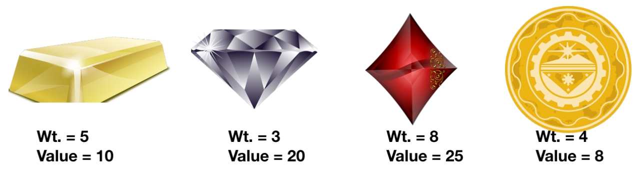
Knapsack Problem | Dynamic Programming

Suppose you woke up on some mysterious island and there are different precious items on it. Each item has a different value and weight. You are also provided with a bag to take some of the items along with you but your bag has a limitation of the maximum weight you can put in it. So, you need to choose items to put in your bag such that the overall value of items in your bag maximizes.



This problem is commonly known as the knapsack or the rucksack problem. There are different kinds of items (i) and each item i has a weight (w_i) and value (v_i) associated with it. x_i is the number of i kind of items we have picked. And the bag has a limitation of maximum weight (W).

i	1	2	3	4	5
Wi	12	32	33	5	34
Vi	100	200	50	60	150

Maximum wt. = 50

So, our main task is to maximize the value i.e., $\sum_{i=1}^{n}(v_i x_i)$ (summation of the *number of items taken * its value*) such that $\sum_{i=1}^{n} w_i x_i \leq W$ i.e., the weight of all the items should be less than the maximum weight.

There are different kind of knapsack problems:

1. **0-1 Knapsack Problem** → In this type of knapsack problem, there is only one item of each kind (or we can pick only one). So, we are available with only two options for each item, either pick it (1) or leave it (0) i.e., $x_i \in \{0, 1\}$.
2. **Bounded Knapsack Problem** (BKP) → In this case, the quantity of each item can exceed 1 but can't be infinitely present i.e., there is an upper bound on it. So, $0 \leq x_i \leq c$, where c is the maximum quantity of i we can take.
3. **Unbounded Knapsack Problem** (UKP) → Here, there is no limitation on the quantity of a specific item we can take i.e., $x_i \geq 0$.
4. **Integer Knapsack Problem** → When we are not available to just pick a part of an item i.e., we either take the entire item or not and can't just break the item and take some fraction of it, then it is called integer knapsack problem.
5. **Fractional Knapsack Problem** → Here, we can take even a fraction of any item. For example, take an example of powdered gold, we can take a fraction of it according to our need.

Some kind of knapsack problems are quite easy to solve while some are not. For example, in the fractional knapsack problem, we can take the item with the maximum $\frac{value}{weight}$ ratio as much as we can and then the next item with second most $\frac{value}{weight}$ ratio and so on until the maximum weight limit is reached.

In this particular chapter, we are going to study the 0-1 knapsack problem. So, let's get into it.

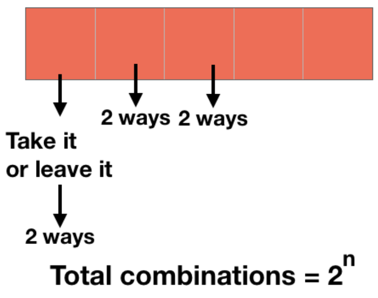
0-1 Knapsack Problem

Let's start by taking an example. Suppose we are provided with the following items:

i	1	2	3	4
Wi	3	2	4	1
Vi	8	3	9	6

and the maximum weight the bag can hold is 5 units i.e., $W = 5$. Since it is a 0-1 knapsack problem, it means that we can pick a maximum of 1 item for each kind. Also, the problem is not a fractional knapsack problem but an integer one i.e., we can't break the items and we have to pick the entire item or leave it.

First take a case of solving the problem using brute force i.e., checking each possibility. As mentioned above, we have two options for each item i.e., either we can take it or leave it.



So, there are two ways for each item and a total of n items and hence, the total possible combinations are

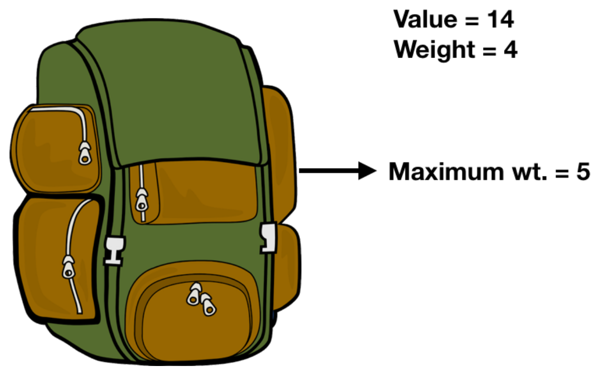
$$\underbrace{2*2*2*\ldots*2}_{n\text{ times}} = 2^n \text{ combinations}$$

So, you can see that this will take $O(2^n)$ time and we already know how bad an exponential running time is and thus there is no chance that we are going to use this as our solution.

One can also think of a solution of always taking the item with the highest $\frac{value}{weight}$ ratio first (known as greedy algorithm) but it is also not going to help here. As mentioned above, it could have helped in the case of the fractional knapsack problem. Let's first test this idea to see that it really doesn't work here.

The item with the highest $\frac{value}{weight}$ ratio is the 4^{th} one, so we will pick it and then the 1^{st} one. Now, we have a total value of $8 + 6 = 14$ and a total weight of $3 + 1 = 4$ units. We still have the space left to take more items having a total maximum weight of 1 unit but there is no such item left (each item left has a weight greater than 1), so this is what we can pick by using this technique.

i	1	4
Wi	3	1
Vi	8	6

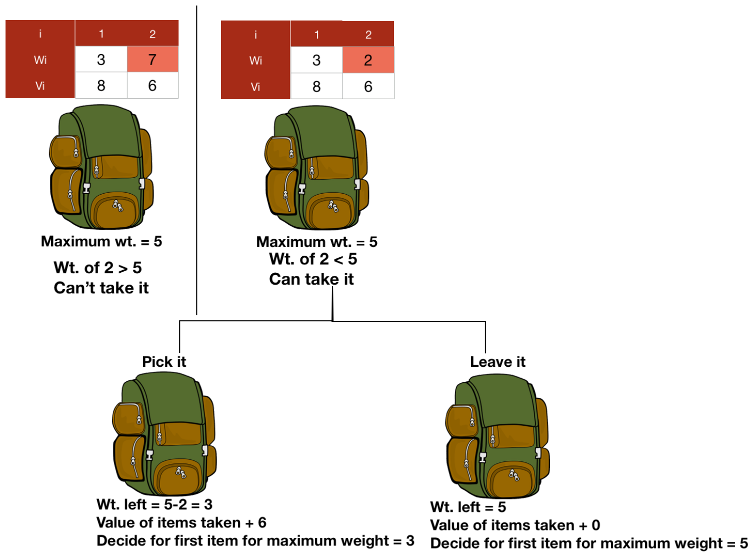


We can pick the 3^{rd} and the 4^{th} item with a total value of $9+6=15$ and a total weight of $4+1=5$ and beat the value of 14.

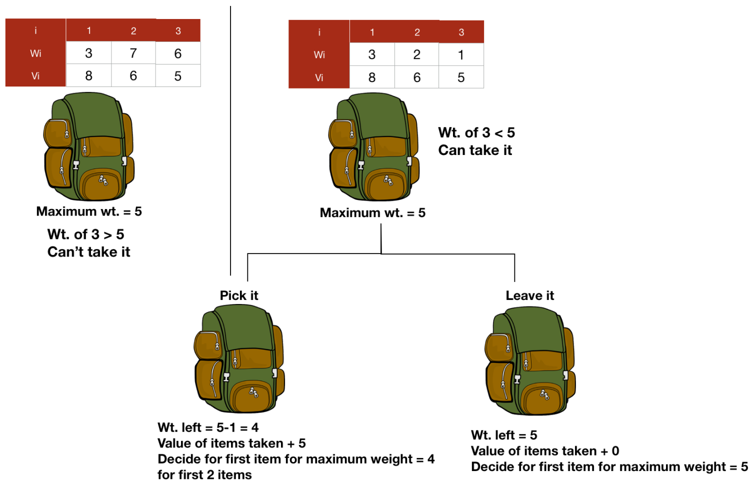
So, let's find out another technique to solve this problem.

Let's say there is only the first item available for us. If its weight is within the limit of the maximum weight i.e., $w_1 \le W$ then we will pick it otherwise, not.

Now, take the case when there are first two items available for us. If the weight of the second element is not within the limit i.e., $w_2 > W$, then we are not going to pick it. But if its weight is less than W, then we can either pick it or not. In case of picking it, the value of the second element (v_2) will be added to the total value and we will now decide for the optimized value we can get from the first element for $W - w_2$ weight limit and if we are not picking it, then we will decide for the first element for W weight limit. To get the maximum total value, we will choose the maximum of these two cases.

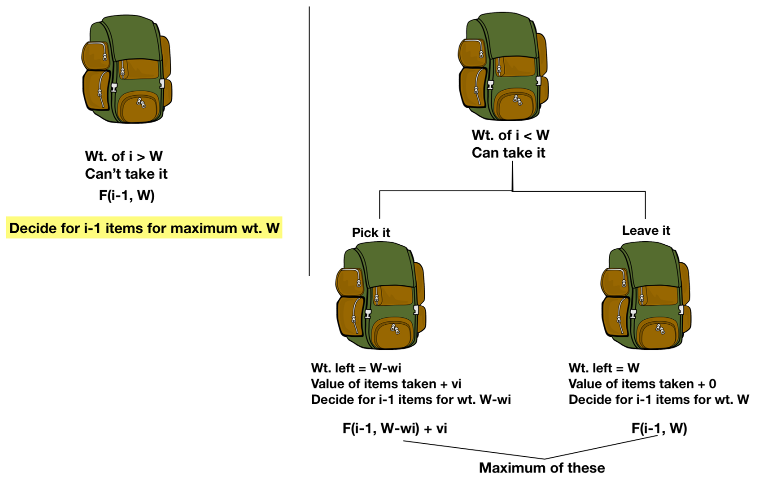


Similarly, for the first three elements, we can't pick the third element if its weight is not within the limit i.e., $w_3 > W$ and in this case, we will be left with the weight limit W and the first two items. If its weight is in the limit of the maximum weight, then we can either pick it or not. In case of picking it, the value of the third item (v_3) will be added to the total value and now we have to get the total optimized value of the first two elements and weight limit of $W - w_3$. In case of not picking it, we are just left with the first two elements and a weight limit of W. So, we will choose the maximum value from the latter two cases i.e., whether we are picking the 3^{rd} item or not.



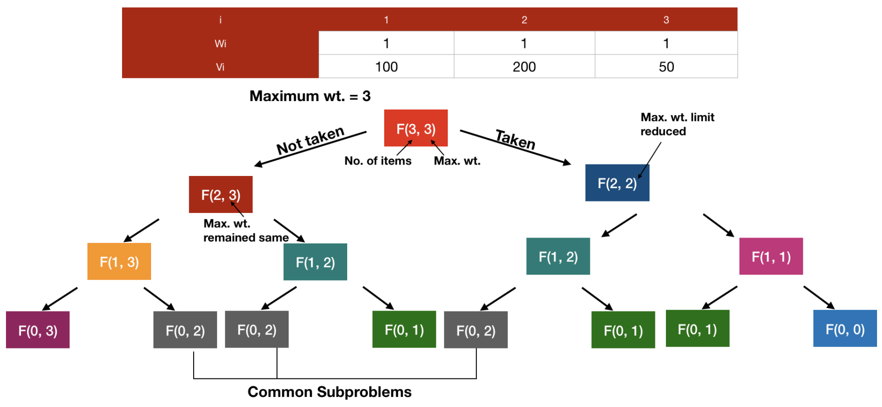
Basically, we are developing a bottom-up approach to solve the problem. Let's say that we have to make a decision for the first i elements to get the optimized value. Now, there can be two cases - the first that the weight of the i^{th} item is greater than the weight limit i.e., $w_i > W$, in this case, we can't take this item at all, so we are left with first $i-1$ items and with the weight limit of W; the second case, when $w_i \le W$, in this case, we will either take it or leave it.

Let's talk about the second case. If we will pick the item, we will add the value of the i^{th} item (v_i) to the total value and then we are left with a total of first $i-1$ items and the total weight limit will reduce to $W - w_i$ for rest of the items. And if we don't pick it, even then we are left with the first $i-1$ items but the weight limit will be still W. And then we will choose the maximum of these two to get an optimized solution.



Let's say $F(i, w)$ is a function which gives us the optimal value for the first i items and weight limit w, so we can write $F(i, w)$ as:

$$F(i,w) = \begin{cases} F(i-1,w), & \text{if } w_i > w \\ max\{F(i-1,w), (F(i-1,w-w_i)+v_i)\}, & \text{if } w_i \le w \end{cases}$$

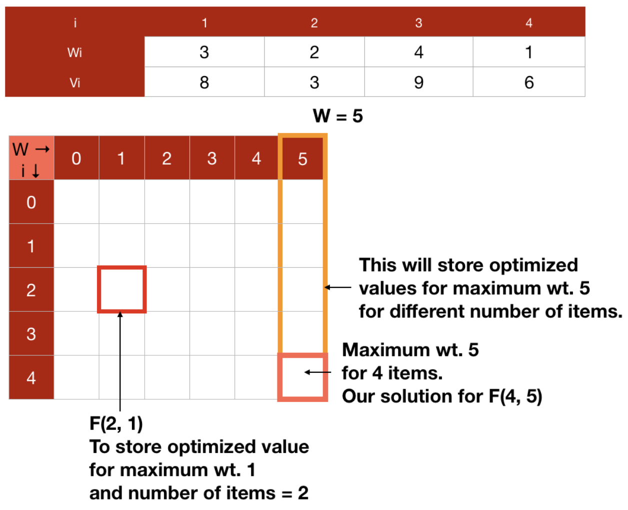


i	1	2	3
Wi	1	1	1
Vi	100	200	50

As you can see from the picture given above, common subproblems are occurring more than once in the process of getting the final solution of the problem, that's why we are using dynamic programming to solve the problem.

As we are using the bottom-up approach, let's create the table for the above function.

Solution Table for 0-1 Knapsack Problem

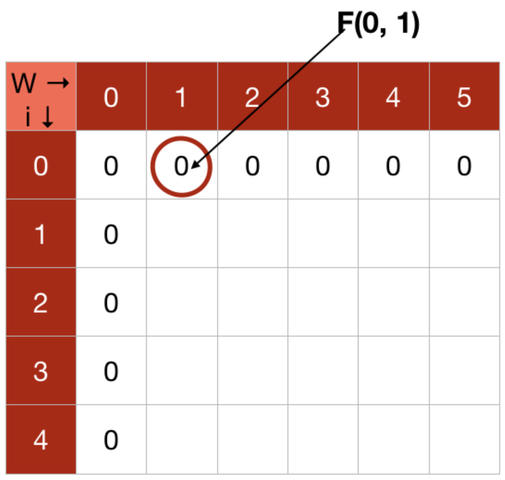


i	1	2	3	4
Wi	3	2	4	1
Vi	8	3	9	6

W = 5
W →
i ↓
0
1
2
3
4
5
This will store optimized values for maximum wt. 5 for different number of items.
Maximum wt. 5 for 4 items. Our solution for F(4, 5)
F(2, 1) To store optimized value for maximum wt. 1 and number of items = 2

The rows of the table are representing the weight limit i.e., the optimal value for each weight limit and the columns are for the items. So, the cell (i, j) of the table contains the optimal value for the first i items and for a weight limit of j units. Thus, we are going to find our solution in the cell (4,5) i.e., first 4 items with a weight limit of 5 units.

If the weight limit is 0, then we can't pick any item making so the total optimized value 0 in each case. This will also happen when i is 0, then also there is no item to pick and thus the optimized value will be 0 for any weight.



W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Now, let's start filling up the table according to the function we have derived above.

Starting from $F(1,1)$,
$w_1 = 3$
$W = 1$
$w_1 > W$
$F(i,W) = F(i-1,W)$
$F(1,1) = F(0,1) = 0$

Similarly, $F(1,2) = F(0,2) = 0$.

i	1	2	3	4
Wi	3	2	4	1
Vi	8	3	9	6

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0			
2	0					
3	0					
4	0					

For $F(1,3)$,
$w_1 = 3$ and $W = 3$
$F(i,W) = max\{F(i-1,W), (F(i-1,W-w_i)+v_i)\}$
$F(1,3) = max\{F(0,3), (F(0,0)+8)\}$
$= max\{0,8\} = 8$

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8		
2	0					
3	0					
4	0					

Similarly, $F(1,4) = max\{F(0,4), (F(0,1)+8)\}$
$= max\{0,8\} = 8$
and so on.

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	
2	0					
3	0					
4	0					

$F(1,5) = max\{F(0,5), (F(0,2)+8)\}$
$= max\{0,8\} = 8$

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0					
3	0					
4	0					

$F(2,1)$
$w_2 < W$
$F(2,1) = F(1,1) = 0$

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0				
3	0					
4	0					

$F(2,2) = max\{F(1,2), (F(1,0) + 3)\}$
$= max\{0, 3\} = 3$

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3			
3	0					
4	0					

$F(2,3) = max\{F(1,3), (F(1,1) + 3)\}$
$= max\{8, 3\} = 8$

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8		
3	0					
4	0					

Similarly,

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11
4	0	6	6	9	14	15

So, you can see that we have finally got our optimal value in the cell (4,5) which is 15.

Now, let's generate the code for the same to implement the algorithm on a larger dataset.

Code for Knapsack Problem

We already discussed that we are going to use tabulation and our table is a 2D one. So, let's start by initializing a 2D matrix i.e., `cost = [n+1][W+1]`, where n is the total number of items and W is the maximum weight limit. Since we are starting from 0, so the size of the matrix is (n+1)x(W+1).

Also, our function is going to take values of n, W, weight matrix (wm) and value matrix(vm) as its parameters i.e., `KNAPSACK-01(n, W, wm, vm)`.

Our next task is to make the cost of the items with 0 weight limit or 0 items 0.

```
for w in 0 to W
  cost[0, w] = 0

for i in 0 to n
  cost[i, 0] = 0
```

Now, we have to iterate over the table and implement the derived formula.

```
for i in 1 to n          // iterating on matrix
  for w in 1 to W        // iterating on matrix
    if wm[i] > w       // w_i > w, we can't pick it
      cost[i, w] = cost[i-1, w]    // F(i,w) = F(i-1, w)
    else           // we can pick the item
      if vm[i]+cost[i-1, w-wm[i]] > cost[i-1, w] // comparing F(i-1, w) with F(i-1, w-w_i)
```

```
          cost[i, w] = vm[i] + cost[i-1, w-wm[i]]
        else
          cost[i, w] = cost[i-1, w]
```

We are first iterating over the table and then checking if we can pick the item or not `if wm[i] > w`. In case of not picking it, we are moving to the i-1 items with the weight limit of w `cost[i, w] = cost[i-1, w]`.

While picking it, we are comparing maximum of F(i-1, w) and F(i-1, w-w_i) `if vm[i]+cost[i-1, w-wm[i]] > cost[i-1, w]` and then proceeding accordingly.

After the completion of the table, we just have to return the cell (n,W) i.e., `return cost[n, W]`.

Thus, the full code can be written as:

```
cost[n+1, W+1]
KNAPSACK-01(n, W, wm, vm)
  for w in 0 to W
    cost[0, w] = 0

  for i in 0 to n
    cost[i, 0] = 0

  for i in 1 to n
    for w in 1 to W
      if wm[i] > w
        cost[i, w] = cost[i-1, w]
      else
        if vm[i]+cost[i-1, w-wm[i]] > cost[i-1, w]
          cost[i, w] = vm[i] + cost[i-1, w-wm[i]]
        else
          cost[i, w] = cost[i-1, w]

  return cost[n, W]
```

C **Python** **Java**

```c
#include <stdio.h>

int cost[4+1][5+1];

int knapsack(int n, int W, int wm[], int vm[]) {
  int w, i;
  for(w=0; w<=W; w++) {
    cost[0][w] = 0;
  }

  for(i=0; i<=n; i++) {
    cost[i][0] = 0;
  }

  for(i=1; i<=n; i++) {
    for(w=1; w<=W; w++) {
      if(wm[i] > w) {
        cost[i][w] = cost[i-1][w];
      }
      else {
        if (vm[i]+cost[i-1][w-wm[i]] > cost[i-1][w]) {
          cost[i][w] = vm[i] + cost[i-1][w-wm[i]];
        }
        else {
          cost[i][w] = cost[i-1][w];
        }
      }
    }
  }
  return cost[n][W];
}

int main() {
  // element at index 0 is fake. matrix starting from 1.
  int wm[] = {0, 3, 2, 4, 1};
  int vm[] = {0, 8, 3, 9, 6};
  printf("%d\n", knapsack(4, 5, wm, vm));
  return 0;
}
```

Analysis for Knapsack Code

The analysis of the above code is simple, there are only simple iterations we have to deal with and no recursions. The first loops (`for w in 0 to W`) is running from 0 to W, so it will take $O(W)$ time. Similarly, the second loop is going to take $O(n)$ time.

Now, while iterating on the matrix, each of the statements is a constant time taking statement. And for every iteration of the first loop from 1 to n, the nested loop will be executed 1 to W times, thus it will take $O(n * W)$ time.

Also, the last statement (`return cost[n, W]`) is going to take constant time.

Since $O(nW)$ easily dominates $O(n)$ and $O(W)$, so our algorithm has a running time of $O(nW)$.

Till now, we have generated the algorithm to find the maximum value we can get from the listed items and the given weight limit but we don't yet know which items are really included in this optimal solution. So, let's take one step further and find out the items which are giving us this optimal solution.

Items Involved in Optimal Solution of Knapsack Problem

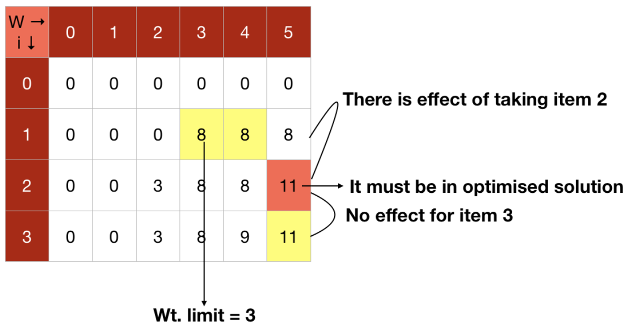
We can also get the items involved in the optimal solution by just using the cost matrix. Let's take an example where only first 3 items are available for us.

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11

We can see that the optimal value for a weight limit of 5 units is 11. Now, take the case when only the first two elements are available, then also the optimal value is 11 for the weight limit of 5 units. This means that the third item is not in our optimal solution because its value is not affecting the total optimized value.

W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11

And this is our strategy, move vertically upward in the table and see if the cell above the current cell has the same value. If the values of both the cells are different, then only we are going to include in our optimal solution.



W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11

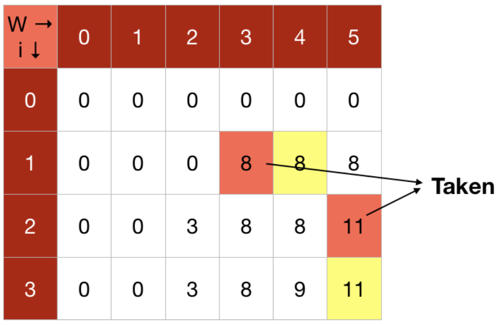
There is effect of taking item 2
It must be in optimised solution
No effect for item 3
Wt. limit = 3

Now, let's say we have rejected the third element and taken the second one (because $11 \neq 8$, so second item will be in the solution). As we have picked the second item, our weight limit is reduced from 5 to $5 - w_2$ = $5 - 2$ = 3. So, now we will move to the column of weight limit 3 and do the same thing.

Here, the first item and the 0th items have different values, so we will include the first item.

So, we ended up by picking the first and the second items having a total weight of $3 + 2 = 5$ and a total value of $8 + 3 = 11$.

Similarly, we can do the same thing for our problem.



W → i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11

Code for Items in Optimal Solution

We can start iterating the matrix backward from the cell (n,W) because this cell has the optimal solution and then compare it with the cell vertically upward i.e., `cost[i, j] == cost[i-1, j]`. If they are not equal, then we will print it and reduce the weight limit `j-wm[i]` and also move one row upward `i = i-1`. Otherwise, we will just move upward.

```
if cost[i, j] != cost[i-1, j]
   print i
   j = j-wm[i]
   i = i-1
else
   i - i-1 //don't include item, just move upward
```

Thus, the entire code can be written as:

```
ITEMS-IN-OPTIMAL(n, W, wm)
  i = n
  j = W

  while i > 0 and j > 0
    if cost[i, j] != cost[i-1, j]
      print i
      j = j-wm[i]
      i = i-1
    else
      i = i-1
```

C **Python** **Java**

(/add_ques

```
#include <stdio.h>

int cost[4+1][5+1];

int knapsack(int n, int W, int wm[], int vm[]) {
  int w, i;
  for(w=0; w<=W; w++) {
    cost[0][w] = 0;
  }

  for(i=0; i<=n; i++) {
    cost[i][0] = 0;
  }

  for(i=1; i<=n; i++) {
    for(w=1; w<=W; w++) {
      if(wm[i] > w) {
        cost[i][w] = cost[i-1][w];
      }
      else {
        if (vm[i]+cost[i-1][w-wm[i]] > cost[i-1][w]) {
          cost[i][w] = vm[i] + cost[i-1][w-wm[i]];
        }
        else {
          cost[i][w] = cost[i-1][w];
        }
      }
    }
  }
  return cost[n][W];
}

void items_in_optimal(int n, int W, int wm[]) {
  int i = n;
  int j = W;

  while (i > 0 && j > 0) {
    if(cost[i][j] != cost[i-1][j]) {
      printf("%d\n",i);
      j = j-wm[i];
      i = i-1;
    }
    else {
      i = i-1;
    }
  }
}

int main() {
  // element at index 0 is fake. matrix starting from 1.
  int wm[] = {0, 3, 2, 4, 1};
  int vm[] = {0, 8, 3, 9, 6};
  knapsack(4, 5, wm, vm);
  items_in_optimal(4, 5, wm);
  return 0;
}
```

You might have already noticed that with each chapter, we are digging deeper into algorithms. So, let's move to the next chapter and make our knowledge even deeper.

" There is no substitute for hard work. "

- Thomas A. Edison

PREV (/course/algorithms-dynamic-programming/) (/course/algorithms-rod-cutting/) NEXT

?