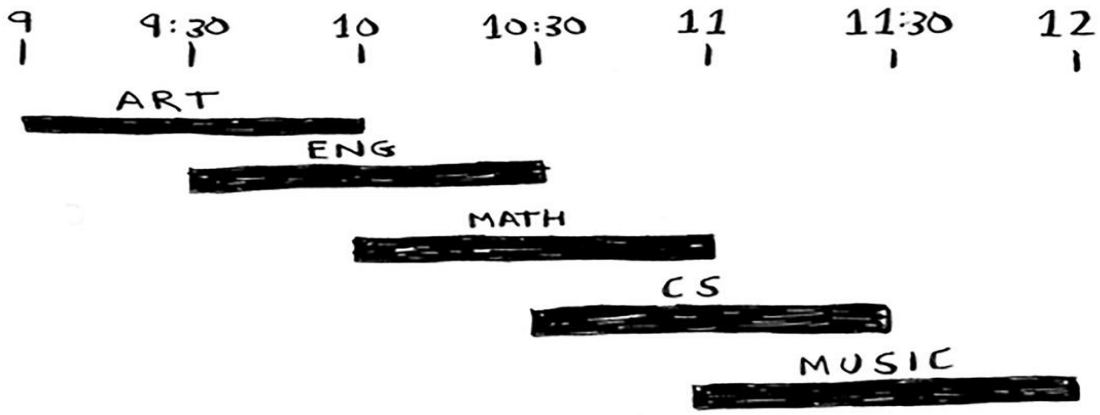
The classroom scheduling problem:

Suppose we have a classroom and want to hold as many classes here as possible.

We get a list of classes as follows:

CLASS	START	END
ART	9 AM	10 AM
ENG	9:30AM	10:30AM
MATH	10 AM	11 AM
C S	10:30AM	11:30AM
MUSIC	11 AM	12 PM

We can't hold all of these classes in there, because some of them overlap:



Problem Statement: We want to hold as many classes as possible in this classroom. How do we pick what set of classes to hold, so that we get the biggest set of classes possible?

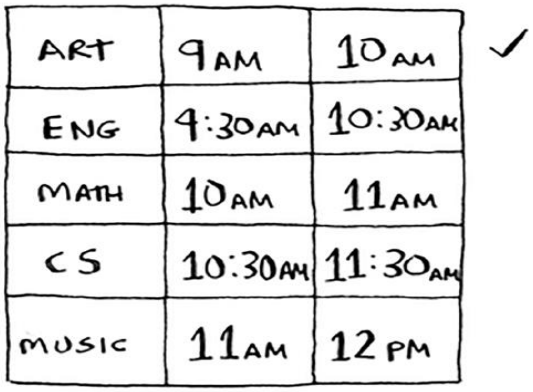
Sol^{n} It seems like a hard problem, right? Actually, the algorithm is so easy, it might surprise you. Here's how it works:

1. Pick the class that ends the soonest. This is the first class you'll hold in this classroom.

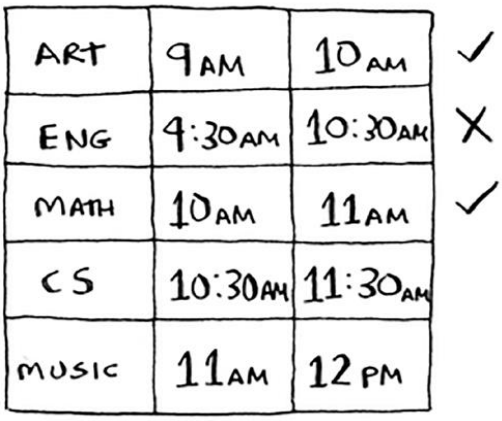
2. Now, we have to pick a class that starts after the first class. Again, pick the class that ends the soonest. This is the second class you'll hold.

Keep doing this, and you'll end up with the answer!

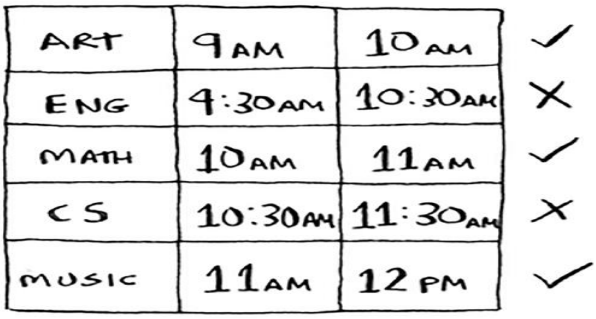
Art ends the soonest, at 10:00 a.m., so that's one of the classes you pick.



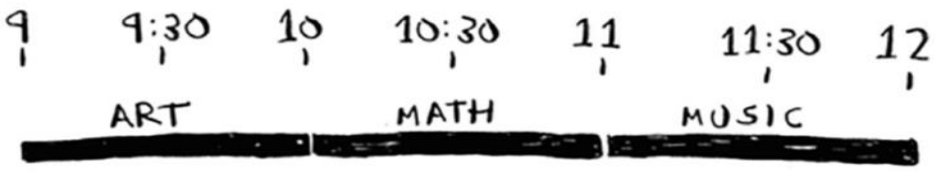
Now we need the next class that starts after 10:00 a.m. and ends the soonest. (English is out because it conflicts with Art, but Math works)



Finally, CS conflicts with Math, but Music works.



So, these are the **three classes** you'll hold in this classroom.



This algorithm may seem to be very easy and too obvious. One may think that so it must not be correct.

But that's the beauty of greedy algorithms: they're easy!

A greedy algorithm is simple: at each step, pick the optimal move.

(In this case, each time you pick a class, you pick the class that ends the soonest.)

In technical terms: at each step you pick the locally optimal solution, and in the end, you're left with the globally optimal solution.

Believe it or not, this simple algorithm finds the optimal solution to this scheduling problem!

Obviously, greedy algorithms don't always work. But they're simple to write!

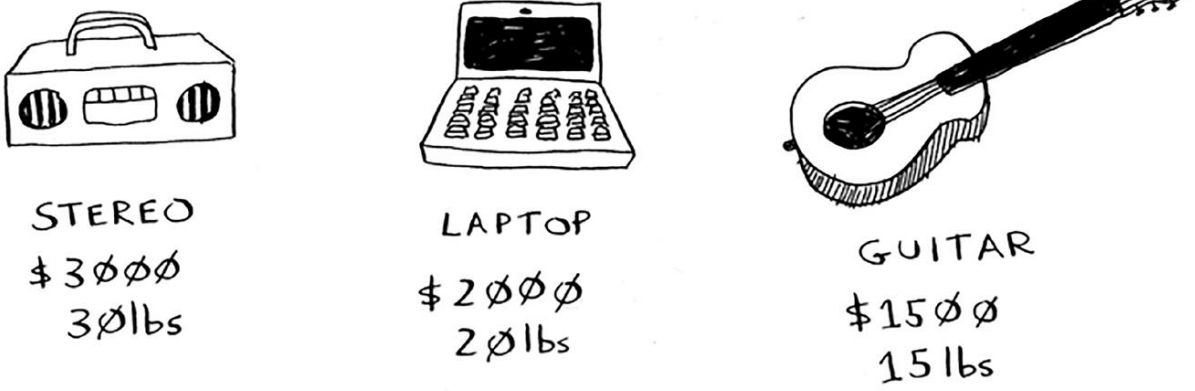
The knapsack problem:

Suppose you're a greedy thief. You're in a store with a knapsack, and there are all these items you can steal. But you can only take what you can fit in your knapsack. The knapsack can hold 35 pounds.

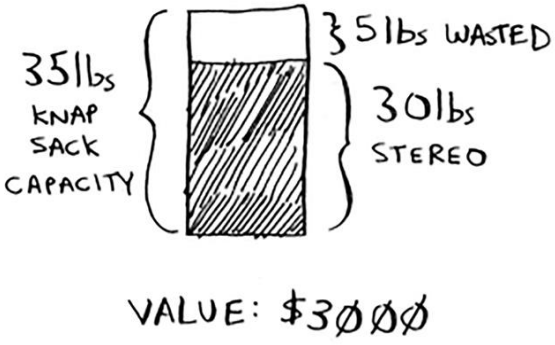
You're trying to maximize the value of the items you put in your knapsack. What algorithm do you use?

Again, the greedy strategy is pretty simple:
1. Pick the most expensive thing that will fit in your knapsack.
2. Pick the next most expensive thing that will fit in your knapsack. And so on.

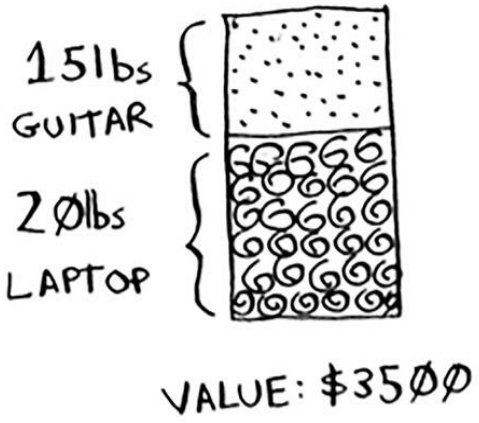
For example, suppose there are following three items you can steal.



Your knapsack can hold 35 pounds of items. The stereo system is the most expensive, so you steal that. Now you don't have space for anything else.



You got $3,000 worth of goods.
But wait! If you'd picked the laptop and the guitar instead, you could have had $3,500 worth of loot!



Clearly, the greedy strategy doesn't give you the optimal solution here. But it gets you pretty close. Using DP, we can explain how to calculate the correct solution. But if you're a thief in a shopping center, you don't care about perfect. "Pretty good" is good enough.

Here's the takeaway from this example: sometimes, perfect is the enemy of good. Sometimes all you need is an algorithm that solves the problem pretty well. And that's where greedy algorithms shine, because they're simple to write and usually get pretty close.

Reference: