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 an array with two indices:

0. An integer representing the **weight** of the cake in kilograms
1. An integer representing the **monetary value** of the cake in British pounds

For example:

```ruby
# weighs 7 kilograms and has a value of 160 pounds
[7, 160]

# weighs 3 kilograms and has a value of 90 pounds
[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 function `max_duffel_bag_value()` that takes **an array of cake type arrays** and **a weight capacity**, and returns **the *maximum monetary value* the duffel bag can hold.**

For example:

```ruby
cake_arrays = [[7, 160], [3, 90], [2, 15]]
capacity = 20

max_duffel_bag_value(cake_arrays, 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.

Gotchas

Does your function work if the duffel bag's weight capacity is 0 kg?

Does your function work if any of the cakes weigh 0 kg? Think about a cake whose weight and value are *both* 0.

We can do this in $O(n * k)$ time and $O(k)$ space, where n is the number of types of cakes and k is the duffel bag's capacity!

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:

```ruby
[[1, 30], [50, 200]]
```

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

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:

```ruby
[[1, 30], [50, 200]]
```

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

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

```ruby
[[3, 40], [5, 70]]
```

If our capacity is **8 kg**, no problem. Our algorithm chooses one of each cake, giving us a haul worth **110 pounds**, 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 pounds**. But the *actual optimal value* is **120 pounds**—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

> A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.
>
> The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

approach to keep track of the max value we've seen so far.

Here's an example solution:

```ruby
def max_duffel_bag_value_with_capacity_1(cake_arrays)

    max_value_at_capacity_1 = 0

    cake_arrays.each do |cake_weight, cake_value|
        if cake_weight == 1
            max_value_at_capacity_1 = [max_value_at_capacity_1, cake_value].max
        end
    end

    return max_value_at_capacity_1
end
```

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 `max_value_at_capacity_2`. 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 `max_value_at_capacity_2`.

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 `max_value_at_capacity_2`.

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!

A problem has **overlapping subproblems** if finding its solution involves solving the *same* subproblem multiple times.

As an example, let's look at the fibonacci squence (the series where each number is the sum of the two previous ones—0,1,1,2,3,5,8...).

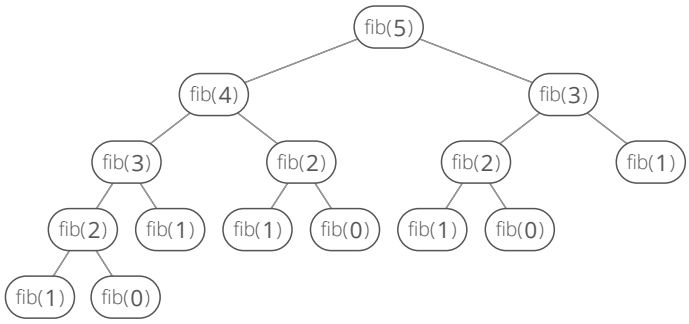
If we wanted to compute the nth fibonacci number, we could use this simple recursive algorithm:

Ruby

```
def fib(n)
    if n == 0 || n == 1
        return n
    end
    return fib(n-1) + fib(n-2)
end
```

We'd call `fib(n-1)` and `fib(n-2)` **subproblems** of `fib(n)`.

Now let's look at what happens when we call `fib(5)`:



Our function ends up recursively calling `fib(2)` ***three times***. So the problem of finding the nth fibonacci number has 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 `max_value_at_capacity_1` and `max_value_at_capacity_2`, now we'll track `max_value_at_capacity_3` and loop through every cake:

```ruby
max_value_at_capacity_3 = 0

cake_arrays.each do |cake_weight, cake_value|
    # only care about cakes that weigh 3 kg or less
    ...
end
```

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 `max_value_at_capacity_3`.

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 `max_value_at_capacity_1`! We can just add that to the current cake's value!

Now we can see which is higher—our *current* `max_value_at_capacity_3`, or the *new* max value if we use the cake:

```ruby
max_value_using_cake    = max_value_at_capacity_1 + cake_value
max_value_at_capacity_3 = [max_value_at_capacity_3, max_value_using_cake].max
```

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

Basically the same as if it weighs 2 kg:

```ruby
max_value_using_cake    = max_value_at_capacity_2 + cake_value
max_value_at_capacity_3 = [max_value_at_capacity_3, max_value_using_cake].max
```

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 function 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, where the keys represent capacities and the values represent the max possible monetary values at those capacities. Hashes are *built on* arrays, so we can save some overhead by just using an array.

```ruby
def max_duffel_bag_value(cake_arrays, weight_capacity)

    # array to hold the maximum possible value at every
    # integer capacity from 0 to weight_capacity
    # starting each index with value 0
    max_values_at_capacities = [0] * (weight_capacity + 1)
end
```

What do we do next?

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

```ruby
# every integer from 0 to the input weight_capacity
(0..weight_capacity).each do |current_capacity|
    ...
end
```

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):

```ruby
cake_arrays.each do |cake_weight, cake_value|

    # if the cake weighs as much or less than the current capacity
    # see what our max value could be if we took it!
    if cake_weight <= current_capacity
        # find max_value_using_cake
        ...
    end
end
```

And put it in our function body so far:

```ruby
def max_duffel_bag_value(cake_arrays, weight_capacity)

    # we make an array to hold the maximum possible value at every
    # duffel bag weight capacity from 0 to weight_capacity
    # starting each index with value 0
    max_values_at_capacities = [0] * (weight_capacity + 1)

    (0..weight_capacity).each do |current_capacity|

        cake_arrays.each do |cake_weight, cake_value|

            # if the cake weighs as much or less than the current capacity
            # see what our max value could be if we took it!
            if cake_weight <= current_capacity
                # find max_value_using_cake
                ...
            end
        end
    end
end
```

How do we compute `max_value_using_cake`?

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

```ruby
# cake weighs 3 kg
max_value_using_cake = cake_value

# cake weighs 2 kg
max_value_using_cake = max_value_at_capacity_1 + cake_value

# cake weighs 1 kg
max_value_using_cake = max_value_at_capacity_2 + cake_value
```

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

1. `max_values_at_capacities`
2. `current_capacity`
3. `cake_weight`
4. `cake_value`

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

Well, let's figure out how much space would be left in the duffel bag after putting the cake in:

```ruby
remaining_capacity_after_taking_cake = current_capacity - cake_weight
```

So `max_value_using_cake` is:

1. the current cake's value, *plus*
2. the best value we can fill the `remaining_capacity_after_taking_cake` with

```ruby
remaining_capacity_after_taking_cake = current_capacity - cake_weight
max_value_using_cake = cake_value + max_values_at_capacities[remaining_capacity_after_takin
```

We can squish this into one line:

```ruby
max_value_using_cake = cake_value + max_values_at_capacities[current_capacity - cake_weight]
```

Since `remaining_capacity_after_taking_cake` is a *lower* capacity, we'll have *always* already computed its max value and stored it in our `max_values_at_capacities`!

Now that we know the max value *if we include the cake*, **should we include it?** How do we know?

Let's allocate a variable `current_max_value` that holds the highest value we can carry at the current capacity. We can start it at zero, and as we go through all the cakes, any time the value *using* a cake is higher than `current_max_value`, we'll update `current_max_value`!

```ruby
current_max_value = [max_value_using_cake, current_max_value].max
```

What do we *do* with each value for `current_max_value`? What do we need to do for each *capacity* when we finish looping through all the cakes?

We save each `current_max_value` in the `max_values_at_capacities` array. We'll also need to make sure we set `current_max_value` to zero in the right place in our loops—we want it to reset every time we start a new capacity.

So here's our function so far:

```ruby
def max_duffel_bag_value(cake_arrays, weight_capacity)

    # we make an array to hold the maximum possible value at every
    # duffel bag weight capacity from 0 to weight_capacity
    # starting each index with value 0
    max_values_at_capacities = [0] * (weight_capacity + 1)

    (0..weight_capacity).each do |current_capacity|

        # set a variable to hold the max monetary value so far for current_capacity
        current_max_value = 0

        cake_arrays.each do |cake_weight, cake_value|

            # if the current cake weighs as much or less than the current weight capacity
            # it's possible taking the cake would give get a better value
            if (cake_weight <= current_capacity)

                # so we check: should we use the cake or not?
                # if we use the cake, the most kilograms we can include in addition to the
                # we're adding is the current capacity minus the cake's weight. we find the
                # value at that integer capacity in our array max_values_at_capacities
                max_value_using_cake = cake_value + max_values_at_capacities[current_capaci

                # now we see if it's worth taking the cake. how does the
                # value with the cake compare to the current_max_value?
                current_max_value = [max_value_using_cake, current_max_value].max
            end
        end

        # add each capacity's max value to our array so we can use them
        # when calculating all the remaining capacities
        max_values_at_capacities[current_capacity] = current_max_value
    end
end
```

Looking good! But **what's our final answer?**

Our final answer is `max_values_at_capacities[weight_capacity]`!

Okay, this seems complete. **What about edge cases?**

Remember, weights and values can be any non-negative integer. What about zeroes? How can we handle duffel bags that can't hold anything and cakes that weigh nothing?

Well, if our duffel bag can't hold anything, we can just return 0. And if a cake weighs 0 kg, we return *infinity*. Right?

Not that simple!

What if our duffel bag holds 0 kg, and we have a cake that weighs 0 kg. What do we return?

And what if we have a cake that weighs 0 kg, but its value is *also* 0. If we have other cakes with positive weights and values, what do we return?

If a cake's weight and value are both 0, it's reasonable to not have that cake affect what we return at all.

If we have a cake that weighs 0 kg and has a positive value, it's reasonable to return infinity, even if the capacity is 0.

For returning infinity, we have several choices. We could return:

1. **Ruby's `Float::INFINITY`.**
2. Return a **custom response**, like the string `'infinity'`.
3. **Raise an exception** indicating the answer is infinity.

What are the advantages and disadvantages of each option?

For the **first option** the advantage is we get the *behavior* of infinity. Compared to any other integer, `Float::INFINITY` will be greater. And it's a *number*, which can be an advantage or disadvantage—we might want our result to always be the same *type*, but without manually checking we won't know if we mean an actual value or the special case of infinity.

For the **second option** the advantage is we can create a custom behavior that we—or our function's users—could know to expect. The disadvantage is we'd *have* to explicitly check for that behavior, otherwise we might end up trying to parse the string "infinity" as an integer, which could give us an error or (perhaps worse) a random number. In a production system, a function that sometimes returns an integer and sometimes returns a string would probably be seen as sloppy.

The **third option** is a good choice if we decide infinity is usually an "unacceptable" answer. For example, we might decide an infinite answer means we've probably entered our inputs wrong. Then, if we *really* wanted to "accept" an infinite answer, we could always "catch" this exception when we call our function.

Any option *could* be reasonable. We'll go with the first one here.

Solution

This is a classic computer science puzzle called **"the unbounded knapsack problem."**

We use a bottom-up⁊

> Going **bottom-up** is a way to avoid recursion, saving the **memory cost** that recursion incurs when it builds up the **call stack**.
>
> Put simply, a bottom-up algorithm "starts from the beginning," while a recursive algorithm often "starts from the end and works backwards."
>
> For example, if we wanted to multiply all the numbers in the range $1..n$, we could use this cute, **top-down**, recursive one-liner:

```ruby
def product_1_to_n(n)
    # we assume n >= 1
    return n > 1 ? n * product_1_to_n(n-1) : 1
end
```

This approach has a problem: it builds up a **call stack** of size $O(n)$, which makes our total memory cost $O(n)$. This makes it vulnerable to a **stack overflow error**, where the call stack gets too big and runs out of space.

To avoid this, we can instead go **bottom-up**:

```ruby
def product_1_to_n(n)
    # we assume n >= 1

    result = 1
    (1..n).each do |num|
        result *= num
    end

    return result
end
```

This approach uses $O(1)$ space ($O(n)$ time).

> *Some* compilers and interpreters will do what's called **tail call optimization** (TCO), where it can optimize *some* recursive functions to avoid building up a tall call stack. Python and Java decidedly do not use TCO. Some Ruby implementations do, but most don't. Some C implementations do, and the JavaScript spec recently *allowed* TCO. Scheme is one of the few languages that *guarantee* TCO in all implementations. In general, best not to assume your compiler/interpreter will do this work for you.

Going bottom-up is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **memoization (/concept/memoization)**.

approach to find the max value at our duffel bag's `weight_capacity` by finding the max value at *every* capacity from 0 to `weight_capacity`.

We allocate an array `max_values_at_capacities` where the indices are capacities and each value is the max value *at that capacity*.

For each capacity, we want to know the max monetary value we can carry. To figure that out, we go through each cake, checking to see if we should take that cake.

The best monetary value we can get if we take a given cake is simply:

1. that cake's value, plus
2. the best monetary value we can carry in our remaining duffel bag capacity after taking the cake—which we'll already have stored in `max_values_at_capacities`

To handle weights and values of zero, we return infinity *only* if a cake weighs nothing and has a positive value.

```ruby
def max_duffel_bag_value(cake_arrays, weight_capacity)

    # we make an array to hold the maximum possible value at every
    # duffel bag weight capacity from 0 to weight_capacity
    # starting each index with value 0
    max_values_at_capacities = [0] * (weight_capacity + 1)

    (0..weight_capacity).each do |current_capacity|

        # set a variable to hold the max monetary value so far for current_capacity
        current_max_value = 0

        cake_arrays.each do |cake_weight, cake_value|

            # if a cake weighs 0 and has a positive value the value of our duffel bag is infinite!
            if (cake_weight == 0 && cake_value != 0)
                return Float::INFINITY
            end

            # if the current cake weighs as much or less than the current weight capacity
            # it's possible taking the cake would give get a better value
            if (cake_weight <= current_capacity)

                # so we check: should we use the cake or not?
                # if we use the cake, the most kilograms we can include in addition to the cake
                # we're adding is the current capacity minus the cake's weight. we find the max
                # value at that integer capacity in our array max_values_at_capacities
                max_value_using_cake = cake_value + max_values_at_capacities[current_capacity - cake_weight]

                # now we see if it's worth taking the cake. how does the
                # value with the cake compare to the current_max_value?
                current_max_value = [max_value_using_cake, current_max_value].max
            end
        end

        # add each capacity's max value to our array so we can use them
        # when calculating all the remaining capacities
        max_values_at_capacities[current_capacity] = current_max_value
    end

    return max_values_at_capacities[weight_capacity]
end
```

Complexity

$O(n * k)$ time, and $O(k)$ space, where n is number of types of cake and k is the capacity of the duffel bag. We loop through each cake (n cakes) for every capacity (k capacities), so our runtime is $O(n * k)$, and maintaining the array of $k + 1$ capacities gives us the $O(k)$ space.

Congratulations! Because of dynamic programming, you have successfully stolen the Queen's cakes and made it big.

Keep in mind: in some cases, it might *not* be worth using our optimal dynamic programming solution. It's a pretty slow algorithm—without any context (not knowing how many cake types we have, what our weight capacity is, or just how they compare) it's easy to see $O(n * k)$ potentially being as bad as $O(n^2)$ if n is close to k.

If we cared about *time*, like if there was an alarm in the vault and we had to move quickly, it might be worth using a *faster algorithm that gives us a **good** answer, even if it's not always the **optimal** answer*. Some of our first ideas in the breakdown were to look at cake values or value/weight ratios. Those algorithms would probably be faster, taking $O(n \lg n)$ time (we'd have to start by sorting the input).

Sometimes an efficient, *good* answer might be more *practical* than an inefficient, *optimal* answer.

Bonus

1. We know the *max value we can carry*, but **which cakes should we take, and how many?** Try adjusting your answer to return this information as well.
2. What if we check to see if all the cake weights have a **common denominator**? Can we improve our algorithm?
3. A cake that's both *heavier* and *worth less* than another cake would *never* be in the optimal solution. This idea is called **dominance relations**. Can you apply this idea to save some time? Hint: dominance relations can apply to *sets of cakes*, not just individual cakes.
4. What if we had an array for *every individual cake* instead of *types of cakes*? So now there's not an unlimited supply of a type of cake—there's exactly one of each. This is a *similar but harder* problem, known as the **0/1 Knapsack** problem.

What We Learned

This question is our spin on the famous "unbounded knapsack problem"—a classic dynamic programming question.

If you're struggling with dynamic programming, we have reference pages for the two main dynamic programming strategies: memoization (https://www.interviewcake.com/concept/memoization) and going bottom-up (https://www.interviewcake.com/concept/bottom-up).