

Knapsack Problem

Input: n objects with:
integer weights w_1, \dots, w_n
integer values v_1, \dots, v_n
total capacity B

Goal: find subset S of objects that:

a) $\sum_{i \in S} w_i \leq B$
b) $\max \sum_{i \in S} v_i$

The next problem we're going to discuss is the knapsack problem. In this problem, the input is n objects. For each object we're given its weight and its value. And we'll assume that the weights and the values are all integers. We'll denote the weights by w_1 through w_n and the values by v_1 through v_n . Now we're given one additional input parameter, which is the total capacity available, which we'll denote as capital B . Our goal is to find a subset of objects. We need the subset of objects to fit in the backpack. That means that their total weight is in most capital B . And we're trying to find the subset with maximum value, maximum total value.

So let's try to restate this in more precise mathematical terms. What do we mean by the total weight is in most capital B ? We want to look at those objects which are in our subset or chosen subset. So these are the i in S . We want to look at the weight of these. And we want to sum over the weight and we want that total weight to be in most capital B . The total value for a subset of objects is the sum over the objects and the subset of their individual values. And we're trying to maximize that sum. We're trying to find the subset of objects with maximum value, but maintaining that it fits in the backpack. So their total weight is in most, B .

Let's summarize the problem one more time just to make sure everybody understands. So we're giving as input, the weights and values. These are the $2n$ numbers, w_1 through w_n and the values v_1 through v_n . And we're also given the total capacity, capital B . Our goal is to find the subset of objects. So a subset of 1 through n , where that subset fits in the backpack. So the chosen subset has total weight at most capital B and the subset we chose has maximum total value. So we're trying to find the subset with maximum value - total value - and fits in the backpack. You can imagine some applications of this are, where we're scheduling jobs and we have limited resources or limited computation time and we want to choose the jobs with most value for us. But really, this is a nice toy example which is going to illustrate some different style dynamic programming solution. And then we're going to see many applications in the homework of some variants which use this style dynamic programming solution.

Knapsack Problem

Input: n objects with:
integer weights w_1, \dots, w_n
integer values v_1, \dots, v_n
total capacity B

Goal: find subset S of objects that:

a) $\sum_{i \in S} w_i \leq B$
2 b) $\max \sum_{i \in S} v_i$

Two versions: ① one copy of each object - without repetition
2) unlimited supply - with repetition

Now, there are two natural variants of this problem, and both have different dynamic programming solutions. So it'd be useful to look at both.

- In the first version, there's one copy of each object. So we're trying to find a subset without repetition.
- In the second version, there's unlimited supply of each object. So we can use an object as many times as we'd like. In this version, the subset S has repetition possibly. So the subset S is actually a multiset.

To summarize, in the two versions of the problem, there is either unlimited supply of each object, so we can use it as many times as we'd like, or there's at most one copy of each object that we can use.

We're going to start up by looking at version 1: So we have at most one copy of each object that we can use, and then we'll go back, and we'll look at the second version of the problem where we have unlimited supply of each object. So let's dive in and look at the first version and try to design an algorithm for it.

Greedy algorithm?

<u>Example:</u>	<u>object:</u>	1	2	3	4
	<u>value:</u>	15	10	8	1
	<u>weights:</u>	15	12	10	5
		$B=22$			
	Solution:	<input type="text"/>		Subset:	<input type="text"/>

Now if you are presented with this problem in real life, the first approach you might try is a Greedy approach. Let's take a look at a specific example, and then this will highlight the pitfalls with the Greedy approach.

Now here's an example with four objects. The values are 15, 10, 8 and 1. The weights are 15, 12, 10, and 5. And, the total capacity will be 22. Now we're looking at the version where we have one copy of each object that we can use.

Let's take a look and make sure that we understand this problem. What is the optimal solution for this problem? What does the subset of objects which attain the maximum value while fitting in the backpack?

Greedy algorithm?

<u>Example:</u>	<u>object:</u>	1	2	3	4
	<u>value:</u>	15	10	8	1
	<u>weights:</u>	15	12	10	5
		$B=22$			

Solution: 18 Subset: 2 & 3

Greedy: Sort objects by $r_i := \frac{v_i}{w_i}$ = value per unit of weight

Greedy solution: 1 & 4 value 16

For this example, the maximum value that we can obtain is 18, and that is obtained by using objects two and three. The total weight of these objects is 22, $12 + 10$ and the value we obtained from them, the total value is $10 + 8$ is 18. Now, let's compare this to the greedy algorithm.

What would a greedy approach do? A greedy approach would take the most valuable object and try to fill up the backpack as much as possible with that most valuable object. What is the most valuable object? That's not the one with the total maximum total value. It's instead the one with the maximum value per unit of weight. If the weights are in pounds or kilograms and the value is in dollars, then we're looking at the object with maximum to the higher value per pound or per kilogram. In summary, the greedy approach would sort the objects by their value per unit of weight, which is this, quantity r_i , which is states value divided by its weight. In this example the objects are already sorted by that ratio. We have that r_1, r_2, r_3, r_4 .

So now what would a greedy approach do? The greedy approach would try to add object one, if it can, in this case it can, then we go to object two, and it would try to add object two if it can put. In this example, once you add in object one, you have 15 units of weight. You only have seven units of weight remaining, so you can no longer add in object two. Then we go to object three. The next most valuable object. We would try to add it in, does it fit? No it doesn't fit. Then we try to add object four, if it can. In this example it can because $15 + 5$ is 20. It fits in the

backpack. So, the greedy approach could obtain the solution using objects one and object four. Notice that the total value of this solution, object one and object four is $15 + 1$, so it has total value 16, whereas our optimal solution has total value 18.

This example illustrates why the greedy approach fails. It would try to add an object one, and once it does that, it's filling up the backpack too much and it can no longer fit in object two or three, and it ends up being more useful to skip object one and, instead, add in objects two and three. If you want to make a sub optimal choice at the beginning to allow you to squeeze in more objects later on. Now, let's go back and try to make our dynamic programming solution for this problem.

DP Design: attempt 1

Step 1: Define subproblem

$K(i)$ = max value achievable using
a subset of objects $1, \dots, i$

Step 2: Express $K(i)$ in terms of $K(1), \dots, K(i-1)$

Recall our basic recipe for designing a dynamic programming algorithm. The first step is always to define the sub-problem in words. Our first attempt is always to try the same problem on a prefix of the input. Therefore, we let $K(i)$ be the max value achievable using a subset of the first i objects. All we've changed is we've changed the set of objects available to us from the first n objects, 1 through n , to a subset of objects, 1 through i . Our second step in our recipe for designing a dynamic programming algorithm is to find a recursive relation which expresses $K(i)$, the solution to the i sub-problem, in terms of smaller subproblems - in this case, $K(1)$ through $K(i-1)$.

DP Design: attempt 1

$K(i)$ = max value achievable using a subset of objects $1, \dots, i$ ← limit capacity available

Express $K(i)$ in terms of $K(1), \dots, K(i-1)$

Example: values =

15	10	8	1
----	----	---	---

 $B=22$
weights =

15	12	10	5
----	----	----	---

 K =

15	15	18	
----	----	----	--

↑ ↑ {2,3}
{1}

take suboptimal solution to $i=2$
capacity $\leq B - w_3$

To summarize, $K(i)$ is the max value we can obtain using a subset of the first i objects. And we're trying to find a recurrence which expresses $K(i)$, the solution to the i th subproblem in terms of smaller subproblems.

So let's go back and look at our earlier example and see if we get some idea for a recursive relation. In our earlier example, the objects had values 15 (the object 1), 10 (object 2), 8 (object 3) and 1 (for object 4). And their weights were 15, 12, 10, and 5.

Now, let's look at this one dimensional table K that we're trying to find a recursive relation for. Now let's look at our one dimensional table K , in this example and see if we can figure out a recursive relation for the solutions of $K(i)$ in terms of smaller subproblems.

So let's fill it in for this example:

- Let's start with $K(1)$. In this case, we're looking at a subset of object 1. So either we use object 1 or we use the empty set. Clearly using object 1 is better because it fits in a backpack and has total value 15. So $K(1)$ is 15 in this case.
- Now let's look at $K(2)$, $i=2$. In this case, we're looking at a subset of objects 1 and 2. So either we use both objects in this case they have total weight 27. So they don't fit in the

backpack so we can't use both objects. We can use either object 1 or object 2 or neither. And in this case in this example it's better to use object 1.

- Now let's go to $i=3$. What's the optimal solution for $i=3$? Well, in this case, we want to use objects 2 and 3. They have total weight 22. And this is our optimal solution to the entire problem as we saw before and that has total value which is a team. Now note this solution is obtained by using subsets 2 and 3 whereas our earlier subproblems, their solution was obtained by using object 1 only. Now the question is "can we obtain this $K(3)$ which in this case is 18 using $K(1)$ and $K(2)$ "?. But $K(3)$ is obtained by taking a suboptimal solution to $i=2$. We don't want to use the optimal solution because that doesn't allow us to add in object 3 into the backpack. There's not enough spare capacity available. So, we need to take a suboptimal solution to $i=2$. The key is that that suboptimal solution to $i=2$ has enough spare capacity to allow us to add in object 3. What we really want to do is we want to take the optimal solution to $i=2$ where the total capacity available is in most the total capacity in the original subproblem minus the weight from using object 3. If we're going to add object 3 to our solution then that takes weight w_3 . And then our capacity available goes down by w_3 . And we want to take the optimal solution to that smaller subproblem which is $i=2$ in this case and we want to look at the optimal solution with this capacity, with the smaller capacity. In this case that capacity is 12. And if we take the optimal solution for $i=2$ with total capacity 12, then object 1 no longer fits and only object 2 fits in there. So the optimal solution will be just using object 2 and the total value we obtained from that is 10. So this will have total value 10 and therefore we can append on object 3 onto it, and we get the solution 2 and 3. And we get the total value 18.

What we see from this example is that this definition of the subproblem does not suffice. We're not able to express $K(3)$ in terms of $K(1)$ and $K(2)$ because the solution to $K(3)$ does not build upon the solution to $K(i)$ – where $i = 1$ and $i = 2$. Instead it uses a suboptimal solution to $i=2$. What is that suboptimal solution? That suboptimal solution has limited capacity available. It has limited capacity available in order to allow us to later add in object 3 and obtain a better solution for $i=3$. This points us in the right direction because what we need to do is limit the capacity available for these subproblems. So in some sense we want to take a prefix of the objects, 1 through i , and we want to take a prefix of the capacity available. This is going to lead us to our second attempt for the design of a dynamic programming algorithm for this problem. We're going to define the subproblems so that it considers a prefix of the objects and it varies the capacity available.

DP Design: Attempt 2

Subproblem Definition:

For i & b where $0 \leq i \leq n$ & $0 \leq b \leq B$:
let $K(i, b) = \max$ value achievable using
a subset of objects $1, \dots, i$
& total weight $\leq b$

Our goal: compute $K(n, B)$

Now, let's revise our subproblem definition trying to utilize some of the insight we just gained. Now, our initial attempt at a subproblem definition was $K(i)$ is the max value we can obtain using a subset of the first i objects. Now, the problem was when we tried to express $K(i)$ in terms of the earlier subproblems $K(1)$ through $K(i-1)$, it didn't suffice to have the solution to $K(i-1)$. But in fact what we needed was the solutions to the $K(i-1)$ subproblem with the additional restriction that the total weight is no longer at most B , but it's at most $B - w_i$ (the weight of the i th object) because we're going to try to include the i th object and therefore, a weight available for the $K(i-1)$ subproblem goes down. So this might be a sub optimal solution when the weight is B , but we need the optimal solution for this restricted weight $(B - w_i)$.

Therefore, what we're going to do is we're going to have two parameters; i and b . i is going to specify the prefix of the objects that we're going to consider. And little b is going to specify the total weight variable. So then we're going to have a two dimensional table. Ok? Let's go ahead and formalize this.


We're going to have two parameters, i and b , as we just said. And i is going to be restricted between 0 and n , just as before. And little b is going to be restricted between 0 and B . And we're going to define the entry $K(i, b)$. This is the entry in our two dimensional table to be the max value which we can obtain using a subset of objects 1 through i that's a prefix of the objects just as before. And the additional restriction is that the total weight is at most little b .

Our goal in this problem is to compute the entry $K(n,B)$. This is the bottom right corner of this table. This corresponds to the max value which we can obtain using a subset of all n objects, n with total weight at most B . This is the original problem that we're trying to solve.

Recurrence: attempt 2

For i & b where $0 \leq i \leq n$ & $0 \leq b \leq B$:

let $K(i, b) = \max$ value achievable using
a subset of objects $1, \dots, i$
& total weight $\leq b$

$K = i$ 

if $w_i \leq b$:
then $K(i, b) = \max\{v_i + K(i-1, b-w_i), K(i-1, b)\}$
else $K(i, b) = K(i-1, b)$

Base cases: $K(0, b) = 0$ & $K(i, 0) = 0$

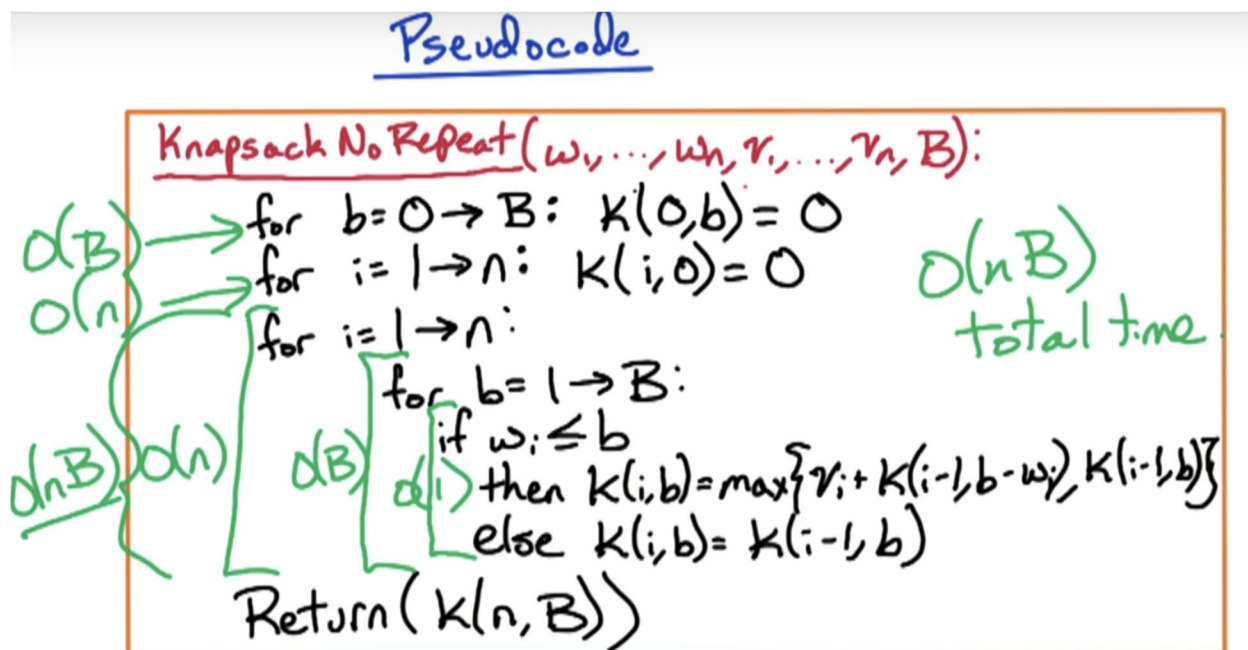
Now, let's summarize the recurrence that we have. Now, a recurrence is going to have two scenarios. Either we include object i , or we don't include object i . First off, we have to know whether object i even fits in the backpack or not. If it doesn't fit, then we know we cannot include object i . So, we have to condition on whether the weight of the i th object, w_i , is smaller than b or not. If it is smaller than b , then the i th object can fit in the back pack. So, we're gonna take the best of the two scenarios, either including object i or not including object i . If we include object i , we gain value v_i , for object i , plus we gain the value from the optimal solution to the subproblem which uses a subset of objects 1 through $i-1$ and has total capacity available $b-w_i$. The " $-w_i$ " is because we included...we forced w_i and object i to be included in the backpack. The other scenario is that we don't include object i in the solution and then the optimal solution to this subset of objects 1 through i is also going to be a subset of objects 1 through $i-1$, since object i is not being included. And the total capacity available - it stays the same. And we're going to take the best of these two scenarios, which means we're going to take the max of these two entries. In the other case, the weight of object i is strictly larger than b , and therefore it can't get included in the backpack. So then, our entry $K(i, b)$ is just going to be the second scenario. This defines a recurrence,

but to be complete, let's define the base cases and then we can go ahead and write the pseudo-code for our dynamic programming algorithm:

- Base cases: For the first row of our table, then $i=0$. That means we're taking a subset of objects which are the empty set. We're taking a subset of the empty set. Therefore,

there's no objects that can get included. So, the max value we can obtain is zero. Similarly, for the first column, we have total weight available, zero. Therefore, no objects can be included and therefore the max value we can obtain is, again, 0.

Now, we can go ahead and write the pseudo-code for our algorithm, but let's take a look first at how we're going to do it. We have this table. It's a two dimensional table and we're going to fill this table row by row. And the point is, that when we fill the entry $k(i,b)$, notice our recurrence, it always uses an entry from the previous row, either the entry right above is $k(i-1, b)$ or an earlier entry in that previous row. So, if we filled the table row by row and the entries we need for the smaller subproblems for our recurrence will be there, already completed in the table.



Now, let's write the pseudocode for our algorithm. We are solving the knapsack version where objects can get used at most once. So it's the no repeat version of knapsack that we're solving.

Now, the input to the algorithm are the weights for the n objects w_1 through w_n , the values for the n objects, v_1 through v_n , and the total capacity available B .

Now let's start with the base cases which are going to be the first row and the first column of the table. For the first row of the table, as we mentioned before, the entries are all going to be zero because we have a subset of the empty set which we are using. The first column of the table, we have total capacity zero available. So once again, we have the max value is zero.

Now let's fill the interior of this table and we'll do it row by row. So, we have a **for loop** where i varies from 1 to n . This will be the current row. And then we'll vary the parameter little b from 1 to B . This will bring us along the current row. To fill the entry $K(i, b)$, we have to first check whether fits in the current capacity available which is little b . So, we need to check whether the weight of the i th object, which is w_i , is smaller than little b or not. If the weight is smaller, then we have two scenarios. We can either include object i , or we don't include object i , and we're going to take the best of those two scenarios. If we include object i , we gain value v_i , and we gain the value from the optimal solution to a subset of the first $i-1$ objects, and with total capacity available $b-w_i$. Or if we don't include object i , we gain value $K(i-1, b)$ which is the optimal solution, which is a subset of the first $i-1$ objects, with the same capacity available.

And we are going to take the best of those two. So we're going to take the maxi of those two entries.

And the other scenario where object i does not fit in the current capacity available, then we know that the entry $K(i, b)$ is just the same as the previous row, $K(i-1, b)$, since the optimal solution, since it doesn't include object i , will be a subset of the first $i-1$ objects.

And finally, what is our algorithm going to return? It's going to return the bottom right entry of the table. This is the max value which we can obtain using a subset of objects 1 through n , and the total capacity B . This is our original problem that we're trying to solve, and that's the solution that we're trying to obtain.

Now, we can go and look at the running time of our algorithm.

- We first have our base cases, the first **for loop** is over B entries, that's over the first row. That takes time $O(B)$.
- Second **for loop** is over the first column, that's of size $O(n)$.
- Now we have our **nested for loops** which are going over the interior of the table. First one is of size $O(n)$, the second **nested for loop** is of size $O(B)$. And then, within this **nested for loop** is an **if-then-else statement** which is going to be $O(1)$. This is $O(1)$. This one is $O(B)$. Our outer loop is of size $O(n)$. And so, the total run time of these nested for loops is $O(nB)$. So, the total run time is $O(nB)$.

That completes the algorithm for the case where objects can be used at most once, and then we'll go back and we'll look at the solutions to the problem when we allow the object to be used multiple times.

(Slide 9) Knapsack in Poly-time?

Poly-time?
Running time: $O(nB)$
Efficient? is the running time polynomial in the input size?
Yes or No ← if No explain why not?

The running time with the algorithm we just described as $O(nB)$. The question I'd like to ask is that "is an efficient algorithm or not?" **What do we mean by efficient?** By efficient, we typically mean polynomial time. **What do we mean by polynomial?** We mean polynomial in the input size.

So, is this running time, of this algorithm that we just described, polynomial in the input size? So I'll let you answer, yes or no? Is it polynomial in the input size, the running time of our argument we just described? And if it's no, I want you to detail exactly why is it not polynomial in the input size, and detail what exactly would it mean to be polynomial in the input size? What would be the requirement on the running time for it to be polynomial in the input size?

(Slide 10) Knapsack in Poly-time? (Answer)

Poly-time?
Running time: $O(nB)$
NP-complete
Efficient? is the running time polynomial in the input size?
Yes or No ← if No explain why not?
To represent number B takes space $O(\log B)$
Goal: running time $\text{poly}(n, \log B)$

Now, the answer is No. This running time is not polynomial in the input size.

Why not? This is a polynomial. I mean this is a polynomial in n and B , but it's not a polynomial in the input size. Why not? The problem is this factor, B , if we wanna represent this number, B , it's just a number, right? How much space does it take to represent this number? The space required is the number of bits. What's the number of bits in B ? It's $\log(B)$. So, the input to this problem is this number, B , and to represent his number, B , it is $O(\log(B))$ space. And so, the input size is $O(\log(B))$. Now, of course, we also have n different numbers for the weights and the values. And those are going to each take $O(1)$ bits for each of those numbers, and there's the $O(n)$ of those numbers for the two end weights and values. So, the input size is n and $\log B$ size. So, our goal is a running time which is polynomial in n and $\log B$, whereas this is exponential in the input size. So, our running time is exponential in input size. Now, this is not surprising.

Why not? What we're going to see is that knapsack is NP-complete. What does that mean? It might be that there is a polynomial time algorithm for this problem, but the fact that it is NP-complete means that, if we design a polynomial time algorithm for this problem, this NP-complete problem, then every problem in NP will have a polynomial time algorithm. So, it's unlikely that we're going to design a polynomial time algorithm for knapsack because that

would imply polynomial time algorithm for a wealth of other problems. And many, many people have tried for many years, so it's unlikely that we're going to design it right now with the simple dynamic programming algorithm.

When we see the proof for this NP-completeness of the knapsack problem, it will be quite illuminating and you'll see why this algorithm is not efficient. Because what we'll do is we'll take a graph problem with n vertices, this will be a hard graph problem, and then we'll convert that into a knapsack problem. So, we reduce it to knapsack and we'll make a knapsack instance where this parameter B will be exponential in the graph size. So, this running time, where it will depend on B is polynomial, and B will give exponential running time for that original graph problem. And that will help it illustrate why this running time is exponential in the input size, whereas this is polynomial in the input size.

Knapsack: Repetition
Unlimited supply: can use an object as many times as we'd like

 $K(i, b) = \text{max value attainable from a multiset of objects } \{1, \dots, i\} \text{ with weight } \leq b$

Now, let's look at the other version of the knapsack problem. We gave a dynamic programming algorithm for the version of the problem where we have one copy of each object, so we can use each object at most, one time. Now, we'll look at the version of the problem where we have unlimited supply of every object. Here, we can use an object as many times as we'd like as opposed to the other version of the problem where we can use an object at most once.

Now, let's go ahead with our recipe for designing a dynamic programming algorithm. The first step is to define the subproblem. And, let's go ahead and try the same subproblem as what we used for the other version of the problem and see if that works. Again, try to gain some insight. So, our subproblem for the previous version of knapsack was $K(i, b)$ is the max value we can obtain using a subset of objects 1 through i with total weight, at most, little b . Now in this version, we're allowed to use objects multiple times. So instead of a subset where an object appears at most once, we're going to consider a multiset where an object can appear multiple times in the set. That's the only difference from the previous definition of the subproblem.

Knapsack: Repetition

Unlimited supply: can use an object as many times as we'd like

$K(i, b) = \text{max value attainable from a multiset of objects } \{1, \dots, i\} \text{ with weight } \leq b$

$K(i, b) = \max \left\{ \begin{array}{l} K(i-1, b), \quad \text{No more copies of object } i \\ v_i + K(i, b - w_i) \quad \text{another copy of object } i \end{array} \right\}$

Now, let's go ahead and see if we can write a recurrence for this subproblem definition. So let's try to express $K(i, b)$ in terms of smaller subproblems. We're going to try to use the insight that we had from the previous version of knapsack. So we're going to have two scenarios. Either we include object i or we don't include object i and we're going to take the best of those two so we're going to take the max. Now, as in the other version of knapsack, we're going to have two scenarios. In the other version of knapsack, we either included object i or we didn't include object i . In this version of the problem, we're going to have two scenarios. Either we include no more copies of object i or we're going to add in another copy of object i .

Now, in the first scenario where we have no more copies of object i then the remainder of the set is going to be a subset or a multiset actually of objects 1 through $i-1$ with the total capacity available staying the same. Therefore the solution is $K(i-1, b)$.

Now in the other scenario where we're adding in another copy of object. And for that copy of object we get value v_i . And in addition we get the optimal solution to this subproblem where the capacity went down by w_i . So that capacity went down by w_i . So the capacity available is now $b - w_i$. But notice here the first index is i , whereas in the other version of knapsack it was $i-1$ because in this version, we're allowed to use object i again, even another copy, additional copies, whereas in the other version of knapsack once we use the object i which is what's happening in this case then we could no longer use object i . So this went down to $i-1$ to keep

track that we didn't allow ourselves to use it multiple times.

Now let's take a look. Is this recurrence in fact a valid recurrence? Are we expressing this current subproblem in terms of smaller subproblems? Previously, when we wrote recurrence for the current entry, we always expressed it in terms of entries in previous rows. So this would be this one in row i and the previous ones were in a row $i-1$. But in this case, we're actually using the same row. Let's look at the table. We're going to fill this table row-by-row and when we get to this entry, $K(i, b)$, this current entry. Okay, we've filled up the previous rows and we filled up this current row up to that entry. Now, what entries does this recurrence require? Well, it requires the entry which is one row above. And in addition, it requires the entry which is in the same row but it's earlier in that row. So that will already be completed in the table. So these two entries that are required by this recurrence are already completed by the time we get to this current entry, $K(i, b)$. So it's a valid recurrence that expresses $K(i, b)$ - the current subproblem in terms of smaller subproblems. So we can go ahead and actually use the same pseudocode as before with the slightly different recurrence. Of course, we also have to condition on we have to make sure that the i th object fits in the remainder remaining capacity. So we can only do this case when this holds. So if w_i is at most b , then we take the best of these two scenarios. If w_i is bigger than b then we can only use this case just as before in the other pseudocode for the other version of knapsack.

And what's our running time going to be? Well, we got a table of size n times b . That's the size of our table and to fill each entry it's going to take us $O(1)$ time. So our running time is going to be $O(nB)$ just as before. So I won't detail it because it's almost the same pseudocode as for the other version of knapsack.


Knapsack: Repetition

Unlimited supply: can use an object as many times as we'd like

$K(i, b) = \text{max value attainable from a multiset of objects } \{1, \dots, i\} \text{ with weight } \leq b$

$K(i, b) = \max \{ \underbrace{K(i-1, b)}_{\text{if } w_i > b}, \underbrace{v_i + K(i, b-w_i)}_{\text{if } w_i \leq b} \}$

$O(nB)$

$K =$ 

Let's take a look at this algorithm for a moment. Often, when we get a solution which uses a two or three dimensional table, it's useful to look at it and see if we can simplify it to get a smaller table. And we might get a faster or less space or just a simpler solution. Okay? And that's what we're going to try to do here.

So, look at our solution here. Now, why do we have this parameter i ? The point of the parameter i in the original version of the knapsack problem, was to keep track of which objects we've considered or not. So, after we consider object i , then we can look at the first $i-1$ objects and look at a subset of those. But in this version of knapsack, we're allowed to use the object multiple times. So actually, it's not at all clear that we need to consider this parameter i . And in fact, we can get rid of it.

So, let's write a new version of knapsack solution which, for this version, we're only going to have a single parameter. So, we'll just have a parameter for the weight available and we'll drop this parameter for the subset of objects that we consider.

Simpler Subproblem

For b where $0 \leq b \leq B$:

$K(b) = \text{max value attainable using weight } \leq b$

Recurrence: try all possibilities for last object to add

$$K(b) = \max_i \{v_i + K(b - w_i) : 1 \leq i \leq n, w_i \leq b\}$$

$K = \boxed{\longrightarrow} \begin{matrix} B \\ \downarrow \\ \text{[shaded box]} \end{matrix}$

So, let's try to do our dynamic programming solution to this version of knapsack where we have a single parameter. The single parameter is going to be little b , corresponding to the total weight available. And this little b is going to vary between the maximum capacity available, B , and zero.

Now, we can define our subproblem. We define $K(b)$ as the max value obtainable using total weight, at most, little b . And we allow ourselves to use a subset of all n objects or actually a multiset of all n objects, whereas, in the previous subproblem, we have an extra parameter i , and we only allowed ourselves to use a subset of the first i objects.

Now, let's try to write a recurrence for this new subproblem definition. For the previous subproblem definition, in order to write a recurrence, we decided whether to include object i or not. Now, in this case, we don't have an object i , the last object. So, we want to try all possibilities for the last object to add.

So, the recurrence for $K(b)$ is going to be, we're going to try all possibilities for the last object to add and we're going to take the best of those. How do we get the best? We take the max, and we'll use i to denote the last object that we're trying to add. So, last object that we're going to add is going to be object i , and we'll consider all i between 1 and n . If we add object i , we gain value v_i . And in addition, we gain the optimal solution to the subproblem where the total weight goes down by w_i . This is expressed as $K(b - w_i)$. And we're trying all possibilities for i between

1 and n . But we need that the i th object fits in the backpack. We can have this weight, this could possibly be a negative number. So, we need that w_i 's are at most little b . So, we're trying all possibilities for the last object to add where that last object can be anything between object 1 and object n . We're trying all these n objects. And, if that object fits in the current capacity, so w_i is smaller than little b , we look at adding that object. So, we gain value v_i and then our capacity available goes down by w_i . So, with the remaining capacity, we take the best solution.

Now, since it's a one-dimensional table, be a straightforward to write the pseudo code. The table is one dimensional. There's not much choice in how we fill up the table. We're just going to fill it starting from $K(0)$ up to $K(B)$. And this last entry is the solution to our problem. Let's go ahead and write the pseudo code for this algorithm just to detail it.

Pseudocode

Knapsack Repeat ($w_1, \dots, w_n, v_1, \dots, v_n, B$):

for $b = 0 \rightarrow B$

$K(b) = 0$

for $i = 1 \rightarrow n$

if $w_i \leq b$ & $K(b) < v_i + K(b - w_i)$

then $K(b) = v_i + K(b - w_i)$

Return ($K(B)$)

So, here's the Pseudocode for our repeat solution for this version of knapsack. This is a knapsack version where we allow objects to be used multiple times. So, the repeat version of the knapsack.

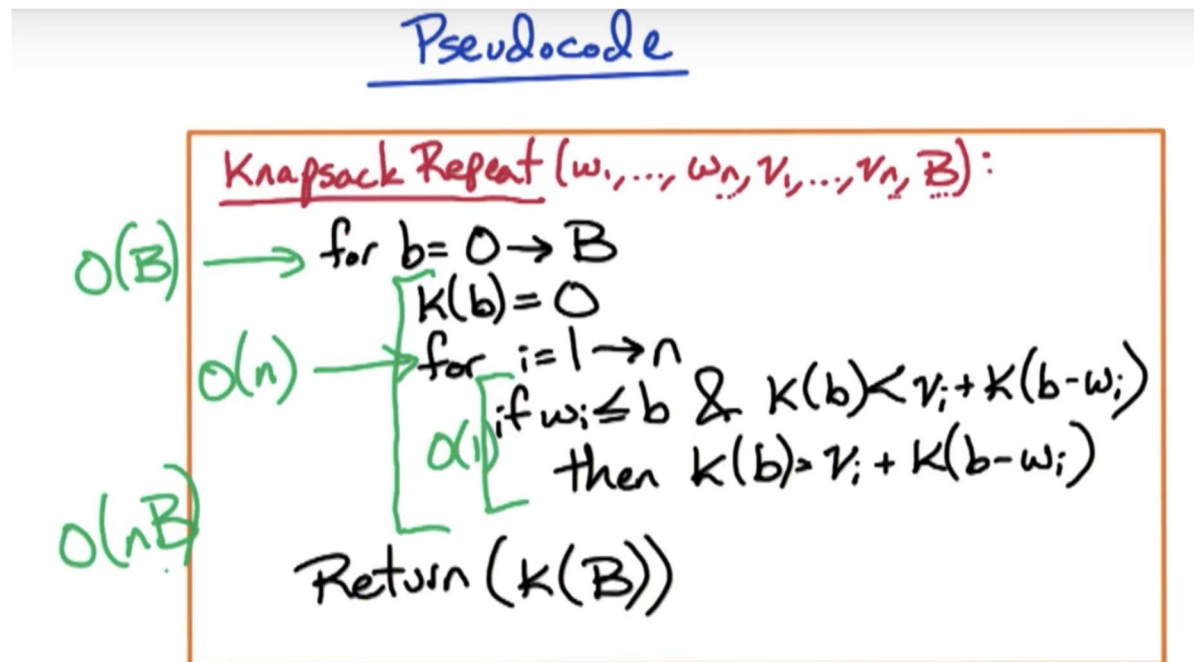
As before, the input to the problem are the weights of the n objects w_1 through w_n , the values of the n objects v_1 through v_n , and the total capacity available B .

Now, it's a one-dimensional table and has no base case to worry about.

And now, we're just going to go through that one-dimensional array from bottom up. Little b is going to be the index for our current position in the array. We start off by setting it equal to zero, in case there are no objects which fit in the current capacity available. Now, we go through each object and we consider that object as our last object to add in to the backpack. And we see if that gives us a better solution than anything we've obtained before. First, we need to check whether this object, object i , fits in the current capacity of $K(i, b)$ - if w_i is at most little b . And now, if it is, we check whether this obtains a solution which is better than anything we've seen previously for this index. So, the previous best solution is $K(b)$ and the new solution we obtain is v_i , for adding object i , plus the best solution for capacity $(b - w_i)$ which is $K(b - w_i)$. So, this is the solution we obtain now by using object i . And this is the

previous best solution. So, if the new solution is better than the previous best, then we're going to update the current best.

Now finally, we just returned the last entry of the table and that's our solution to our problem. That's the max value we can obtain using total weight, at most, B , which is the solution to the original problem.



Finally, let's take a look at the writing time. We have this one **for loop**, which is a size $O(B)$. Then, we have a **nested for loop** inside, it's of size $O(n)$. Each step, in this nested for loop, takes $O(1)$ time. Therefore, the total run time is $O(nB)$. It's the same run-time as the original solution we had to this version. The space is smaller, and also it's simpler solution. It's a very simple algorithm, just a one dimensional table.

Pseudocode

Output
the
multiset?

Knapsack Repeat ($w_1, \dots, w_n, v_1, \dots, v_n, B$):

for $b = 0 \rightarrow B$

$k(b) = 0, s(b) = 0$

for $i = 1 \rightarrow n$

if $w_i \leq b$ & $k(b) < v_i + k(b - w_i)$

then $\begin{cases} k(b) = v_i + k(b - w_i) \\ s(b) = i \end{cases}$

Return ($k(B)$)

So, to output the actual multi-set, which obtains the optimal solution, what we need to keep track of is "what is the last object we add in?" What is the object i , which we add in, which obtains this optimal solution? In order to maintain that, we make a separate array S . We initialize $S(b)=0$ corresponding to the empty set solution. And then when we update our current solution, so, when we get into this if then statement then we set $S(b)=i$ corresponding to that. The optimal solution for this subproblem is obtained by adding object i and then recursing on this smaller subproblem. Now, we can use this set S to backtrack. And, so, we can use it to produce a multiset, which obtains the maximum value. The details of that backtracking are similar to what we did for longest common subsequence.

Chain Matrix Multiply

Example: 4 matrices A, B, C, D

Goal: compute $A \times B \times C \times D$ most efficiently

Say A is of size 50×20
B is 20×1
C is 1×10
D is 10×100

$A \times B = \begin{bmatrix} \text{---} \end{bmatrix}_A \times \begin{bmatrix} \vdots \end{bmatrix}_B$

Our next dynamic programming problem is chain matrix multiply. This one will be a little different style from some of our early examples. Actually, the solution will be a bit more complicated than the earlier examples that we looked at.

So, let's look at a specific example so we can motivate this problem and then we'll go back and define the general problem. Our example will have four matrices A, B, C, D. Think of these matrices as having integer values for the entries. Our goal is to compute the product of these matrices A times B times C times D. And we'd like to do this in the most efficient manner possible.

What exactly do we mean by most efficient? Let's look at a specific example. Let's say, A has of size 50 by 20, so it has 50 rows and 20 columns, B is of size 20 by 1, C is of size 1 by 10, D is of size 10 by 100. Notice one thing, the number of columns of A has to match the number of rows of B. Also columns B has to match rows of C. Columns of C has to match rows of D. Why is that? When we multiply $A \times B$, what we do is we take a row of A, this is A, and we take a column of B and we take the inner product. So we multiply entering and then we add it to the product of these racks of these and so on. So the number of entries in this row has to equal the number entries in this column. And then we move onto the next row with the next column,

next row with the next column. So this row has to have the same number of entries as this column. That's why columns here has to equal the number of rows here.

(Slide 18) Order of Operation?

Which Parenthesization?

Goal: compute $A \times B \times C \times D$

standard way: $(A \times B) \times C \times D$

or $(A \times B) \times (C \times D)$

or $(A \times (B \times C)) \times D$

or $A \times (B \times (C \times D))$

which is best?

What is the cost?

Recall, our goal is to compute $A \times B \times C \times D$. Now, matrix multiplication is associative. So, there are many ways to compute them. The standard ways to compute $A \times B$. Take that product, $\times C$, take that product, $\times D$. But there are other things we can do. For instance, we can do $A \times B$ first, then we can do $C \times D$ second, and then we can multiply these two together. Or we can start with $B \times C$, multiply that with A , and finally, multiply that with D . Or you start with $C \times D$, multiply that with B , finally multiply that with A .

Which of these is best? That's what we want to determine. Which is the best or what is the cost of the optimal parenthesization. In order to figure out which is the best or most efficient method for computing the product of these matrices, we need to assign a cost for each of these operations. So, let's take a look again at matrix multiplication and then we can figure out a reasonable notion of cost.

Cost for Matrix Multiply

Take W of size $a \times b$ & Y of size $b \times c$
 $Z = W \times Y$ is of size $a \times c$ cost abc

$Z_{ij} = \sum_{k=1}^b W_{ik} Y_{kj}$

Z has ac entries \Rightarrow acb multiplications
 $ac(b-1)$ additions

Let's take two matrices, W , and Y , where W is of size $a \times b$. So, it's got a rows and b columns. And Y is of size $b \times c$. So, it's got b rows and c columns. And this will create the product of these matrices. So, we're looking at Z , which is $W \times Y$. Now, note that Z is going to be of size $a \times c$. It's going to have a rows, it's going to have the same number of rows as W and it's going to have c columns the same number of columns as Y .

Now, let's look at the multiplication a little more carefully. Now, here's an example where we're multiplying $W \times Y$, W has a lot of rows and a few columns, b columns. Similarly, Y has a few rows, it has b rows, and it's got a lot of columns, c . The product matrix, Z , is going to have a rows and c columns. Let's look at a specific entry Z_{ij} . row i . column j . How do we get this entry? What we do is, we take row i from W and we take column j from Y and do the inner product of those entries. So, we move along the row and the column, k going from 1 to b and we take the k th entry of this row times the k th entry of this column. We multiply those together and then we take the sum of these b terms. So, to compute this one entry Z_{ij} , it took us b multiplications and $b-1$ additions. Now, Z has $a \times c$ entries. So, in total, there will be acb multiplications and there'll be roughly the same number of additions. So, therefore, we're going to say the cost of multiplying these matrices is abc . Since these two terms are about the same and multiplications take longer than additions. So, this is a dominant factor. So, the cost of computing Z , the product matrix, W times Y is abc where W has size $a \times b$ and Y has size $b \times c$.

General Problem

For n matrices A_1, A_2, \dots, A_n where A_i is $m_{i-1} \times m_i$

Goal: What is the min cost for computing $A_1 \times A_2 \times \dots \times A_n$

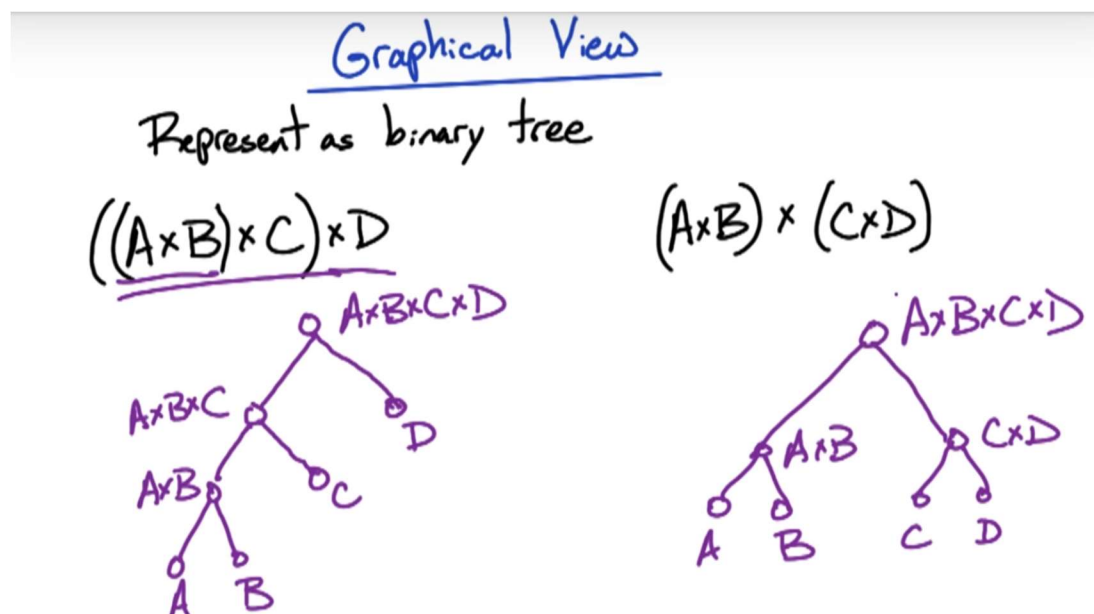
$m_0 \times m_1$ $m_1 \times m_2$ $m_{n-1} \times m_n$

Input: m_0, m_1, \dots, m_n

Goal: find min cost for computing $A_1 \times A_2 \times \dots \times A_n$

In the general problem, we have n matrices A_1, A_2 , up to A_n . Our goal is to determine the minimum cost for computing the product of these n matrices ($A_1 \times A_2 \times \dots \times A_n$). Now, the key parameter is the sizes of these matrices. So, we'll denote the size of the i th matrix, A_i , as m_{i-1} rows and m_i columns. So A_1 will be of size $m_0 \times m_1$. It has m_0 rows and m_1 columns. A_2 will have m_1 rows \times m_2 columns. So, the number of columns of A_1 matches the number of rows of A_2 . Finally, the last matrix has m_{n-1} rows \times m_n columns.

All we need for this problem is the sizes of these matrices. We don't need to know the entries of the matrices given our cost. Our cost just depends on the sizes of matrices. Therefore, the input to the problem are these sizes. These $n+1$ parameters defining the sizes of these n matrices. And our goal is to find the minimum cost for computing the product of these n matrices. And we're just putting the minimum cost, if we can do that, then we can go back and figure out the parenthesization which realizes that minimum cost.

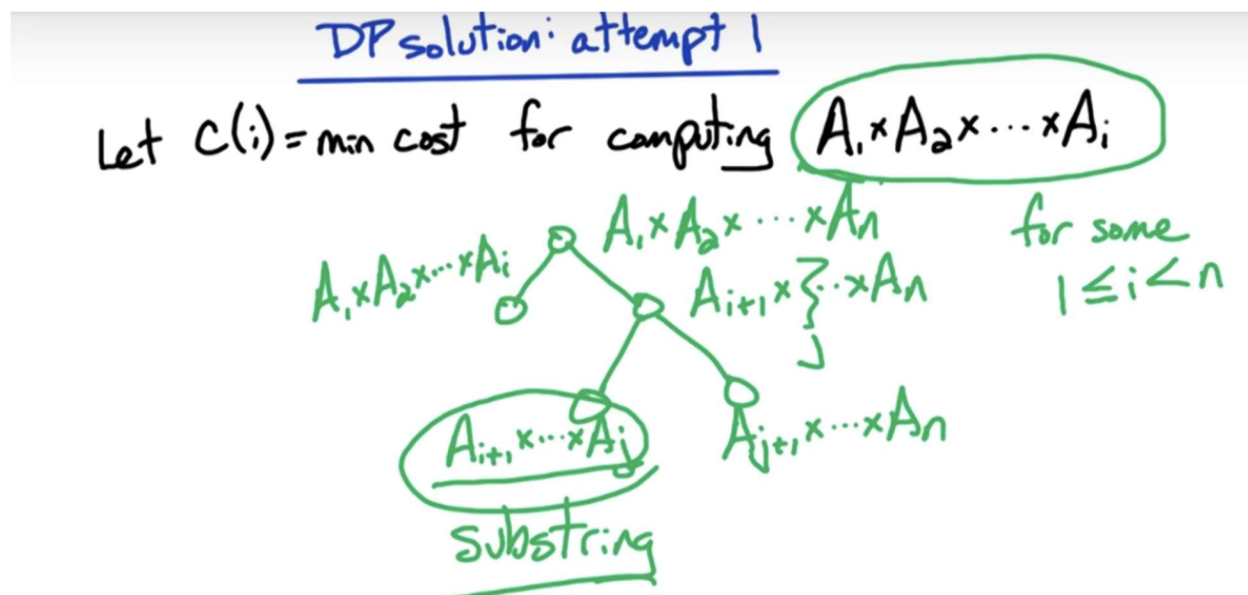


To get some intuition for this problem, I want to look at an alternative representation of the problem and instead of looking at it as parenthesization we're going to represent it as a binary tree. Let's go back and look at our earlier example to see what we mean here. In our earlier example, we were looking at the product of four matrices, A times B times C times D . Now, the standard way of computing this was A times B and take that times C and then take that times D . Now we're going to represent this as a binary tree. The leaves of the trees are going to be the four matrices and the internal nodes are going to represent the intermediate computations. So, the root is going to represent the final computation, A times B times C times D .

Now, for this parenthesization our first computation is A times B . So, we have the leaves for A and B and our first computation corresponds to the parent of those leaves, which corresponds to A Times B . Then, we take that matrix and multiply it by C . So, we have the leaf for C and then we have the parent of A times B and C , which corresponds to $A \times B \times C$. This internal node corresponds to $A \times B \times C$. How the subtree is structured tells us how the parenthesization is done for this subproblem. Finally, we take $A \times B \times C$ and we multiply by D . So, this tree captures this parenthesization - $((A \times B) \times C) \times D$.

Here's another parenthesization - $(A \times B) \times (C \times D)$ - we can do $A \times B$ and we can do $C \times D$ and then we multiply those two together. This is represented by the following tree. We first compute $A \times B$. Then we compute $C \times D$. So, this represents $A \times B$. This represents $C \times D$. And then, we multiply those two together. This gives us our final matrix, $A \times B \times C \times D$. So, the

roots of these two trees represent the same product, $A \times B \times C \times D$. How this subtree is structured tells us the parenthesization.



Now, let's go ahead and try to define our dynamic programming algorithm for this problem.

The first step in our recipe is to define the subproblem in words and we'll always try prefixes as our first attempt. Therefore, we let $C(i)$ be the minimum cost for computing the product of the first i matrices in the input.

Now, let's go back and see if we can define a recurrence for this subproblem definition. Let's look at our graphical view, our root, which we're trying to compute is $A_1 \times A_2 \times \dots \times A_n$, the product of these n matrices. In our graphical view, we're going to have a left child and right child. The left child is going to correspond to some prefix. It's going to be the product of $A_1 \times A_2 \times \dots \times A_i$ for some i . The right child is going to correspond to the product of $A_{i+1} \times A_{i+2} \times \dots \times A_n$. Now, let's look at a recurrence which is going to tell us the minimum cost for computing $A_1 \times A_2 \times \dots \times A_n$. What we're going to do is we're going to try all possibilities for the split i and then we're going to recursively look up what is the minimal cost for computing this subtree, which has root $A_1 \times A_2 \times \dots \times A_i$. And we're going to look up, hopefully, recursively, what is the minimum cost for computing this subtree, which has a root $A_{i+1} \times A_{i+2} \times \dots \times A_n$.

Now, we're aiming for prefixes, but this subproblem is a suffix. So, you might think, well, instead of just doing prefixes, why don't we just do prefixes and suffixes? Well, let's go one more level in this tree, in this binary tree and we'll see that it gets worse. Let's look at the children of this node ($A_{i+1} \times \dots \times A_n$). There's going to be some split here at some j , index j and the left's child is going to correspond to the product of $A_{i+1} \times A_{i+2} \times \dots \times A_j$ and the right child

is going to correspond to the product of $A_{j+1} \times A_{j+2} \times \dots \times A_n$. Now, we'd like to try all possibilities for the split index j and then we'd like to look up in our table the minimum cost for *this* $(A_{i+1} \times \dots \times A_j)$ subtree and the minimum cost for *this* $(A_{j+1} \times \dots \times A_n)$ subtree. Well, *this* $(A_{j+1} \times \dots \times A_n)$ subtree is a suffix. That's good.

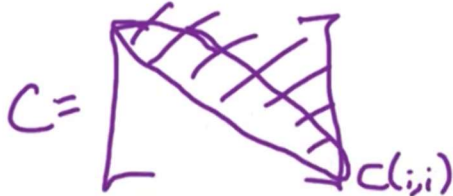
But what is *this* $(A_{i+1} \times \dots \times A_j)$ tree ? This is not a prefix or suffix. This is a substring. That's the key thing, is that all the intermediate computations are going to correspond to substrings. It's going to be some index i and some index j . And we're going to look at the product from i to j . And this is going to suffice ... to consider substrings. So, we're going to have to go back and revise our subproblem definitions, so that we don't consider prefixes, we're going to look at substrings.

DP substrings

For $i \& j$ where $1 \leq i \leq j \leq n$
let $C(i,j) = \text{min cost for computing } A_i \times A_{i+1} \times \dots \times A_j$

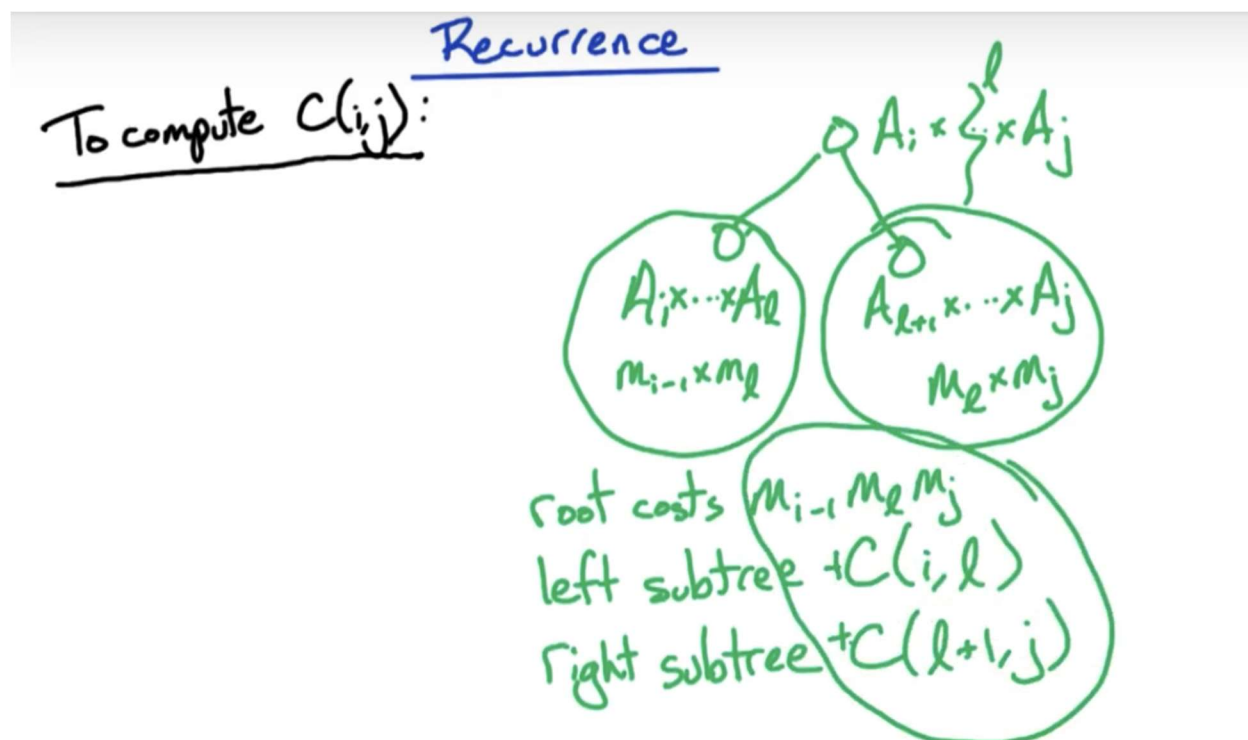
Recurrence for $C(i,j)$:

$C(i,i) = 0$



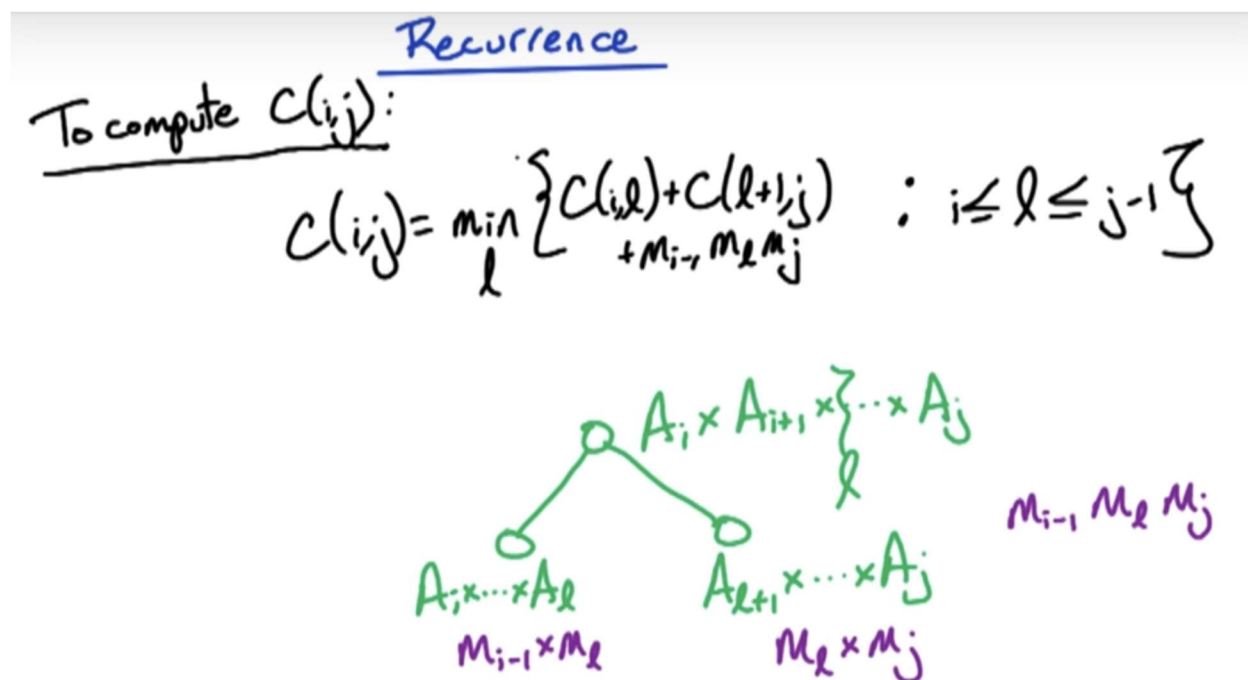
There's going to be two parameters, i and j - i is going to be the start of the substring; j is going to be the end of the substring. So, i is between j and 1 , and j is between i and n . And then we're going to define our subproblem as $C(i,j)$ is going to be the minimum cost for computing the product of the matrices A_i through A_j . Now, let's try to write recurrence for $C(i,j)$.

Let's start with the base case. What is the easiest case to compute for $C(i,j)$? That's the case when $i = j$. Then we're just computing A_i . So, the entry $C(i,i)$ what's the cost for it? It's zero, because there's no work to be done. Let's look at our matrix. Actually what do these base cases correspond to? Here's your matrix C , and these are the diagonal entries. And notice, we're computing the entries where j is at least i . So, we're just trying to do this upper diagonal over here. We're not trying to compute this lower diagonal of the matrix. Now, let's try to do in general what the recurrence for $C(i,j)$ is.



We're trying to find recurrence for the entry $C(i,j)$. This corresponds to computing the product of the matrices defined by the substring from i to j . Let's go back and look at our graphical representation. The root of the tree that we're trying to compute, corresponds to the product of the matrices A_i through A_j . Now, what are we trying to do? We're trying to find the split. Let's say at l and then the left subtree corresponds to the product of the matrices A_i through A_l . The right subtree corresponds to the product of the matrices A_{l+1} through A_j .

How is our recurrence going to work? We're going to try all possibilities for this index l for the split, and then we're going to look up in our table to minimum cost for computing *this* ($A_i \times \dots \times A_l$) subtree, which corresponds to a smaller substring, and we're going to look up in our table the minimal costs for computing *this* ($A_{l+1} \times \dots \times A_j$) subtree, and then we're going to combine those together. How much does it cost to combine them together? Well, *this* ($A_i \times \dots \times A_l$) matrix is of size m_{i-1} by m_l , and *this* ($A_{l+1} \times \dots \times A_j$) matrix is of size m_l by m_j . We multiply these together - it costs $m_{i-1} \times m_l \times m_j$. So computing the root costs ($m_{i-1} \times m_l \times m_j$), this amount computing the left subtree costs this entry ($A_i \times \dots \times A_l$, $m_{i-1} \times m_l$) corresponding to this substring which is $C(i,l)$ and similarly, the right subtree corresponds to the entry $C(l+1,j)$. So the total cost for this split at index l is the sum of these three ($m_{i-1} \times m_l \times m_j + C(i,l) + C(l+1,j)$). And what are we going to do? We're going to try all possibilities for l and we're going to take the one which has minimum sum.



Now, let's go ahead and write a recurrence for $C(i,j)$. In our graphical view, this corresponds to the root of the tree which corresponds to A_i through A_j . We're going to try all possibilities for the split at index l and we're going to take the best of those. The best means minimum costs - so we're going to do a minimization over the choices of l , and l is allowed to vary between i and $j-1$. And we have this left subtree and this right subtree. The left subtree corresponds to A_i through A_l . The right subtree corresponds to A_{l+1} through A_j . The minimum cost for computing this left subtree is the entry $C(i,l)$. The minimum costs for computing this right subtree is $C(l+1,j)$. Finally, we have to combine these together. Recall this product matrix is of size $m_{i-1} \times m_l$ and this product matrix is of size $m_l \times m_j$. To multiply these together, the cost is $m_{i-1} \times m_l \times m_j$.

Adding that term into our recurrence. We take the mean over the choices of l where l can vary from i to $j-1$. And for that specific l , the cost is the cost for the left subtree $C(i,l)$ plus the costs for the optimal right subtrees $C(l+1,j)$ plus the cost of merging that left subtree with that right subtree which is $m_{i-1} \times m_l \times m_j$. We take the sum of those three terms and we take the l which minimizes that sum. That's our recurrence for $C(i,j)$.

Filling Table

To compute $C(i,j)$:

$$C(i,j) = \min_l \left\{ C(i,l) + C(l+1,j) + m_i \cdot m_l \cdot m_j : i \leq l \leq j-1 \right\}$$

$C(i,i+1)$ uses $C(i,i)$ & $C(i+1,i+1)$

width $s=j-i$
 $s=0 \rightarrow n-1$

Before we detail the Pseudocode for this dynamic programming algorithm, let's go back and look at our recurrence a little more carefully, and see how we're going to fill the table up. This recurrence is a little different from earlier examples, so how we're going to fill the table up is actually going to be a little bit more complicated than before.

We're looking at this two-dimensional table C , and we're trying to compute the upper diagonal of this table. So those entries where j is at least i . Now, what was our base case? Our base case was the diagonal, these are the entries $C(i, i)$. This is the first thing we're going to fill in.

What is the next thing that we're going to fill in? The next entries we're going to fill in are the entries $C(i, i+1)$. Look at the recurrence for these entries. l is going to vary between i and that's it, that's the only choice for l . And then what are subproblems looks like? Our subproblems are $C(i, i)$ and $C(i+1, i+1)$. So to compute this entry, we use these diagonal entries which are there in our base case. What is this? What are these entries correspond to in our table? These are the off diagonals, these are the second type of entries that we're going to fill in. So we're going to first do the diagonal and then we're going to do these off diagonal. And in order to compute the off diagonal, we use the diagonal entries.

What is the next ones we're going to do? $C(i, i+2)$. Look at the recurrence in order to compute these, we're going to use either diagonal entries or the off diagonal entries. So there are going

to be there on the table.

Finally, what is the last one we're going to compute? It's this one right here that corresponds to $C(1,n)$ what is that? That's our final answer. That's the one we're trying to compute. This is the minimum cost to compute the product of matrices from A_1 up to A_n . So what our algorithm is going to do? It's going to start at this diagonal, and then it's going to move up, okay? How do we index that in our algorithm? Well look - look at this difference between the j and i . Let's call it the width and let's call that s . So $s = j - i$. For the diagonal entries which are the base cases, the width is $s = 0$. The off-diagonals, which we do next, all have $s = 1$, they have width 1, then we have width two, and so on until we get to width $n-1$. So we're going to vary the width from 0 up to $n-1$.

Now, we can go ahead and detail our Pseudocode for our dynamic programming algorithm.

Pseudocode

ChainMultiply(m_0, m_1, \dots, m_n):

$O(n) \rightarrow$ For $i = 1 \rightarrow n, C(i, i) = 0$

$O(n) \rightarrow$ For $s = 1 \rightarrow n-1$:

$O(n) \rightarrow$ For $i = 1 \rightarrow n-s$:

Let $j = i + s$

$C(i, j) = \infty$

$O(n) \rightarrow$ For $l = i \rightarrow j-1$:

$cur = m_{i-1} \cdot m_l \cdot m_j + C(i, l) + C(l+1, j)$
 If $C(i, j) > cur$ then $C(i, j) = cur$

Return $(C(1, n))$

$O(n^3)$ total time

Now, let's go ahead and detail the pseudocode for our dynamic programming algorithm to compute the minimum cost for multiplying these n matrices. Recall the input to the problem are the sizes...these $n+1$ numbers representing the sizes of the n matrices = m_0, m_1 , up to m_n . Let's start with the base case which corresponds to diagonal entering. The cost for these diagonal entries is zero since there's no computation to be done. Now, we're going to use our width parameter s . We already did the case where the width is zero. So we're going to start with one and go up two with $n-1$ which is our final solution. Then we have a parameter i which corresponds to the row. Notice that the rows are getting truncated at the end. Let's look at our matrix just to see what we mean by this. Our diagonal as these entries. And then, when we do the off-diagonal, I'm going to start at this entry $C(1, 2)$ and is going to end at this entry $n-1$. So it doesn't go down to the bottom row, okay? So that's why it stops at $n-1$. Once it tell you the index i and it tell you the width, then that defines the index j which is the end of the substring. Therefore, we let $j = i + s$. Now we're going to compute the entry $C(i, j)$. We're going to take a min and we're going to vary over l and keep track of the current min so far. So we're going to initialize the min - the value - the current minimum to infinity ($C(i, j) = \infty$).

- If using infinity makes you uncomfortable, you can think of setting this to some huge number.

Now, we're going to vary over the choices for the split at l . Recall l can vary between i and $j-1$. Now, for that split at l , let's look at the cost. Let's define a variable cur which is the current cost for the current index l . It costs $m_{i-1} \times m_l \times m_j$ to combine the left and right subtrees. $C(i,l)$ for the left subtree and $C(l+1,j)$ for the right subtree. And we want to compare this to the current best. So if the current best is larger than this value cur , then we're going to reset the current best to this current value. Then, the for loop. This is in this for loop, this is in this for loop. We have a bunch of nested for loops. Finally, what we return? Our final answer is the top right of our matrix.

We return this entry $(C(1,n))$ which corresponds to the cost - the minimum cost for computing the product of the matrices A_1 through A_n . That completes our dynamic programming algorithm.

Now let's take a look at the running time. We have this base case computation which takes $O(n)$ time. Now, we have this first for loop which is of size $O(n)$. Then, we have another nested for loop which is the size at most n . And now we have another for loop which is of size at most n , again. And then within these for loops, it takes $O(1)$ time for this computation. So we have three nested for loops of size $O(n)$ each. So the total run time is $O(n^3)$ total time.

So that completes our chain multiply dynamic programming algorithm. And the key thing here was that we, instead of using prefixes we had to move on to substrings for the subproblem definition. And then, how we filled in the table was a little bit more complicated. Usually, it's straightforward to fill in the table, we'd go row by row. But for this, when we were using substrings, we have to go from the diagonal and then work our way back up to the top row.

Practice Problems:

6.18

6.19

6.7 (Palindrome subsequence) substring

Subproblem: Try prefix, then substrings

- Problem 6.17, problem 17 in chapter 6, is about change making. Given a set of coins, a set of denominations, and a particular value, can you make change for that value using that set of denominations? In fact, there are three variants of the change making problem in the textbook (6.17, 6.18, and 6.19). I suggest doing all three.
- Another good problem is problem 6.20 which is about building an optimal binary search tree.
- There's problem 6.7 which is about finding the longest palindrome subsequence, and you might also try the variant where instead of doing a palindrome subsequence you look for a palindrome substring - so they have to be contiguous.

45 / 46

using substrings if necessary and then go back and check and think about whether actually substrings were required or not. It's better to have a correct algorithm which is a bit slower than an incorrect algorithm.

Once again, the key for getting fluent in dynamic programming is to do lots of practice problems. There are a lot of practice problems in the textbook, but there are a lot available in the web too from other courses and from other books. Do as many as you can and at some point you'll get the hang of it and they'll feel easy. The solutions will start to seem similar to each other but the only way to get to that point is to do lots of practice problems. So good luck. I hope you start to enjoy it once you get the hang of it.