Interview Cake

You are a renowned thief who has recently switched from stealing precious metals to stealing cakes because of the insane profit margins. You end up hitting the jackpot, breaking into the world's largest privately owned stock of cakes—the vault of the Queen of England.

While Queen Elizabeth has a *limited number of types of cake*, she has an *unlimited supply of each type*.

Each type of cake has a weight and a value, stored in objects of a `CakeType` class:

```java
public class CakeType {

    final int weight;
    final int value;

    public CakeType(int weight, int value) {
        this.weight = weight;
        this.value  = value;
    }
}
```

For example:

```java
// weighs 7 kilograms and has a value of 160 shillings
new CakeType(7, 160);

// weighs 3 kilograms and has a value of 90 shillings
new CakeType(3, 90);
```

You brought a duffel bag that can hold limited weight, and you want to make off with the most valuable haul possible.

Write a method `maxDuffelBagValue()` that takes **an array of cake type objects** and **a weight capacity**, and returns **the *maximum monetary value* the duffel bag can hold.**

For example:

Java ▾

```
CakeType[] cakeTypes = new CakeType[] {
    new CakeType(7, 160),
    new CakeType(3, 90),
    new CakeType(2, 15),
};

int capacity = 20;

maxDuffelBagValue(cakeTypes, capacity);
// returns 555 (6 of the middle type of cake and 1 of the last type of cake)
```

Weights and values may be any non-negative integer. Yes, it's weird to think about cakes that weigh nothing or duffel bags that can't hold anything. But we're not just super mastermind criminals—we're also meticulous about keeping our algorithms flexible and comprehensive.

Breakdown

The **brute force approach** is to try *every* combination of cakes, but that would take a really long time—you'd surely be captured.

What if we just look at **the cake with the *highest value?***

We could keep putting the cake with the highest value into our duffel bag until adding one more would go over our weight capacity. Then we could look at the cake with the *second* highest value, and so on until we find a cake that's not too heavy to add.

Will this work?

Nope. Let's say our capacity is **100 kg** and these are our two cakes:

Java

```java
new CakeType(1, 30);
new CakeType(50, 200);
```

With our approach, we'll put in two of the second type of cake for a total value of 400 *shillings*. But we could have put in a *hundred* of the first type of cake, for a total value of 3000 *shillings*!

Just looking at the cake's values won't work. **Can we improve our approach?**

Well, *why* didn't it work?

We didn't think about the **weight!** How can we factor that in?

What if instead of looking at the **value** of the cakes, we looked at their **value/weight ratio?** Here are our example cakes again:

Java

```java
new CakeType(1, 30);
new CakeType(50, 200);
```

The second cake has a higher value, but look at the value **per kilogram**.

The second type of cake is worth 4 shillings/kg ($200/50$), but the first type of cake is worth 30 shillings/kg ($30/1$)!

Ok, can we just change our algorithm to use the highest value/weight ratio instead of the highest value? We know it would work in our example above, but try some more tests to be safe.

We might run into problems if the weight of the cake with the highest value/weight ratio doesn't fit evenly into the capacity. Say we have these two cakes:

Java

```java
new CakeType(3, 40);
new CakeType(5, 70);
```

If our capacity is **8 kg**, no problem. Our algorithm chooses one of each cake, giving us a haul worth **110 shillings**, which is optimal.

But if the capacity is **9 kg**, we're in trouble. Our algorithm will again choose one of each cake, for a total value of **110 shillings**. But the *actual optimal value* is **120 shillings**—three of the first type of cake!

So even looking at the value/weight ratios doesn't always give us the optimal answer!

Let's step back. **How can we *ensure* we get the *optimal* value we can carry?**

Try thinking small. How can we calculate the maximum value for a duffel bag with a weight capacity of **1 kg**? (Remember, all our weights and values are integers.)

If the capacity is 1 kg, we'll only care about cakes that weigh 1 kg (for simplicity, let's ignore zeroes for now). And we'd just want the one with the *highest* value.

We could go through every cake, using a greedy approach to keep track of the max value we've seen so far.

Here's an example solution:

Java

```java
public static long maxDuffelBagValueWithCapacity1(CakeType[] cakeTypes) {

    long maxValueAtCapacity1 = 0L;

    for (CakeType cakeType : cakeTypes) {
        if (cakeType.weight == 1) {
            maxValueAtCapacity1 = Math.max(maxValueAtCapacity1, cakeType.value);
        }
    }

    return maxValueAtCapacity1;
}
```

(We're using `long` because we're looking for a *max* value.)

Ok, **now what if the capacity is 2 kg**? We'll need to be a bit more clever.

It's *pretty* similar. Again we'll track a max value, let's say with a variable `maxValueAtCapacity2`. But now we care about cakes that weigh 1 *or* 2 kg. What do we do with each cake? And keep in mind, **we can lean on the code we used to get the max value at weight capacity 1 kg.**

1. **If the cake weighs 2 kg**, it would fill up our whole capacity if we just took one. So we just need to see if the cake's value is higher than our current `maxValueAtCapacity2`.
2. **If the cake weighs 1 kg**, we could take one, and we'd still have 1 kg of capacity left. How do we know the best way to fill that extra capacity? We can use the max value at capacity 1. We'll see if adding the cake's value to the max value at capacity 1 is better than our current `maxValueAtCapacity2`.

Does this apply more generally? If we can use the max value at capacity 1 to get the max value at capacity 2, can we use the max values at capacity 1 and 2 to get the max value at capacity 3?

Looks like this problem might have overlapping subproblems!

Let's see if we can build up to the *given* weight capacity, *one capacity at a time*, using the max values from *previous* capacities. How can we do this?

Well, **let's try one more weight capacity by hand—3 kg.** So we already know the max values at capacities 1 and 2. And just like we did with `maxValueAtCapacity1` and `maxValueAtCapacity2`, now we'll track `maxValueAtCapacity3` and loop through every cake:

Java

```
long maxValueAtCapacity3 = 0L;

for (CakeType cakeType : cakeTypes) {
    // only care about cakes that weigh 3 kg or less
    ...
}
```

What do we do for each cake?

If the current cake weighs 3 kg, easy—we see if it's more valuable than our current `maxValueAtCapacity3`.

What if the current cake weighs 2 kg?

Well, let's see what our max value would be *if we used the cake*. How can we calculate that?

If we include the current cake, we can only carry 1 more kilogram. What would be the max value we can carry?

We already know the `maxValueAtCapacity1`! We can just add that to the current cake's value!

Now we can see which is higher—our *current* `maxValueAtCapacity3`, or the *new* max value if we use the cake:

Java ▾

```java
long maxValueUsingCake = maxValueAtCapacity1 + cakeType.value;
maxValueAtCapacity3 = Math.max(maxValueAtCapacity3, maxValueUsingCake);
```

Finally, **what if the current cake weighs 1 kg?**

Basically the same as if it weighs 2 kg:

Java ▾

```java
long maxValueUsingCake = maxValueAtCapacity2 + cakeType.value;
maxValueAtCapacity3 = Math.max(maxValueAtCapacity3, maxValueUsingCake);
```

There's gotta be a pattern here. We can keep building up to higher and higher capacities until we reach our input capacity. Because the max value we can carry at each capacity is calculated using the max values at *previous* capacities, we'll need to solve the max value for *every* capacity from 0 up to our duffel bag's actual weight capacity.

Can we write a method to handle **all the capacities?**

To start, **we'll need a way to store and update *all* the max monetary values for each capacity**.

We could use a hash map, where the keys represent capacities and the values represent the max possible monetary values at those capacities. Hash maps are *built on* arrays, so we can save some overhead by just using an array.

Java ▾

```java
public static long maxDuffelBagValue(CakeType[] cakeTypes, int weightCapacity) {

    // array to hold the maximum possible value at every
    // integer capacity from 0 to weightCapacity
    // starting each index with value 0 long
    long[] maxValuesAtCapacities = new long[weightCapacity + 1];
}
```

What do we do next?

We'll need to work with every capacity up to the input weight capacity. That's an easy loop:

Java ▾
```java
// every integer from 0 to the input weightCapacity
for (int currentCapacity = 0; currentCapacity <= weightCapacity; currentCapacity++) {
    ...
}
```

What will we do inside this loop? This is where it gets a little tricky.

We care about any cakes that weigh *the current capacity or less*. Let's try putting *each cake* in the bag and seeing how valuable of a haul we could fit from there.

So we'll write a loop through all the cakes (ignoring cakes that are too heavy to fit):

Java ▾
```java
for (CakeType cakeType : cakeTypes) {

    // if the cake weighs as much or less than the current capacity
    // see what our max value could be if we took it!
    if (cakeType.weight <= currentCapacity) {
        // find maxValueUsingCake
        ...
    }
}
```

And put it in our method body so far:

Java

```
public static long maxDuffelBagValue(CakeType[] cakeTypes, int weightCapacity) {

    // we make an array to hold the maximum possible value at every
    // duffel bag weight capacity from 0 to weightCapacity
    // starting each index with value 0
    long[] maxValuesAtCapacities = new long[weightCapacity + 1];

    for (int currentCapacity = 0; currentCapacity <= weightCapacity; currentCapacity++) {

        for (CakeType cakeType : cakeTypes) {

            // if the cake weighs as much or less than the current capacity
            // see what our max value could be if we took it!
            if (cakeType.weight <= currentCapacity) {
                // find maxValueUsingCake
                ...
            }
        }
    }
}
```

How do we compute `maxValueUsingCake`?

Remember when we were calculating the max value at capacity 3kg and we "hard-coded" the `maxValueUsingCake` for cakes that weigh 3 kg, 2kg, and 1kg?

Java

```
// cake weighs 3 kg
long maxValueUsingCake = cakeType.value;

// cake weighs 2 kg
long maxValueUsingCake = maxValueAtCapacity1 + cakeType.value;

// cake weighs 1 kg
long maxValueUsingCake = maxValueAtCapacity2 + cakeType.value;
```

How can we generalize this? With our new method body, look at the variables we have in scope:

1. `maxValuesAtCapacities`
2. `currentCapacity`

3. `cakeType`

Can we use these to get `maxValueUsingCake` for *any cake*?