combinations(n)

3. max\_pro = 0

4. for b in bit:

5. total\_pro = 0

6. for i in range(n):

7. total\_pro += int(b[i])\*pro[i]

8. total\_wei = 0

9. for j in range(n):

10. total\_wts += weight[j] \* int(bit[j])

11. total\_prof += profit[j] \* int(bit[j])

12.

13. if total\_wts >= capacity:

14. previous\_wt = total\_wts - weight[j] \* int(bit[j])

15. total\_wts = previous\_wt

16. previous\_pro = total\_prof - profit[j] \* int(bit[j])

17. total\_prof = previous\_pro

18. remaining\_wt = capacity - total\_wts

19. new\_prof = previous\_pro + remaining\_wt \* (profit[i] / weight[i])

20. total\_prof = new\_prof

21. break

22. if total\_pro > max\_pro:

23. max\_pro = total\_pro

24. return max\_pro

get\_combinations(n)

1. return [bin(x)[2:].rjust(n,'0') for x in range(2\*\*n)]
2. **Fractional Knapsack using greedy**

Fractional\_Knapsack\_greedy(Array Weight, Array Value, int M)

1. for i <- 1 to size (Value)

2. calculate cost[i] <- Value[i] / Weight[i]

3. Sort-Descending (cost)

4. i ← 1

5. while (i <= size(Value))

6. if Weight[i] <= M

7. M ← M – W[i]

8. total ← total + V[i];

9. if Weight[i] > M

10. i ← i+1

1. **0 / 1 Knapsack problem using dynamic programming**

0/1 knapsack\_dynamic(n, Weight)

1. for w = 0, Weight

2. do V [0,w] ← 0

3. for i=0, n

4. do V [i, 0] ← 0

5. for w = 0, Weight

6. do if (wi≤ w & vi + V [i-1, w - wi]> V [i -1,Weight])

7. then V [i, Weight] ← vi + V [i - 1, w - wi]

8. else V [i, Weight] ← V [i - 1, w]

**Source Code**

**(brute\_force.py)**

#this gives range of strings 0000-1111

def get\_all\_combinations(n):

    return [bin(x)[2:].rjust(n,'0') for x in range(2\*\*n)]

def knapsack\_bruteforce(p, wt, capacity):

    assert len(p)  == len(wt),"p and wt must be same"

    n = len(p)

    bit\_combinations = get\_all\_combinations(n)#0000 to 1111 all

    max\_profit = 0

        for b in bit\_combinations:  #0000 to 1111

        #calculate all the profits

        total\_profit = 0

        for i in range(n):

            total\_profit += int(b[i]) \* p[i] #eg for 1010 (1\*6+0\*2+1\*7+0\*3)

        total\_weight = 0

        for i in range(n):

            total\_weight += int(b[i]) \* wt[i]

        if total\_weight <= capacity and total\_profit > max\_profit:

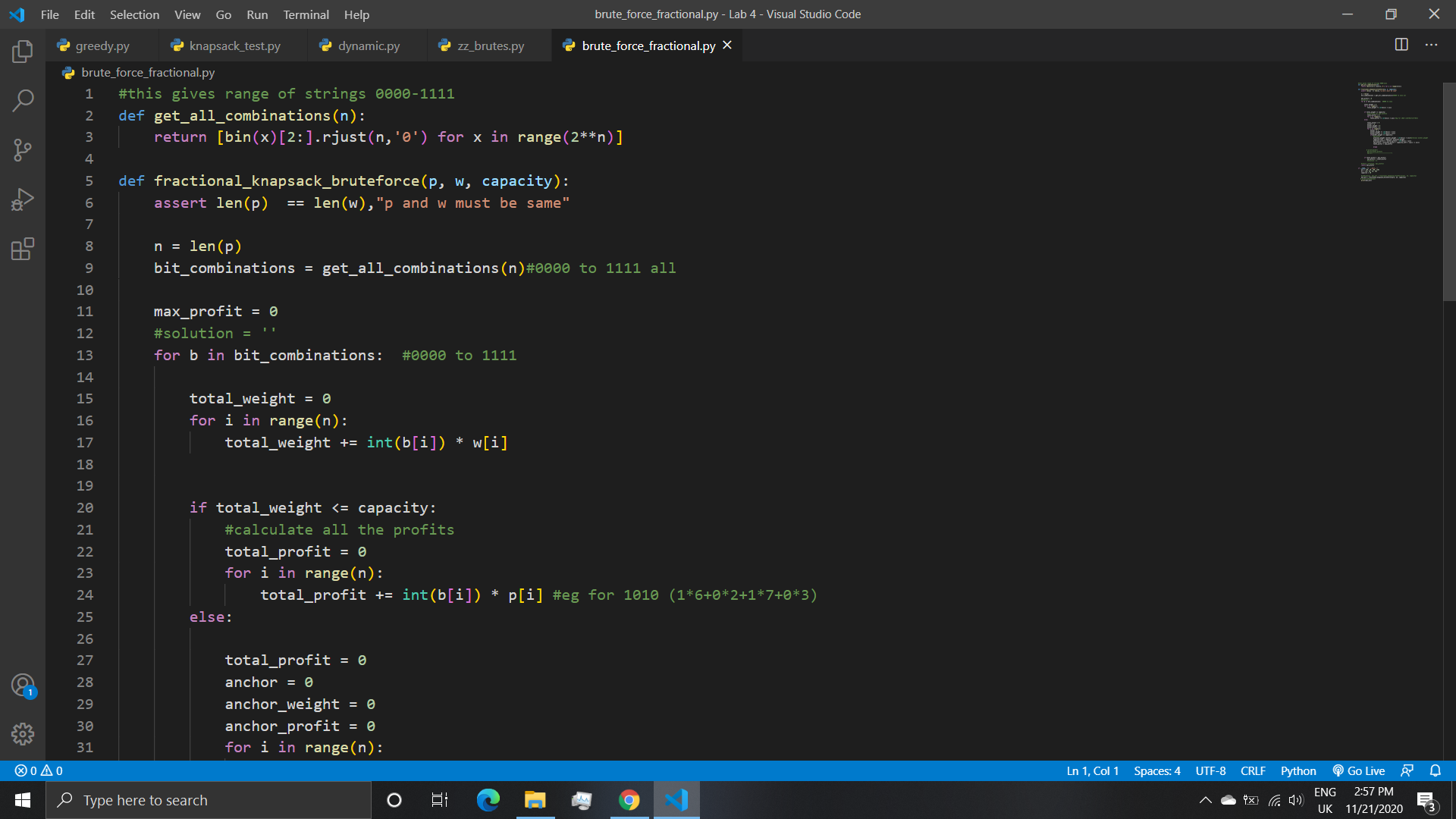
            max\_profit = total\_profit

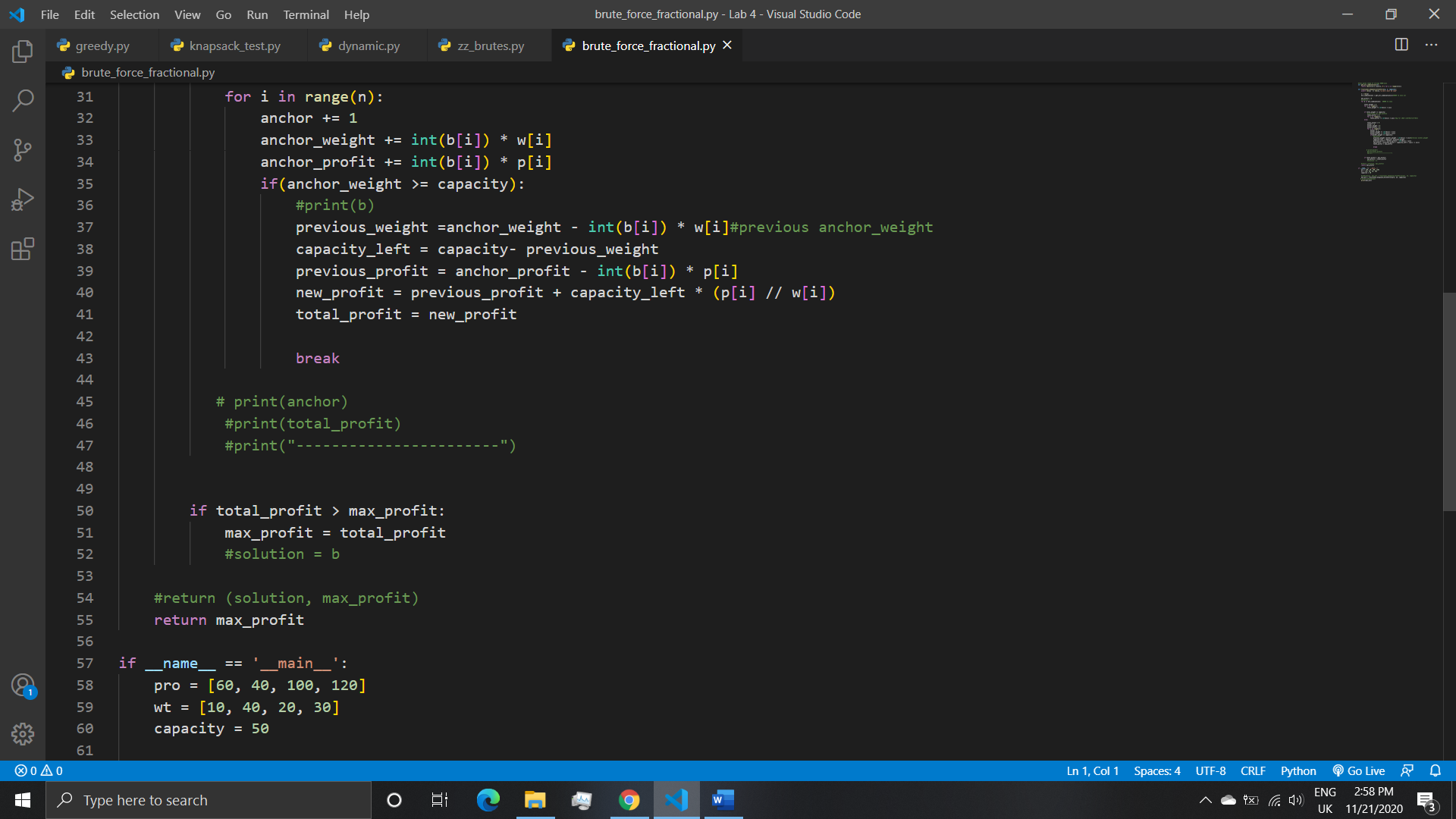
           # solution = b

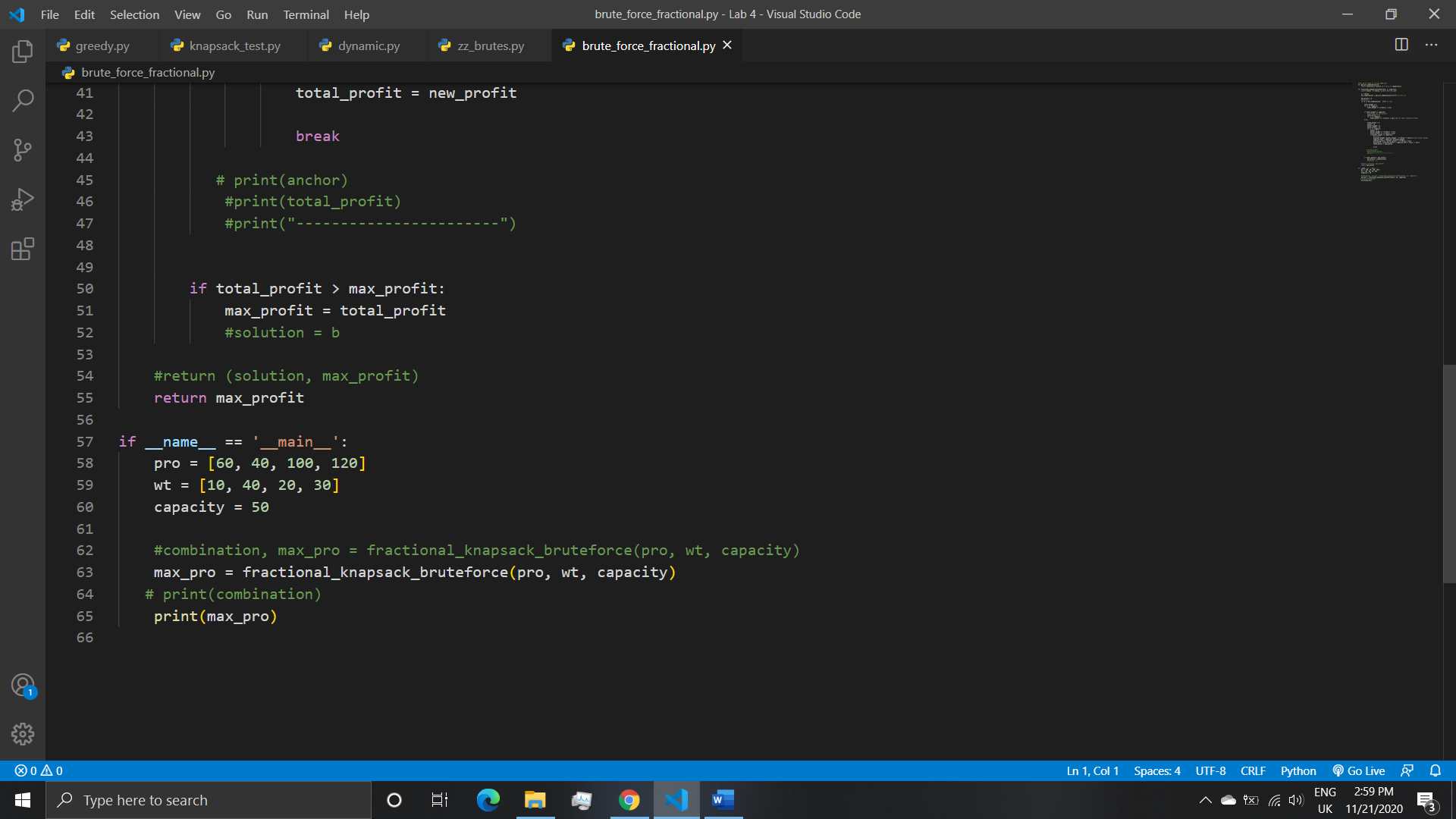
    #return (solution, max\_profit)

    return max\_profit

**(brute\_force\_fractional.py)**







**(greedy.py)**

#this is to make ratio of weight and value for fractional

class Ratio:

    def \_\_init\_\_(self, weight, value, cap):

        self.weight = weight

        self.value = value

        self.cap = cap

        #ratio

        self.cost = value // weight

    def \_\_lt\_\_(self, other):

        return self.cost < other.cost

class fractionalknapsack:

    @staticmethod

    def maximumValue(weight, value, capacity):

        ratios = []

        for i in range(len(weight)):

            ratios.append(Ratio(weight[i], value[i], i))

        # sorting ratios in descending order

        ratios.sort(reverse=True)

        totalValue = 0

        for i in ratios:

            current\_weight = int(i.weight)

            current\_value = int(i.value)

            #if capacity left then add the total value of that one

            if capacity - current\_weight >= 0:

                capacity -= current\_weight

                totalValue += current\_value

            else:

                #if the weight is full and only fractional is allowed then, now fracture the weight

                fraction = capacity / current\_weight

                totalValue += current\_value \* fraction

                capacity = int(capacity - (current\_weight \* fraction))

                break

        return totalValue

# testing

if \_\_name\_\_ == "\_\_main\_\_":

    weight = [10, 40, 20, 30]

    value = [60, 40, 100, 120]

    capacity = 50

    print(fractionalknapsack.maximumValue(weight, value, capacity) )

**(dynamic.py)**

# Dynamic programming for 0/1 knapsack problem

def knapsackDynamic(capacity, weight, value, n):

    #making 2 by 2 matrix of all 0 (capcity \* n)

   table = [[0 for x in range(capacity + 1)] for x in range(n + 1)]

   #Table in bottom up manner

   for i in range(n + 1):

      for c in range(capacity + 1):

          #this is making the side value of the table as 0 (weight 0 and value 0)

         if i == 0 or c == 0:

            table[i][c] = 0

            #if the current weight is smaller than total capcaity

         elif weight[i-1] <= c:

             #max(profit+ total baki profit, or upper cell ko profit)

            table[i][c] = max(value[i-1] + table[i-1][c-weight[i-1]], table[i-1][c])

         else:

            table[i][c] = table[i-1][c]

   return table[n][capacity]

#Main

value = [2,4,3,5,5]

weight = [3,4,1,2,6]

capacity = 12

n = len(value)

print(knapsackDynamic(capacity, weight, value, n))

**Test case(knapsack\_test.py)**

import unittest

from greedy import fractionalknapsack

from dynamic import knapsackDynamic

from brute\_force import knapsack\_bruteforce

from brute\_force\_fractional import fractional\_knapsack\_bruteforce

class KnapsacktestCase(unittest.TestCase):

    def test\_greedy\_fractional(self):

        weight = [10, 40, 20, 30]

        values = [60,40,100,120]

        cap =50

        output =240

        self.assertEqual(fractionalknapsack.maximumValue(weight, values, cap), output)

    def test\_dynamic(self):

        value = [2,4,3,5,5]

        weights = [3,4,1,2,6]

        caps = 12

        n = len(value)

        outs = 15

        self.assertEqual(knapsackDynamic(caps, weights, value, n),outs)

    def test\_brute\_zero\_one(self):

        prof = [5, 6, 7, 2]

        wt = [4, 2, 3, 1]

        caps = 8

        outs = 15

        self.assertEqual(knapsack\_bruteforce(prof, wt, caps),outs)

    def test\_fractional\_brute(self):

        pros = [60, 40, 100, 120]

        wt = [10, 40, 20, 30]

        capss = 50

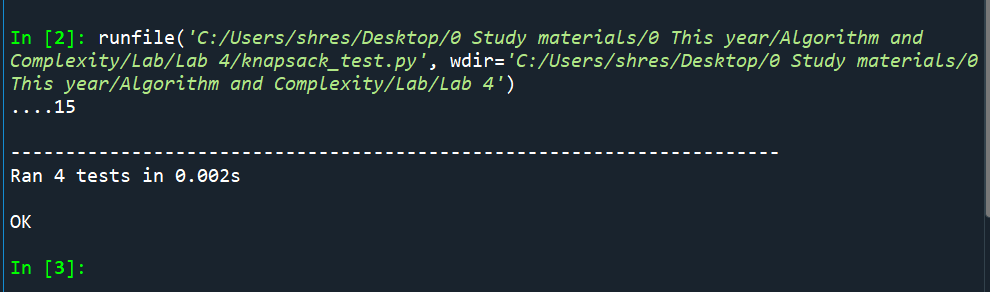
        output = 240

        self.assertEqual(fractional\_knapsack\_bruteforce(pros,wt,capss),output)

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Output**



**Observation**

The knapsack problem was solved using greedy approach, dynamic approach and brute force approach also. The knapsack problem was done using python programming language. First brute force algorithm was used to solve fractional knapsack and 0/1 knapsack. for 0/1 knapsack using brute force, all the possibilities that we can get from the input was calculated using bin. The higher profit from the possibilities was stored as maximum value in a variable. Later the maximum value obtained from the algorithm was returned. This method is slow as calculating all the possibilities is time consuming and has higher time complexity than other algorithm. Second, we did greedy method for solving fractional knapsack. Here firstly, all the profit to weight ratio was calculated. Then the highest ratios were taken multiplied by weight and profit until the capacity was full. The remaining capacity left was multiplied by fraction of the weight of the other ratio weight in order because this is a fractional knapsack problem. For, 0/1 knapsack problem using dynamic programming, table was made which is a matrix of size capacity + 1 and n + 1, then the outer area of matrix was assigned 0. The inner table were filled using the formula. The last value of the table is the solution to the 0/1 knapsack problem from input. The program was then tested with program knapsack\_test.py. All the program passed the test successfully. The test was done using the imported functions from the python files. All the tests were successfully completed using unittest in python.