1 Dynamic programming

Dynamic programming isn't an algorithm by itself. This is rather technique that can be applied in different algorithms in order to increase their efficiency. Mostly it is applied in situations, where classic "brute force" approach is optimized. This technique relies on saving intermediate results, so same calculations do not have to be performed several times. Therefore, additional memory is nedeed. It is important to underline that each algorithm, which uses this technique, is different and specific for solving particular problem. Here we will present this technique on very common knapsack problem as well as on solutions of two problems, given in Google Kickstart competition.

1.1 Knapsack Problem

Even though knapsack problem and its dynamic solution are not untypical, it is still worth to debug behavior of the solution algorithm here. We will use following dataset to demonstrate it.

Item	1	2	3	4	5
Weight	2	3	3	4	5
Value	3	4	3	5	8

Problem is given as follows: we have knapsack of given capacity (in our example we use 15). We want to place in it as many items as possible in such a way, that total value is maximized.
Problem is solved by solving subproblems and saving partial results. We need to go through all items and all sizes of knapsacks.

Step 1 - define memory table
Prepare table $M[i, j]$ where partial results will be placed. Here i denotes number of items considered, while j denotes capacity of rucksack.

Step 2 - assign values for zero items
We will assign value 0 to $M[0, 0], M[0, 1], ..., M[0, 15]$. This is needed for further iterations. It simply means that 0 items in knapsack of capacity 0 to 15 are worth 0.

Step 3 - iterate through items
Here the most complex part of algorithm starts. We go trough items, sizes of knapsack, and consider following cases:

- item taken does not fit in given knapsack
- previous soultion for given capacity is better without considering given item
- taking current item is providing better solution

How these checks are actually made? Let us go step by step. At first we will consider first item. First part of algorithm says:
if weight of item is greater then capacity of knapsack, assign previous solution. In other words:

```
IF weight(i) > j:  M[i,j] := M[i-1,j]
```

We interpret it in a way that there is no possibility at all to fit a given item. So the only solution we can consider is the one, where we did not consider this element but $i - 1$ previous items. For first item, we come back

to zero items. Hence, for $M[1,0]$ and $M[1,1]$ we will assign value 0.
When we go to $M[1,2]$, it means we consider kanpsack of maximum weight 2 for this single item. So, here we will assign value of this item, 3. So, for $M[1,2],...,M[1,20]$ we have value 3.

Let us go to the second item - so now we consider items 1 and 2. As we stated already, if we have knapsack that cannot fit the examined item, we have no other way but use the previous solution.
E.g. we take knapsack of capacity 2, and have two items, 1 and 2. We already saved value for first item, $M[1,2] = 3$. We cannot fit here second item, cause it is too big. So the best solution for $M[2,2]$ will be "previous" value, so $M[1,2]$.
When we go step further, we examine solution for $M[2,3]$. We are choosing here one of two following answers:

- $M[i-1,3]$, so $M[1,3]$ which equals 3.
- $M[i-1, j-weight(i)] + value(i)$

How to interpret second option? We get back to $M[1,3-3]+4$. This means, go back to solutions from previous iteration. Then consider weight, that would allow this item to fit. Finally add value of item considered.
As a result we take better solution, which here will be 4. Going foward we will go to $M[2,5]$. Here we can notice, that we will come back to $M[1,5-3] = 3$. Then we will add value of this item. So, we know that both items fit in such knapsack and indeed assign value of both of them to this partial solution.

If we go to a third item, we can observe that for each knapsack we actually consider combinations of all three elements. But, we refer to solutions to previous step only, as these already contain results from combinations for first two elements.
Table below shows, what elements we do consider in each iteration.

Target value	Value 1	Value 2	Elements finally considered
$M[3,0]$	$M[2,0]=0$		()
$M[3,1]$	$M[2,1]=0$		()
$M[3,2]$	$M[2,2]=3$		(1)
$M[3,3]$	$M[2,3]=4$	$M[2,3-3]=0+3$	(2)
$M[3,4]$	$M[2,4]=4$	$M[2,4-3]=0+3$	(2)
$M[3,5]$	$M[2,5]=7$	$M[2,5-3]=3+3$	(1,2)
$M[3,6]$	$M[2,6]=7$	$M[2,6-3]=3+3$	(1,2)
$M[3,7]$	$M[2,7]=7$	$M[2,7-3]=3+3$	(1,2)
$M[3,8]$	$M[2,8]=7$	$M[2,8-3]=7+3$	(1,2,3)

Here it is very well visible for $M[3,5]$. We know that here we could either place items (1,2) or (1,3). For comparison we take $M[2,5]$, so value where solution for items 1 and 2 is stored. From the other side, we take $M[2,2]$. There, value 3 is stored, which corresponds to item 1. To that, we add value of item 3, so we get finally 6 for comparison. As we can see, it is better to take items 1 and 2, cause together they have better value than 1 and 3.
Similarly, for value $M[3,8]$, we know that all three items would fit. To calculate Value 2, we take $M[2,5]$, which corresponds to value of items 1 and 2, and still are able to add value of third item. Obviously it is better than simply taking $M[2,8]$ where only two first items where stored.

Answer to our problem will be placed in last cell of table M, in our case it will be in $M[5,15]$. It is also worth to underline that in this problem, solution will be proper regardless of order of elements that are considered.

1.2 Plates

This problem is given as follows:

1. We are given set of stack
2. Each stack has a given, same number of plates
3. Each plate is given a beauty index
4. Task is to take a given number of plates and maximize total beauty index of plates taken
5. There is one restriction, if we want to take a plate from the stack, we also need to take all plates that are above this plate

Here is our example that we will use to explain the solution. We have three stacks:

10	80	40
10	50	60
100	10	50
30	50	100
Stack 1	**Stack 2**	**Stack 3**

Let's say, that we need to take 6 plates. (e.g. if we want to take plate 100 from Stack 1, we also need to take two "10" plates above.) We want to do it this way that total sum of numbers taken is maximized. Naive approach says that we take all possible combinations and return the best one. However, this approach is inefficient, especially if number of stacks and plates increases.

Step 1
In first step we calculate for each stack separately, sums of values for given number of plates taken:

1 plate taken	10	80	40
2 plates taken	10+10	80+50	40+60
3 plates taken	10+10+100	80+50+10	40+60+50
4 plates taken	10+10+100+30	80+50+10+50	40+60+50+100

If we store this sums in memory, we will not have to calculate them each time it is needed. This is needed in next step.

Step 2

Second step will be iteration through all stacks. Here actual *dynamic programming* technique is being used, as in each step we will refer to previously computed partial results.
We will go through each stack, from top to the bottom. Table below shows sums of stacks (as calculated above) with marked direction of calculation.

↓	10	80	40
	20	130	100
	120	140	150
	150	190	200
⟶			

Pseudocode for looking for solution looks as follows:

```
for each Stack:
   for i in [0..plates to be taken]:
      for x in [1, min(i, number of plates in stack)]:
         dp[Current stack][i] = max(dp[Current stack][i],
         sum[Current stack][x] + dp[Previous stack][i-x])
```

This sounds a little big enigmatic, so let's try to go trough it step by step. Our target is to take 6 plates. We examine first stack (most outer loop) and calculate for that stack, how much total value we get if we take i plates. This is the second and third loop. We can take up to four plates.
As a result we receive table dp which gives these values:
$dp[1][1] = 10$
$dp[1][2] = 20$
$dp[1][3] = 120$
$dp[1][4] = 150$
In our table dp with intermediate results, first index denotes, how many stacks we have visited. Second index determines, how many plates in total we have taken.
Then we go to second plate (most outer loop). Here we will calculate values into $dp[2]$, which will simply mean that we calculate values taking two stacks into consideration.
Here however, when we go through number of plates (middle loop) we also need to take into consideration previous results.
We will save our partial results in $dp[2]$ - this time we can go up to 6 plates, because there is more (eight, from two stacks) to be selected from. For each number of plates we decide how many plates to take from current stack and how many from previous stacks.

For instance, let's say we consider picking up 3 plates. We have following possibilities:

- Take 3 plates from previous solution and none from current stack
- Take 2 plates from previous solution and 1 from current stack
- Take 1 plates from previous solution and 2 from current stack
- Take 0 plates from previous solution and 3 from current stack

These possibilities describe very well this inner loop. We check each of such variants and take the one that returns the maximized solution.
Let's go in detail through first two iterations:

- We take 1 plate, so we have one iteration in inner loop. In this loop we compare two values:
  - Sum of i plates from current stack - this we take from stored sums, so here we refer to sum for Stack 2 and 1 plate - value is 80
  - Stored sum of i plates from previous stacks - this we take from $dp[1][1]$ - so one plate from one stack
- We take 2 plates, here we will have three iterations, where we consider following values:
  - $sum[Currentstack][2]$ and none from previous calculations
  - $sum[Currentstack][1]$ and $dp[1][1]$ (one from current, one from previous)
  - $dp[1][2]$ - two plates from previous stack

  We take maximum value and store in $dp[2][2]$ - which means two plates form total of two stacks
  Ok, so $dp[2][1]$ will be either 80 or $dp[1][1] = 10 \rightarrow$ we take 80.
  For $dp[2][2]$ we compare:

Number of plates	dp[1]	Sum for 2^{nd} stack
0	0	0
1	10	80
2	20	130
3	120	140
4	150	190

  $0 + 130 = 130$
  $10 + 80 = 90$

$20 + 0 = 20$
Therefore in $dp[2][2]$ we will save 130 - cause when we want to take two plates from two stacks, the best would be to take both of them from second one.

When we go to 6 plates we will go from variants, where we take 4 plates from previous stack (dp) and 2 from current stack, to the totally opposite variant where we take two stacks from dp and four from current stack.
As a result of this iteration, we will have values for $dp[2][1]$ to $dp[2][6]$.
In third iteration, we will refer only to third stack and calculations from $dp[2][i]$, so from already two previous stacks.

Final answer will be placed in $dp[3][6]$ - which simply means six plates taken from three stacks.

1.3 Energy stones

This problem is given as follows:

1. There is a rock monster that has collected energy stones for lunch
2. Each stone has different amount of energy (energy units)
3. Each stone can be eaten in different amount of time (eaten in seconds). However, for simplicity here we will use same time for each stone
4. Each stone loses given amount of energy with time, when not being eaten (energy units lost per second)
5. Monster can consume only one stone at a time
6. Question is, what is the maximum amount of energy that monster can receive from eating given stones

We will describe algorithm using following example:

Stone	Energy units	Eaten in seconds	Energy units lost per second
1	10	5	1
2	30	5	5
3	30	5	1
4	80	5	60

We will store partial results in table $M[i][j]$, where i is number of stones considered and j are possible time slots. To simplify logic, we assume that j will describe number of seconds that passed, so we start in $j = 0$.

Similarly as in oter problems described here, we also consider, whether the stone should be taken at each step or not. We will iterate through all items and all possible time slots. Our maximum time is sum of all times that are needed to eat all stones. In our example it will be then 20.
There is nevertheless one tricky thing, namely we have to consider adjusted energy at each time. Thus, when we are in time t, we consider two following cases:

- We do not eat stone, so we take $M[i+1][t]$
- Or we eat stone, which means we take $M[i+1][t + seconds(i)] + stoneenergy(t)$

Because of the fact of losing energy, we need to go backwards and "look to the future". Hence, in each step we try to imagine that we eat stone at time t and then consider situation where we at the next stone $(i+1)$ and we have eaten current stone $(t + seconds[i])$. Let's see stone by stone, how it would

look like.
We order stones by lost energy and start with stone number 1, but asume that all other stones were considered (that is why we will use index 4). For this stone we will define simple values in our M matrix. Also, we will go backwards in time, from last second to first second.
We will have M[1,20] = 0, as in last second stone does not have energy anymore. Going back, we will notice that is starts to have energy in 10^{th} second. Finally, we will get following matrix:

M[4,0]	10
M[4,1]	9
M[4,2]	8
M[4,3]	7
...	...
M[4,9]	1
M[4,10]	0
...	...
M[4,20]	0

Now, we also take second stone. We can collate results for first and second stone:

t	M[i+1,t]	Stone 3 energy	M[i+1, t+5]
0	10	30	5
1	9	29	4
2	8	28	3
...	...	...	...
5	5	25	0
...	...	...	...
10	0	20	0
11	0	19	0
...	...	...	...
20	0	10	0

Hence, in time $t = 0$ we take value 35, cause we assume we eat Stone 3 at time 0 and after its consuming time, 5 seconds, we take Stone 1, which at this point has energy 5.
Let us go one stone further, Stone 2. Similarly as above, we get following results:

t	M[i+1,t]	Stone 2 energy	M[i+1, t+5]	Value taken
0	35	30	25	55
1	33	25	24	49
2	31	20	23	43
...	...	...	...	...
5	25	5	20	25
...	...	...	...	...
10	20	0	15	20
11	19	0	14	19
...	...	...	...	...
20	10	0	0	10

As final value we select between second column, or sum of third and fourth one. If we think of that, in $t = 0$ we are taking Stone 2, with energy 30. It takes 5 second to be consumed, so to that we add value from previous calculation in $t+5$. And there we have already selected best option of eating two stones, so we have best combination of all three stones now. Similarly it works for other time slots. We can also observe, that starting from $t = 5$ we omit Stone 2, and take result only from eating Stone 3 and Stone 1.
Now, we consider last stone, Stone 4. With it, our table will look as follows:

t	M[i+1,t]	Stone 4 energy	M[i+1, t+5]	Value taken
0	55	80	25	105
1	49	20	24	49
2	43	0	23	43
...	...	...	...	...
5	25	0	20	25
...	...	...	...	...
10	20	0	15	20
11	19	0	14	19
...	...	...	...	...
20	10	0	0	10

Our calculation was done backwards, from $t = 20$ to $t = 0$. Hence, our answer fill we placed in first cell of matrix M.
This approach however needs one assumption. Namely, we need to consider stones in particular order, from the one that loses energy slowest to the one that loses it fastest.
With this assmumption we can also see, why it has sense to enumerate matrix M backwards. In first iteration, we did id as $M[4, t]$ even though we had only one stone. However, because we know that this stone was losing energy

slowest, then if its energy was 0, we knew that other stones also had value 0 at this time. But this is mostly syntax, and could be modified as long as we ensure that general approach stays the same.

Ok, but what if we have different eating time, as given in problem description? In this case, we still need to sort stones by how fast they lose energy. But to build this sort, we need to include two factors: time needed to eat stone and units of energy lost per second. We can calculate thing, which is called total loss of energy.
If we want to compare two stones, i and j, we can compare values: $Seconds_i * Losing_j$ and $Seconds_j * Losing_i$. Let us take a look at following example:

Stone	Energy units	Eaten in seconds	Energy units lost per second
1	10	20	1
2	30	5	5
3	30	100	1
4	80	5	60

If we want to compare Stone 1 and Stone 2, we will compare respectively $Seconds_2 * Losing_1 = 5 * 1$ and $Seconds_1 * Losing_2 = 20 * 5$. If we eat Stone 1 at first, Stone 2 will lose meanhwile 100 energy units (so its all energy). If we eat Stone 2 first, then Stone 1 will lose 5 units meanwhile. It means that Stone 1 loses energy slower comparing to Stone 2.
But such calculation means that we would have to compare all pairs and apply some kind of ordering. However, this rule can be provided in more straightforward way. Namely, for each stone we can compare fraction of $Seconds_i/Losing_i$. In this way, we combine these two factors, as intended. We also sort them from biggest to smallest value. Stones with losing value 0 will be then our first ones in this approach.
In our example Stone 1 will have value 0.05, while Stone 2 will have value 1. This comparison comes from mathematics, cause we do it as follows:

$$L1 * S2 \overset{?}{>} L2 * S1 \quad / : S1 * S2$$

$$\frac{L1*S2}{S1*S2} \overset{?}{>} \frac{L2*S1}{S2*S1}$$

$$\frac{L1}{S1} \overset{?}{>} \frac{L2}{S2}$$