**Final Project**

* **GitHub Link**

HTTPS:- <https://github.com/rajpatel1904/Project-Advance-Algorithm-.git>

SSH:- [git@github.com:rajpatel1904/Project-Advance-Algorithm-.git](mailto:git@github.com:rajpatel1904/Project-Advance-Algorithm-.git)

* **Description of the problem: -**

You have a block of gold that you want to sell. You are able to cut the block into smaller pieces before making any sales. You are given the following information: a list of

weights, along with the amount of money you can sell a piece of gold of the given weight

for. Assume that if a weight isn’t on your list, you would not get any money for selling a

piece of gold of that weight. How many pieces should you cut your block of gold into to

maximize your profit?

Here the weight of gold in **grams (g)** and the price of gold in **dollars ($)**

* **Inputs and Outputs**

**Input: - A** block of gold, a list of weights, and a list of prices to be made for each weight

**Output: -** The number of pieces the block should be cut into maximum profit.

* **Pseudocode For all Three Algorithms**

1. **Brute Force Algorithm:**

def brute\_force(g, W, P):

max\_price = 0 num\_pieces = 0

// Iterate through all possible combinations of items

for i = 0 to 2^len(W):

weightLeft = g

price = 0

pieces = 0

for j = 0 to len(W):

if 2\*j & I:

if weightLeft - W[j] < 0:

break

weightLeft -= W[j]

price += P[j]

pieces += 1

if price > max\_price and weightLeft == 0:

max\_price = price

num\_pieces = pieces

return num\_pieces

1. Greedy Algorithm:

function greedy(g, W, P)

arr= []

for(i=0; i<W.length; i++){

arr[i]=[P[I], W[I]]

}

arr.sort((a,b) => b[0] - a[0]) //sort in descending order by P

price = 0

weightLeft = g

numPieces = 0

while(weightLeft > arr[arr.length-1][1]){

for(i=0; i<arr.length; i++){

if(W[i] <= weightLeft){

weightLeft -= W[i]

price += P[i]

numPieces += 1

break;

}

}

}

return numPieces;

}

1. **Dynamic Programming:**

def dynamic\_method(g, W, P):

table = [[0 for w in range(g + 1)] for j in range(len(W))]

for j in range(len(W)):

for w in range(g + 1):

if w == 0:

table[j][w] = 0

elif W[j] > w:

table[j][w] = table[j - 1][w]

else:

table[j][w] = max(table[j - 1][w], P[j] + table[j - 1][w - W[j]])

res = table[len(W) - 1][g]

w = g

numPieces = 0

for i in range(len(W) - 1, -1, -1):

if res <= 0:

break

if res == table[i][w]:

continue

else:

numPieces += 1

res = res - P[i]

w = w - W[i]

return numPieces

* **Time Complexity of all three algorithms**

1. **Brute Force Algorithm:**

**The time complexity of the brute force algorithm to solve this problem is O(x\*2^n)**

This is because we need to consider every pair of weights in the list and determine the optimal combination of pieces to cut the block of gold is x to in order to get maximize profit for each weight in the list, we need to consider all the others and calculate the profit for each combination. This means we must consider n weights, and for each weight time complexity is O(x\*2^n)

1. **Time complexity of Greedy algorithm:**

**The time complexity of the greedy algorithm is O(ng/min(list of weights)) is equivalent to O(n/list of weight)**

The greedy algorithm time complexity is O(ng/(list of weights)) because in tunning time do not need to look every weight in the list.

1. **Time complexity of Dynamic Programming:**

**The time complexity of the dynamic programming algorithm to solve this problem is O(nW) because it depends on the list of weights**

The algorithm works by recursively computing the optimal profit for each weight in the list and storing the result in the table. The time complexity of the algorithm is O(nW) a here n is the number of weights in the list. As the algorithm needs to traverse into the entire list of weights, the time complexity is directly proportional to the number of items in the list.

* Explanation of why the greedy algorithm has a different solution

The greedy algorithm would attempt to maximize the profit by selecting the most profitable option available at any given point. This means that it would select the piece of gold with the highest price per unit weight to sell first. As it progresses, it will continue to select the most profitable option while disregarding any remaining gold that is not as profitable. The other algorithms, such as dynamic programming, would focus on the overall profit instead of the profit at any given point. This means that it would look at the entire problem holistically and try to maximize the overall profit, instead of just the profit at any given point. This could lead to different solutions as it may select a sequence of cuts that are not necessarily the most profitable at each step but overall provide a higher profit.

* Explanation of any missing test cases

The algorithm to solve this problem is a greedy approach. We start by taking the highest-priced weight and subtracting it from the total block of gold weight. We then take the next highest-priced weight and subtract it from the remaining weight. We continue this process until all the weights have been used or we have reached a point where there is no more gold left to cut. For example, if the weights and prices are given as follows: for TEST CASE 3 Weight 1: 5g, Price 1: $100 Weight 2: 10 g, Price 2: $600 Weight 3: 15 g, Price 3: $1000 and we have 1 g gold block so We can maximize our profit by cutting our block of gold into 0 pieces of 5 g, 10g, 15 g. This will fetch us a total profit of $0. If the test cases were taking too long, we could use a different algorithm like Dynamic Programming or Branch and Bound to solve the problem more efficiently. Dynamic Programming would store the optimal solution for each subproblem, while Branch and Bound would look for the optimal solution by pruning off certain branches that could not lead to a better solution.

* Algorithm recommendation

I would recommend using a greedy algorithm to solve this problem. A greedy algorithm is an approach to solving problems where the aim is to make the locally optimal choice at each stage to reach a globally optimal solution. In this case, the goal is to maximize the profit from cutting and selling the block of gold. A greedy algorithm would solve this problem by choosing the highest possible price for each piece of gold that is cut from the block, thus maximizing the profit. This approach is simpler than other algorithms, such as dynamic programming, and can be used to quickly find the optimal solution.

**Test Cases**

1. **1st test case:**

**g= 50g**

**Weight= [1,5,10,15]**

**Price= [100,600,1000,1400]**

1. **2nd test case:**

**g= 150g**

**Weight= [10,50,80,100,130]**

**Price= [500,1200,1800,1900,2100]**

1. **3rd test case:**

**g= 500g**

**Weight= [5,18,20,28,40,60,100,145]**

**Price= [50,80,130,180,230,280,340,400]**

1. **1st test case:**

**g= 1000g**

**Weight= [1,5,10,15,20,100,1500]**

**Price= [100,600,1000,1400,5000,10000]**

1. **1st test case:**

**g= 1g**

**Weight= [1,5,10,15]**

**Price= [100,600,1000,1400]**

**OUTPUT**

**Greedy algorithm:**

*def* greedy\_method(*w*, *Weight*, *Price*):

*Weight*, *Price* = zip(\*sorted(zip(*Weight*, *Price*), *reverse*=True, *key*=*lambda* *pair*: *pair*[1]))

    p = 0

    wl = *w*

    num\_pieces = 0

    while wl > min(*Weight*):

        for i in range(len(*Weight*)):

            if *Weight*[i] <= wl:

                break

        wl -= *Weight*[i]

        p += *Price*[i]

        num\_pieces += 1

    return num\_pieces

**output for 1st test case**

**Text

Description automatically generated**

**Time: 0.136sec**

**Output for 2nd test case**

**Text

Description automatically generated**

**Time: 0.152 sec**

**output for 3rd test case**

**Text

Description automatically generated**

**Time:0.165sec**

**Output for 4th test case**

**Text

Description automatically generated**

**Time: 0.212 sec**

**output for 5th test case**

**Text

Description automatically generated**

**Time :0.102 sec**

**Brute Force algorithm:**

**Output for 1st test case**

**Text

Description automatically generated**

**Time: 1.180 sec**

**Output for 2nd test case**

**Text

Description automatically generated**

**Time: 1.215 sec**

**Output for 3rd test case**

**Text

Description automatically generated**

**Time: 1.236 sec**

**Output for 4th test case**

**Text

Description automatically generated**

**Time: 1.265 sec**

**Output for 5th test case**

**Text

Description automatically generated**

**Output: 1.152 sec**

**Dynamic Programming**

**Output for 1st test case**

**Text

Description automatically generated**

**Time:- 0.180 sec**

**Output for 2nd test case**

**Text

Description automatically generated**

**Time: 0.202 sec**

**Output for 3rd test case**

**Text

Description automatically generated**

**Time: 0.308 sec**

**Output for 4th test cases**

**Text

Description automatically generated**

**Time: 0.310 sec**

**Output for 5th test cases**

**Text

Description automatically generated**

**Time: 0.520 sec**