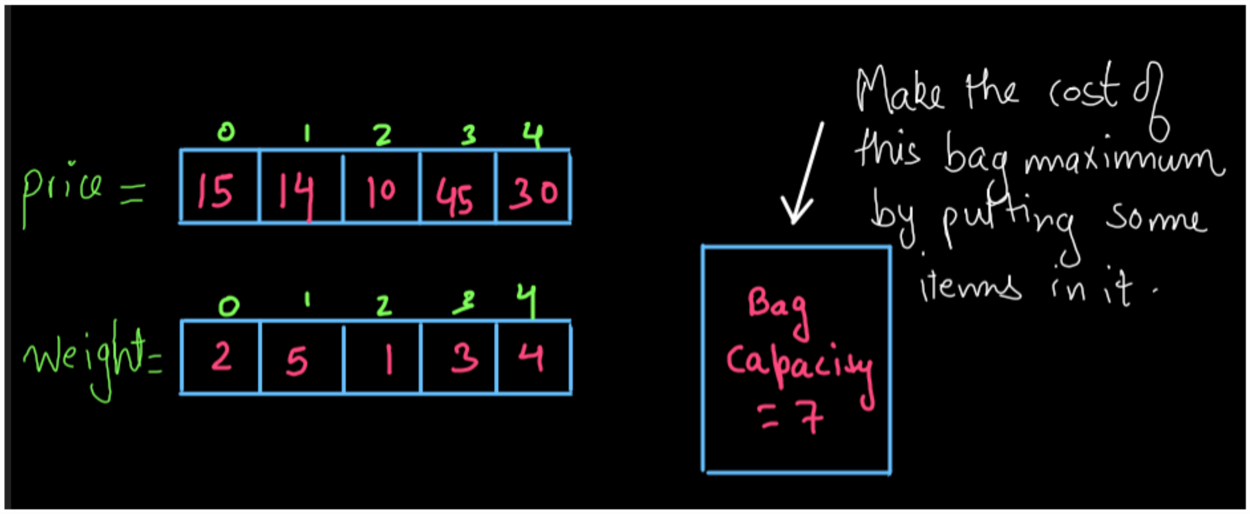
Dynamic Programming

Content

Unbounded Knapsack



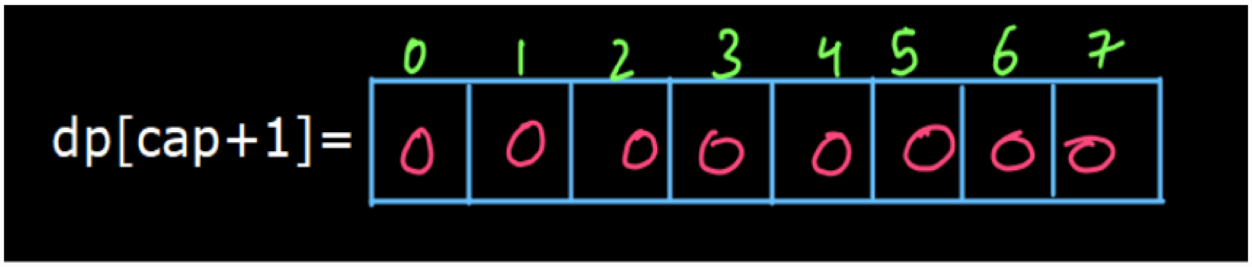
<u>Problem</u>

We are given two arrays representing the weights and costs of each item. We have a bag of a limited capacity and we have to fill the bag in such a way that its cost becomes maximum. This was also the case with the 0-1 KNAPSACK problem but we have a change here. We can still not include any element partially, but we can include an element more than once i.e. repetition is allowed.

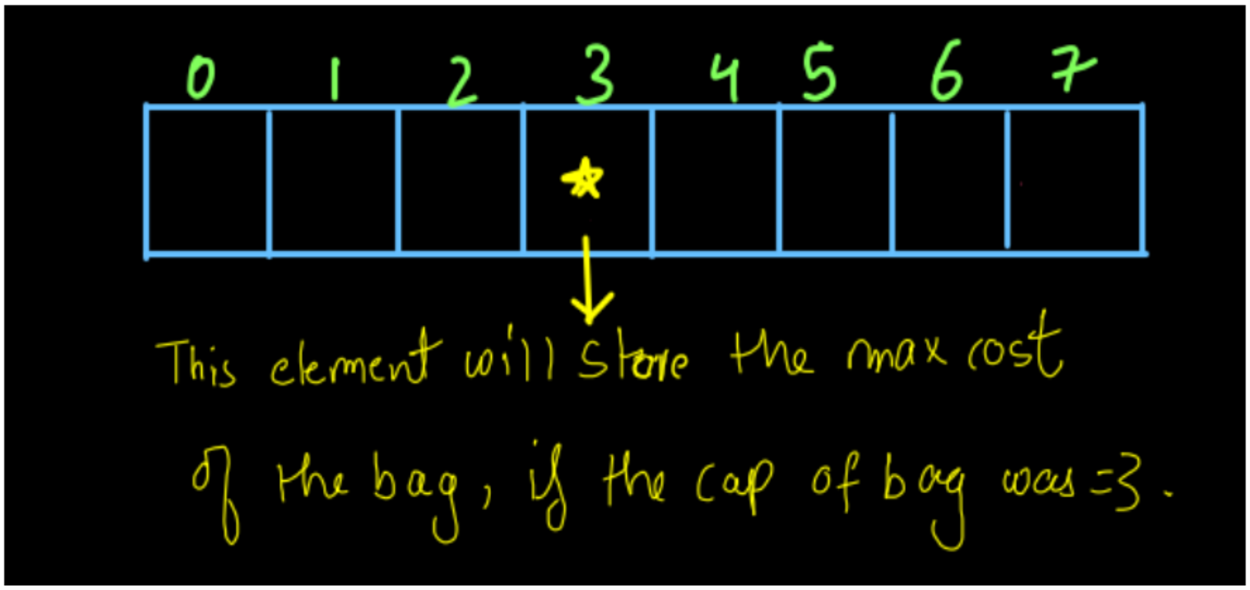
Now, we recommend you meditate on the fact that repetition is allowed and think of some situation like this that you have already encountered. So, did you get any ideas? If you remember, we solved the TARGET SUM SUBSET and the COIN CHANGE PROBLEMS. There was a difference between these two problems. In the TARGET SUM SUBSET problem, repetition was not allowed but in the COIN CHANGE problems (both permutation and combination), repetition was allowed. (Meditate on this fact and analyze how these problems are different from each other).

Approach :

1. Storage and memory: Have a look at the array given below:

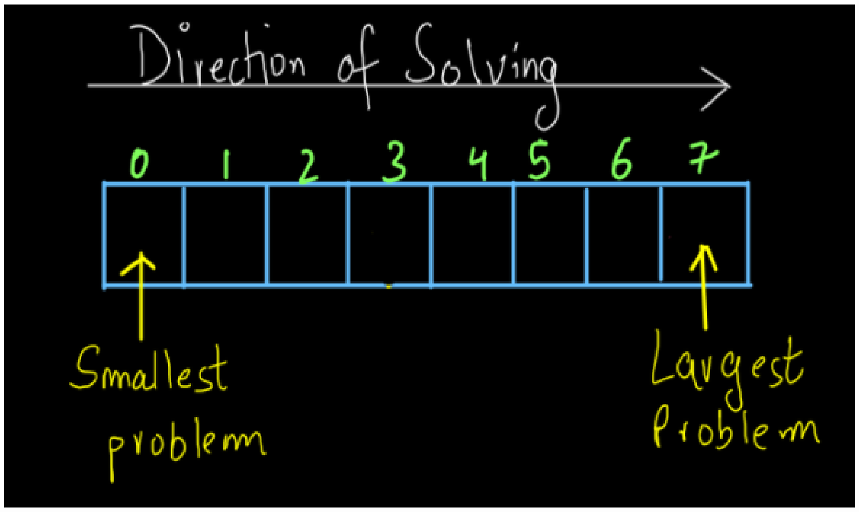


2. We have made an array dp of size=cap+1, where the cap is the capacity of the bag given to us. So, we have assigned the storage. Now, the question is, what does each element in the array represent?

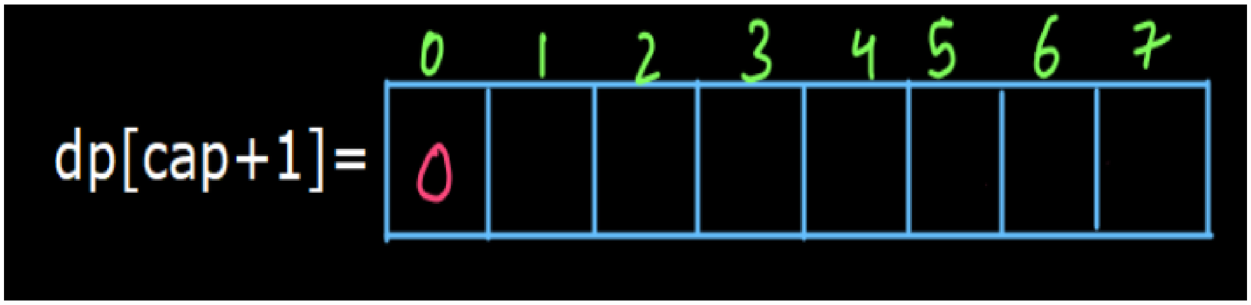

0
1
2
3
4
5
6
7
This element will store the max cost of the bag, if the cap of bag was =3.

3. As shown in the figure, the element at i=3 will store the maximum cost of the bag that can be obtained by filling the same set of items given to us, if the capacity of bag=3. So, the element dp[i] depicts the max cost of the bag that can be obtained by filling the bag with the same set of elements given to us, if the capacity of bag cap=i. You may watch the solution video to understand the storage and meaning stage of dynamic programming if you have any doubts regarding this.
4. Dear reader, we want you to meditate at this point and think, why did we choose a one-dimensional array for storage and not a 2-D array? Why did we do the same for the coin change problem and why do we take 2-D arrays in the 0-1 knapsack problem as well as the target sum subset problem?
5. <u>Find Solution Direction:</u> We have now assigned the storage and we also know the meaning of our assigned storage. Now is the time to think about the direction of the solution. So, what is the smallest and the largest problem according to the storage that we have assigned?
   The smallest problem is when we are at index 0. Since dp[i] represents the max cost of the bag when the capacity of our bag=i, so, when the capacity of the bag is 0, then the max cost is also 0. This is the smallest problem and the largest is when the capacity of the bag reaches the capacity given in the question. So, we always

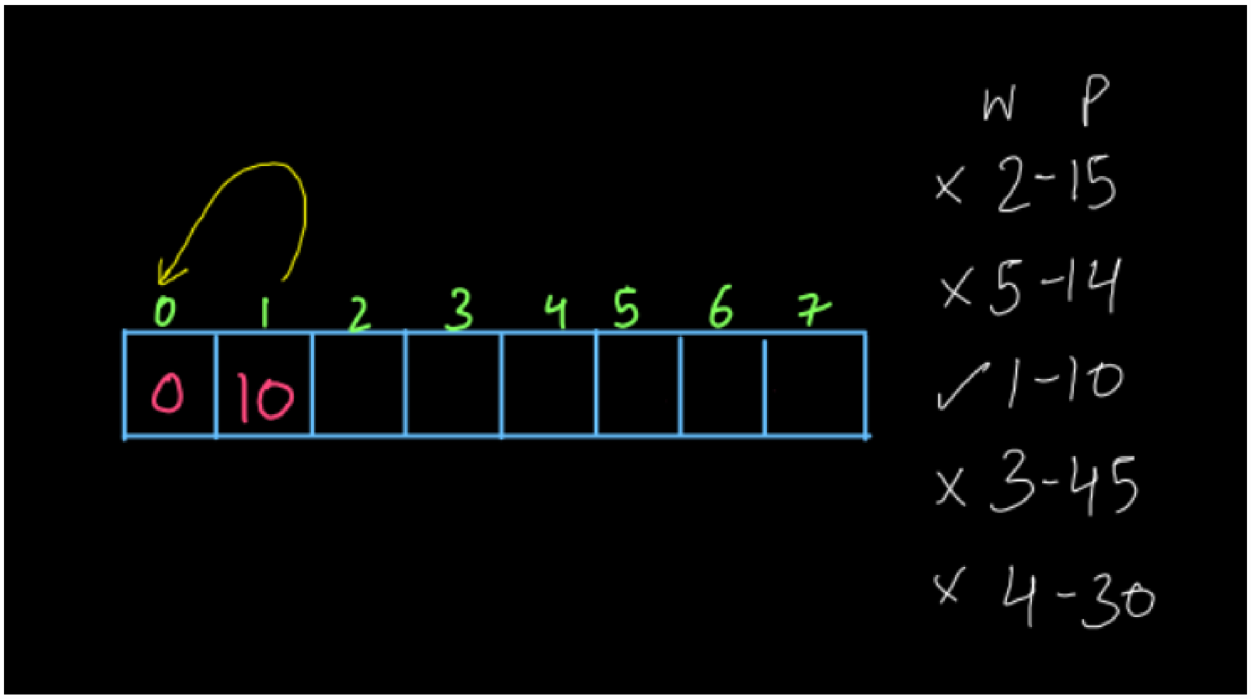
solve in the direction starting from the smallest problem to the biggest problem. Therefore, we will solve in the direction from i=0 to i=cap. Have a look at the image given below: (fig-4)



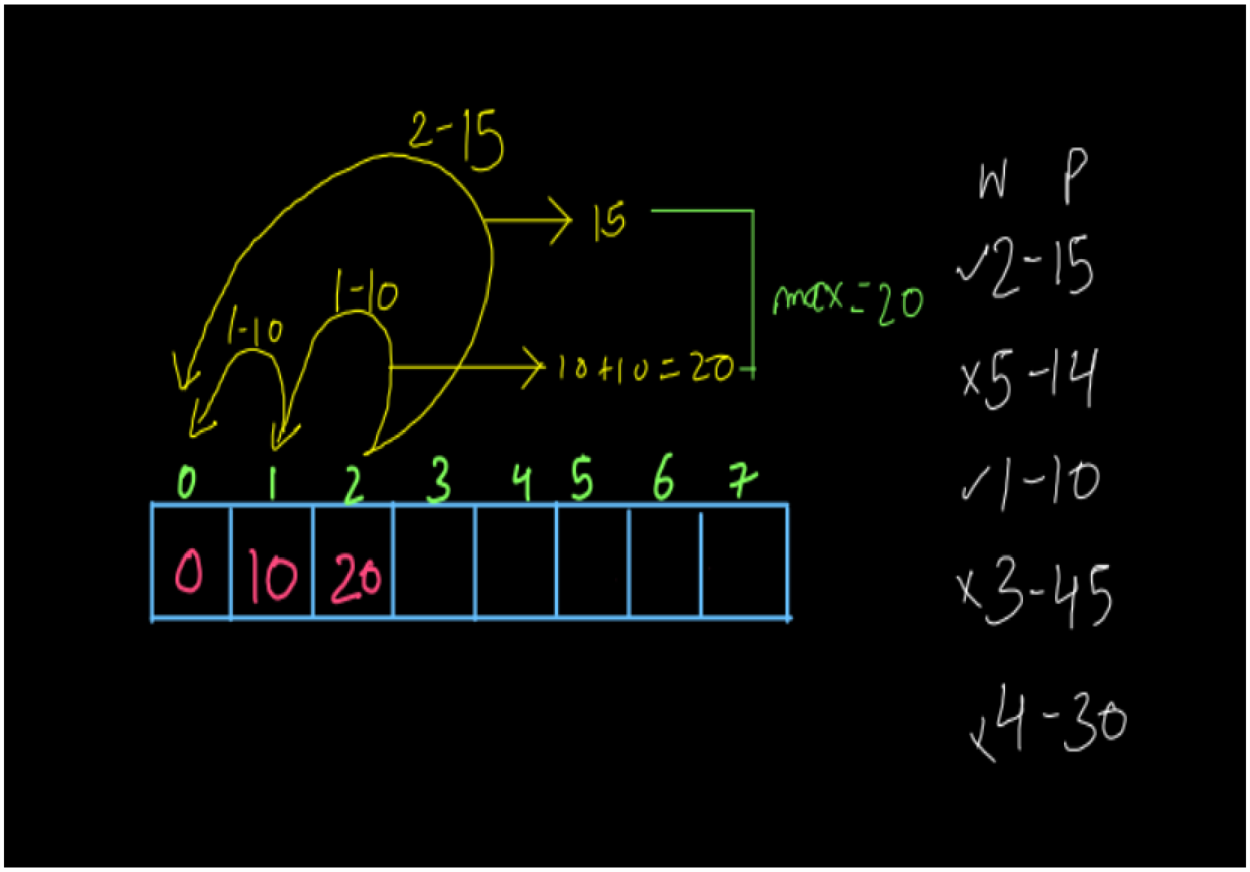
6. Traverse and Solve: Dear reader, now we recommend you to study each step from here carefully. You are requested to watch the solution video to understand how we are traversing the array and solving the problem as it is difficult to explain every step in writing. Still, let's try to understand the first few steps:
7. First of all, we know that when the capacity of our bag is zero, no item can fit in it. Therefore the dp[0] will be 0.



8. Now, we move to i=1. This means that the capacity of our bag is currently 1. So, we will find an item whose weight is equal to 1 so that it can fit into the bag. Look at the diagram given below:

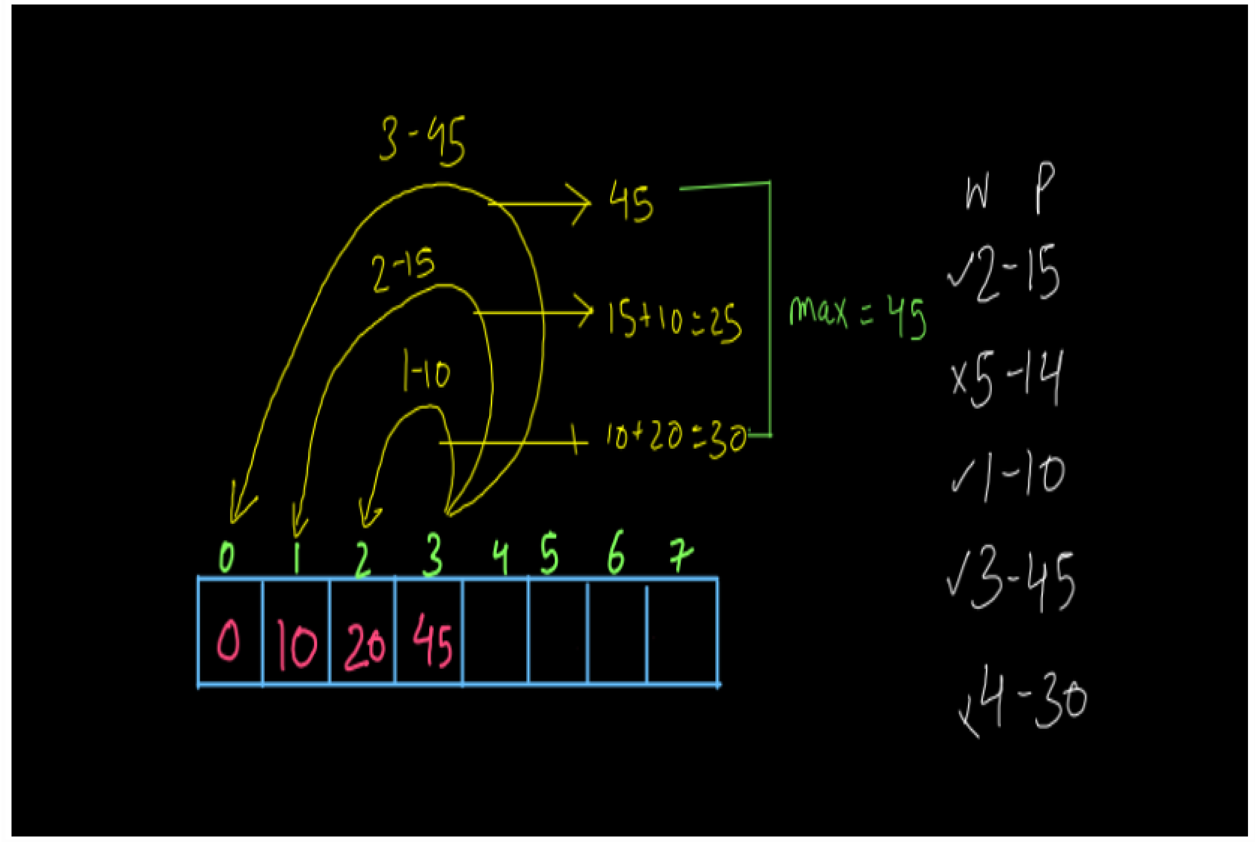


9. The wights and the prices of items are shown in the figure. Since the current capacity of our bag is 1, we can not fit any item whose weight is greater than 1 in our bag. We have an item whose weight is 1. So, if we put that item into the bag, the remaining capacity is now 0. And the max cost when capacity was 0 is 0. So, the max cost of the bag when capacity is 1 is the max cost at cap=1 + max cost at capacity=0 which equals 0+10=10. Therefore, at dp[1] we put 10.
10. Now we move to i=2. Now, we can add all those items whose weight is less than or equal to 2. Look at the diagram given below:



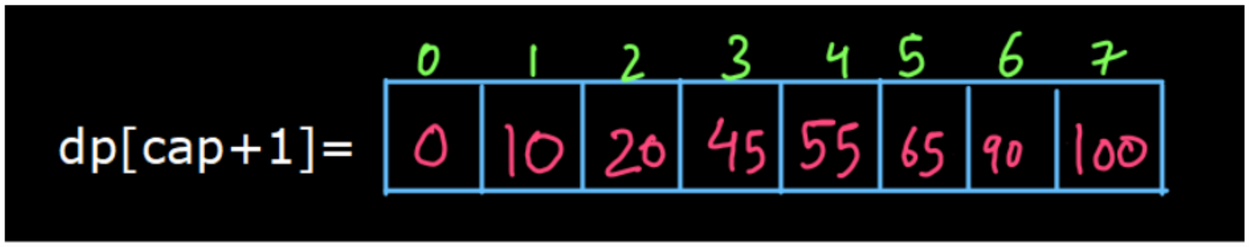
11. We have 2 options i.e. item 0 which has a weight 2 and item 2 which weighs 1. So, we can either put only 1 item of weight 2 or we can put 2 items of weight 1. If we put only 1 item of weight 2, then the remaining capacity of the bag is 0. The max cost of the bag at cap=0 is 0. So, the cost of the bag when we put only a single item of weight=2 is the cost of the item of weight 2 i.e. 15. If we put an item of weight=1 in the bag the remaining capacity of the bag is 1. The maximum cost of the bag when the capacity is 1 is present at dp[1] i.e=10. So, the cost of the bag when we put 2 items of weight 1 is the cost of the item of weight=1 + cost of the same item=10+10=20. So, the cost of the bag is 20 when we put 2 items of weight 1 and it is 15 when we put only 1 item of weight=2. So, the max cost is 20 (when we put 2 items of weight=1). So at dp[2] we will store 20 i.e. the max cost of the bag when capacity=2 is 20.

12. Now we have reached i=3. Now, we can add all those items whose weight is less than or equal to 3. Look at the diagram given below:



13. We now have 3 options. We can either add 1 weight of 3 units which costs 45. The other option is to add one weight of 2 units and the other of 1 unit. So, the max cost for this case is (the max cost of the item of weight=2)+(the max cost of the item of weight=1) i.e. 15=10=25. Similarly, the other option is to add 3 items of weight=1. The max cost for this will be 10+10+10=30. Now, if we compare these three cases, the maximum cost is when we add 1 weight of 3 units. So at dp[3] we will add 45.

Similarly, we can fill the entire array and our dp array will look like this:



Again, we request you to watch the solution video if you haven't as it is very difficult to explain this procedure in writing. Moreover, watching the video will also clear some of your extra doubts.

Now that we know the entire procedure let us try to analyze it in programming terms:

1. Make an array dp[cap+1].
2. Now, we will iterate through the dp array and the items and we will fill a value in dp array's ith position only if the weight of the (i-1)th item is less than the value of i which is the current capacity of the bag.
3. So, the remaining bag capacity will be rbagc=cap-weights[i-1]. Also the remaining bag value i.e. the max cost of the bag after inserting element of weight =-weight[i-1] is rbagv=dp[rbagc].
4. The total max cost of inserting the item with weight= weight[i-1] will be rbagv+price[i-1].

<u>Pseudo Code:</u>

Storage and Meaning: We will assign storage and also assign meaning for that storage. Here, we take an array dp[cap+1]. This is a one-dimensional array and dp[i] i.e. ith element in this array represents the maximum cost of the bag if the capacity of the bag was 'i'.

Find Solution Direction: As we know what dp[i] means, we know that dp[0] is the max cost of the bag when the capacity of the bag is 0. So, this is the smallest problem as we already know the solution to it is that if the capacity is 0 the maximum cost of the bag will be 0. The largest problem is dp[cap] when the capacity of the bag is equal to our given cap[acity. So, the direction of solving is from 0 to cap.

Traverse and Solve: Traverse the array by comparing the value of current capacity to the weights of the elements and try to fill the bag with any elements if possible. Then, try to compare all the combinations and fill the bag with a combination of maximum capacity.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/unbounded_knapsack.cpp