1 Dynamic Programming

We continue our discussion on dynamic programming.

1.1 Problem: Knapsack

You are a burglar and are in the process of robbing a home. You have found several valuable items, but the sack you brought can only hold so much weight. What is the best combination of items to steal?

Some alternative formulations are:

- You are packing for a trip and your suitcase can only store so much stuff. You want to pack the most useful items for this trip.
- You're building a spacecraft, but launching a spacecraft is incredibly expensive based on its weight. So, you want to decide what are the best modules to put on a spacecraft.

Problem Statement: You have an available list of items. Each item has a non-negative integer weight. Your sack also has a capacity. The goal is to find the collection of items so that:

1. The total weight of all of the items is less than or equal to the capacity.
2. Subject to condition 1, the total value is as large as possible.

There are two slight variations of this problem:

- Each item can be taken as many times as you want.
- Each item can be taken at most once.

In our case, we'll assume that you can take each item as many times as you want.

1.1.1 Example: Knapsack

Suppose you have the following items:

Item	**Weight**	**Value**
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6. What is the best set of items to take, assuming you can only take one copy of each item?

We can brute-force this.

- With weights $1 + 2 + 3 = 6$ (A, B, C), you can get $1 + 4 + 3 = \$8$.
- With weights $4 + 2 = 6$ (B, D), you can get $5 + 4 = \$9$.

The other weights smaller than 6 do not give us the best choice. Thus, the best set of items to take is $\{B, D\}$ with a value of \$9.

1.1.2 Greedy Algorithms Don't Work

Problems like this one usually suggest greedy algorithms as a solution. However, greedy algorithms don't work. To see that this is the case, consider the following two counterexamples.

1. Most Valuable Item: Suppose the greedy algorithm defines "best" to be the most valuable item. Suppose you have a knapsack of capacity 6 and the following set of items that you can steal, where you can only take one of each item.

Item	**Weight**	**Value**
A	6	\$10
B	3	\$9
C	3	\$9

   Then:

   - The greedy solution would immediately go for A, since it is the most valuable item, which has value **\$10** and weight 6.
   - The optimal solution is B and C, which has value **\$18** and weight 6.

2. Biggest Value/Weight: Suppose the greedy algorithm defines "best" to be the item with the highest value to weight ratio. Again, suppose you have a knapsack of capacity 6 and the following set of items that you can steal, where you can only take one of each item.

Item	**Weight**	**Value**
A	4	\$5
B	3	\$3
C	3	\$3

   Then:

   - The greedy solution would immediately go for A, since A has the highest value/weight ratio, which has value **\$5** and weight 4. Since the other items have weight 3 and $4 + 3 > 6$, we can't pick any of those items.
   - The optimal solution is B and C, which has value **\$6** and weight 6.

1.1.3 Subproblems

Suppose we make *one* choice of an item to go into the bag. Then, what is left?

- The remaining items must have a total weight at most Capacity − Weight of Item.
- The total value of the items is

$$\text{Value of Item} + \text{Value of Other Items}$$

- Therefore, we want to maximize the value of the other items subject to their weight not exceeding the Capacity − Weight of Chosen Item.

Therefore, the subproblem is

$$\text{BestValue(c')}$$

where c' is the new capacity.

1.1.4 Recursion

What is BestValue(C), where C is the capacity?

- If there are *no items* in the bag, then
$$\text{Value} = 0$$
- If item i is in the bag, then the value is given by
$$\text{Value} = \text{BestValue}(C - \text{Weight}(i)) + \text{Value}(i)$$

So, the best attainable value for a sack with capacity C is the maximum of either:

- 0, *or*
- For the item i with weight such that $\text{Weight}(i) \leq C$ and $\text{Weight}(i)$ is maximal,
$$\text{Value}(i) + \text{BestValue}(C - \text{Weight}(i))$$

Therefore, the recursion is given by

$$\text{BestValue}(C) = \max(0, \max_{\text{Weight}(i) \leq C} (\text{Value}(i) + \text{BestValue}(C - \text{Weight}(i))))$$

Here, the 0 takes into account the possibility that the sack might be too small to fit any items.

1.1.5 Algorithm

The algorithm, which makes use of the recurrence above, can be described like so:

```
Knapsack(Wt, Val, Cap):
    Create Array T[0..Cap]
    // Consider all capacities from 0..Cap
    For C = 0 to Cap:
        // Accounts for the fact that the sack might be too small to
        // hold any items
        T[C] = 0
        // This is computing the maximum of all of the possible items
        For items i with Wt(i) <= C:
            If T[C] < Val(i) + T[C - Wt(i)]:
                T[C] = Val(i) + T[C - Wt(i)]
    Return T[Cap]
```

To see the runtime, we note that:

- There are $\mathcal{O}(\text{Cap})$ subproblems (the outer `for`-loop).
- For each subproblem, we need to (possibly) consider all of the possible items (the inner `for`-loop).

Therefore, the runtime is given by
$$\mathcal{O}(\text{Cap} \cdot \text{Num. Items})$$

1.1.6 Example: Knapsack Redux

Suppose you have the following items:

Item	**Weight**	**Value**
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6.

1. What is the highest possible value you can get, (assuming you can only take as many of the same item as you want)?

We can make use of the algorithm above. We define the table T like so:

C	0	1	2	3	4	5	6
BestValue							

- Start at $c = 0$. Then, it's trivial to see that you can't get anything of value since, well, your capacity is 0. So, $T[0] = 0$ and

C	0	1	2	3	4	5	6
BestValue	0						

- Now, we're at $c = 1$. By the algorithm we can initially set $T[1] = 0$. There are two choices that we can consider:
  - We can fit item A in and get \$1.
  - Or, we can fit nothing and get \$0.

  Clearly, the better choice is \$1, so we put that into $T[1]$.

C	0	1	2	3	4	5	6
BestValue	0	1					

- Next, we're at $c = 2$. By the algorithm we can initially set $T[2] = 0$. There are several choices we can consider.
  - We can fit item B in and get \$4.
  - Or, we can take A, along with $T[1]$, and get $\$1 + \$1 = \$2$.

  Clearly, \$4 is the better choice, so we put that into $T[2]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4				

- Next, we're at $c = 3$. There are several choices to consider.
  - We can fit item C in and get \$3.
  - We can fit item A, along with what we already have in $T[2]$, and get $\$1 + \$4 = \$5$.
  - We can fit item B, along with what we already have in $T[1]$, and get $\$2 + \$1 = \$3$.

  Clearly, \$5 is the better choice, so we put that into $T[3]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5			

- Next, we're at $c = 4$. There are, again, several cases to consider.
  - We can fit item D in and get \$4.
  - We can fit item C, along with what we have in $T[1]$, and get $\$3 + \$1 = \$4$.
  - We can fit item B, along with what we have in $T[2]$, and get $\$4 + \$4 = \$8$.
  - We can fit item A, along with what we have in $T[3]$, and get $\$1 + \$5 = \$6$.

  Here, we see that \$8 is the best choice, so we put that into $T[4]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8		

- Next, we're at $c = 5$. There are several choices to consider.
  - We can fit item D, along with $T[1]$, and get $\$5 + \$1 = \$6$.

  - We can fit item C, along with $T[2]$, and get $\$3 + \$4 = \$7$.
  - We can fit item B, along with $T[3]$, and get $\$4 + \$5 = \$9$.
  - We can fit item A, along with $T[4]$, and get $\$1 + \$8 = \$9$.

  Here, we see that \$9 is the best choice. So, we put that into $T[5]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	

- Next, we're at $c = 6$. There are several choices to consider.
  - We can fit item D, along with $T[2]$, and get $\$5 + \$4 = \$9$.
  - We can fit item C, along with $T[3]$, and get $\$3 + \$5 = \$8$.
  - We can fit item B, along with $T[4]$, and get $\$4 + \$8 = \$12$.
  - We can fit item A, along with $T[5]$, and get $\$1 + \$9 = \$10$.

  Here, we see that \$12 is clearly the better choice. So, we put that in $T[6]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	12

So, the answer is given by $T[6] = 12$.

2. What was the best possible set of items to collect?

Remember that our table T was populated to have the following values:

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	12

So, we can backtrack to find out what items we collected.

- First, we note that, to get \$12, we had to pick item B along with whatever was in $T[4]$. So, we know that **B** was an item that we collected. But, since we had to use whatever was in $T[4]$, we consider that entry next.
- At $T[4] = 8$, we note that, to get \$8, we had to pick item B along with whatever was in $T[2]$. So, we know that **B** was an item that was collected. Additionally, since we had to use whatever was in $T[2]$, we consider that entry next.
- At $T[2] = 4$, we note that, to get \$4, we had to pick item B. So, we know that **B** was an item that was collected. Since we didn't have to depend on the maximum value at some other entry, we're done.

Therefore, we collected three **B**'s for a total of **\$12**.

1.1.7 Non-Repeating Items: Subproblems

Suppose we tried to do this process with non-repeating items. We can try to repeat what we did above (with the subproblems) for this case. That is, suppose we put one item into the bag.

- Then, the remaining items must have total weight at most Capacity − Weight of Item.
- The total value of the items is

$$\text{Value of Item} + \text{Value of Other Items}$$

- Therefore, we want to maximize the value of the other items subject to their weight not exceeding the Capacity − Weight of Chosen Item **and** such that the chosen item(s) cannot be picked again.

The recursion needs to now keep track of the remaining capacity, since the item that was picked cannotbe used again. This makes the problem a little harder. So, we can try to make several subproblems (where c' is the new capacity).

1. Suppose the subproblem we use is $\text{BestValue}_{\neq i}(c')$. In other words, the best value we can achieve without using item i that doesn't go over the capacity. However, we **cannot** make a recursion out of this. This is because after using item j, the remaining items cannot include i and j. However, we need to create a recursion such that a recursive subproblem can be solved in terms of other recursive subproblems.
2. Suppose the subproblem we include is $\text{BestValue}_{\neq i, \neq j}(c')$. Again, we cannot make a recursion for this because this would imply the need to exclude a third item.
3. Suppose the subproblem we use is

   $$\text{BestValue}_S(c') = \max\left(0, \max_{i \in S}(\text{Value}(i) - \text{BestValue}_{S \setminus \{i\}}(\text{Cap} - \text{Weight}(i)))\right)$$

   In other words, the best value achievable using only items from S with total weight at most Cap. This recursion *works*, but this doesn't give us a very good algorithm. This is because the number of subproblems is more than $2^{\text{Number of Items}}$ since we're effectively considering every possible subset $(S \setminus \{i\})$.
4. Instead of trying the above approaches, let's think of something else. Suppose we have items coming along a conveyor belt. You decide, one at a time, whether to add the item to your sack. Then, the question we can think about is – do we take the last item or leave it alone?

   - If we take the item, the recursion becomes

     $$\text{BestValue}_{\leq n-1}(\text{Cap} - \text{Weight}(n)) + \text{Value}(n)$$

   - If we don't take the item, then the recursion becomes

     $$\text{BestValue}_{\leq n-1}(\text{Cap})$$

   If we tried to convert this into a recursion, then we only need subproblems of the form

   $$\text{BestValue}_{\leq k}(\text{Cap})$$

   In other words, what happens to the first k items? Somehow, by imposing this order, we don't need to deal with every possible subset of items.

1.1.8 Non-Repeating Items: Recursion

So, our recursion $\text{BestValue}_{\leq k}(\text{Cap})$ is defined by the highest total value of items with the total weight at most Cap using only items from the first k. Thus:

- Base Case: If $k = 0$, then we can't take any items. So:

  $$\text{BestValue}_{\leq 0}(\text{Cap}) = 0$$

- Recursion: $\text{BestValue}_{\leq k}(\text{Cap})$ is the maximum of either:
  1. $\text{BestValue}_{\leq k-1}(\text{Cap})$: You don't take the item, so we're finding the best value of the $k - 1$ items.
  2. If $\text{Weight}(k) \leq \text{Cap}$, then $\text{BestValue}_{\leq k-1}(\text{Cap} - \text{Weight}(k)) + \text{Value}(k)$: You take the item, add on the value, and then add on the best value of the $k - 1$ items.

1.1.9 Non-Repeating Items: Example

Suppose you have the following items:

Item	**Weight**	**Value**
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6.

1. Find the highest possible value you can get, assuming you can only take one copy of each item.

Instead of a one-dimensional table, we have a two-dimensional table T.

`Cap`	0	1	2	3	4	5	6
∅							
`A`							
`AB`							
`ABC`							
`ABCD`							

Here, we say that $T[\mathtt{x}, n]$ refers to the cell with the row being `x` (the items that we can pick) and the column being n (the capacity).

- At the row with the ∅, we essentially have $k = 0$. In other words, we're not able to pick anything. So, it follows that, regardless of capacity, if there's nothing to pick, we'll end up with \$0.

`Cap`	0	1	2	3	4	5	6
∅	0	0	0	0	0	0	0
`A`							
`AB`							
`ABC`							
`ABCD`							

- Suppose we're at the row with `A`; that is, we have $k = 1$ so we can pick one item, which in our case is A. Now, when the capacity is 0, clearly we cannot pick anything. But, as long as the capacity is greater than or equal to 1, we can pick A (value of 1). So:

`Cap`	0	1	2	3	4	5	6
∅	0	0	0	0	0	0	0
`A`	0	1	1	1	1	1	1
`AB`							
`ABC`							
`ABCD`							

- Suppose we're at the row with `AB`; that is, we're at $k = 2$ so we can pick two items, which in our case is A or B. Now, when the capacity is 0, clearly we cannot pick anything. When the capacity is 1, clearly we can only pick A, so we can take $T[\mathtt{A}, 1]$. When the capacity is 2, clearly the most optimal choice is B (value of 4). When the capacity is greater than 2, we can pick B and $T[\mathtt{A}, c]$, where c is the capacity such that $c > 2$ (value of 5).

`Cap`	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
`A`	0	1	1	1	1	1	1
`AB`	0	1	4	5	5	5	5
`ABC`							
`ABCD`							

- Suppose we're at the row with `ABC`; that is, we're at $k = 3$ so we can pick three items, which in our case is A or B or C.
  - Now, when the capacity is 0, clearly we cannot pick anything.
  - When the capacity is 1, clearly we can pick $T[\texttt{AB}, 1]$ (value of 1).
  - When the capacity is 2, clearly the most optimal choice is $T[\texttt{AB}, 2]$ (value of 4). This is because there isn't any space for item C.
  - When the capacity is 3, clearly the most optimal choice is just $T[\texttt{AB}, 3]$. Note that if we picked C, then we would be worse-off.
  - When the capacity is 4, clearly the most optimal choice is just $T[\texttt{AB}, 4]$. Note that if we picked C, then we would be worse-off.
  - When the capacity is 5, we can pick C and also $T[\texttt{AB}, 2]$ (value of 7).
  - When the capacity is 6, we can pick C and also $T[\texttt{AB}, 3]$ (value of 8).

  Thus:

`Cap`	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
`A`	0	1	1	1	1	1	1
`AB`	0	1	4	5	5	5	5
`ABC`	0	1	4	5	5	7	8
`ABCD`							

- Suppose we're at the row with `ABCD`; that is, we're at $k = 4$ so we can pick all four items if we can, which in our case is A or B or C or D.
  - Now, when the capacity is 0, clearly we cannot pick anything.
  - When the capacity is 1, clearly we can pick $T[\texttt{ABC}, 1]$ (value of 1).
  - When the capacity is 2, clearly the most optimal choice is $T[\texttt{ABC}, 2]$ (value of 4).
  - When the capacity is 3, clearly the most optimal choice is just $T[\texttt{ABC}, 3]$.
  - When the capacity is 4, clearly the most optimal choice is just $T[\texttt{ABC}, 4]$. Here, we could also just pick D.
  - When the capacity is 5, the optimal choice is just $T[\texttt{ABC}, 3]$ (value of 7). Note that the other option is D and A, which is worse.
  - When the capacity is 6, we can pick D and also $T[\texttt{ABC}, 2]$ (value of 9).

  Thus:

`Cap`	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
`A`	0	1	1	1	1	1	1
`AB`	0	1	4	5	5	5	5
`ABC`	0	1	4	5	5	7	8
`ABCD`	0	1	4	5	5	7	9

Thus, the final answer is given by $T[\texttt{ABCD}, 6] = 9$.

2. Find the items that you took to get the maximum value found in the previous part.

Using the table T from the previous part:

`Cap`	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
`A`	0	1	1	1	1	1	1
`AB`	0	1	4	5	5	5	5
`ABC`	0	1	4	5	5	7	8
`ABCD`	0	1	4	5	5	7	9

We can backtrack to find out what items we collected.

- Starting at $T[\texttt{ABCD}, 6]$, note that, in order to get to this position, we picked $T[\texttt{ABC}, 2]$ and D. So, we know that **D** was an item that we collected, and we additionally consider $T[\texttt{ABC}, 4]$.
- Now that we're at $T[\texttt{ABC}, 2]$, note that, in order to get to this position, we picked $T[\texttt{AB}, 2]$. As we didn't pick up any other items along th way, we can just consider $T[\texttt{AB}, 2]$.
- Now that we're at $T[\texttt{AB}, 2]$, note that, in order to get to this position, we picked **B**. Since we didn't consider any other entries in the table, we're done.

Thus, the solution is **B** and **D**.