**8. Linked Lists**

**Write a method for reversing a linked list. ↴ Do it in place. ↴**

Your method will have one input: the head of the list.

Your method should return the new head of the list.

Here's a sample linked list node class:

public class LinkedListNode{

public int Value { get; set; }

public LinkedListNode Next { get; set; }

public LinkedListNode(int value) {

Value = value;

}

}

**Gotchas**

We can do this in O(1)*O*(1) space. So don't make a new list; use the existing list nodes!

We can do this is in O(n)*O*(*n*) time.

Careful—even the right *approach* will fail if done in the wrong *order*.

Try drawing a picture of a small linked list and running your method by hand. Does it actually work?

The most obvious edge cases are:

1. the list has 0 elements
2. the list has 1 element

Does your method correctly handle those cases?

**Breakdown**

Our first thought might be to build our reversed list "from the beginning," starting with the head of the final *reversed* linked list.

The head of the reversed list will be the *tail* of the input list. To get to that node we'll have to walk through the whole list once (O(n)*O*(*n*) time). And that's just to get started.

That seems inefficient. **Can we reverse the list while making just one walk from head to tail of the input list?**

We can reverse the list by changing the Next pointer of each node. Where should each node's Next pointer...point?

Each node's Next pointer should point to the *previous* node.

How can we move each node's Next pointer to its *previous* node in one pass from head to tail of our current list?

**Solution**

In one pass from head to tail of our input list, we point each node's Next pointer to the previous item.

**The order of operations is important here!** We're careful to copy currentNode.Next into Next *before* setting currentNode.Next to previousNode. Otherwise "stepping forward" at the end could actually mean stepping *back* to previousNode!

public static LinkedListNode Reverse(LinkedListNode headOfList){

LinkedListNode currentNode = headOfList;

LinkedListNode previousNode = null;

LinkedListNode nextNode = null;

// Until we have 'fallen off' the end of the list

while (currentNode != null) {

// Copy a pointer to the next element

// before we overwrite currentNode.Next

nextNode = currentNode.Next;

// Reverse the 'Next' pointer

currentNode.Next = previousNode;

// Step forward in the list

previousNode = currentNode;

currentNode = nextNode;

}

return previousNode;

}

We return previousNode because when we exit the list, currentNode is null. Which means that the last node we visited—previousNode—was the tail of the *original* list, and thus the head of our *reversed* list.

**Complexity**

O(n)*O*(*n*) time and O(1)*O*(1) space. We pass over the list only once, and maintain a constant number of variables in memory.

**Bonus**

This in-place ↴ reversal destroys the input linked list. What if we wanted to keep a copy of the original linked list? Write a method for reversing a linked list out-of-place.

**What We Learned**

It's one of those problems where, even once you know the procedure, it's hard to write a bug-free solution. Drawing it out helps a lot. Write out a sample linked list and walk through your code by hand, step by step, running each operation on your sample input to see if the final output

**4. Greedy Algs**

A greedy algorithm builds up a solution by choosing the option that looks the best at every step.

Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?

Whenever picking which coin to use, you'd take the highest-value coin you could. A quarter, another quarter, then a dime, a nickel, and finally two pennies. That's a *greedy* algorithm, because you're always *greedily* choosing the coin that covers the biggest portion of the remaining amount.

Some other places where a greedy algorithm gets you the best solution:

* Trying to fit as many overlapping meetings as possible in a conference room? At each step, schedule the meeting that *ends* earliest.
* Looking for a minimum spanning tree in a [graph](https://www.interviewcake.com/concept/graph)? At each step, greedily pick the cheapest edge that reaches a new vertex.

Careful: sometimes a greedy algorithm ***doesn't*** give you an optimal solution:

* [When filling a duffel bag with cakes of different weights and values](https://www.interviewcake.com/question/cake-thief), choosing the cake with the highest value per pound doesn't always produce the best haul.
* To find the cheapest route visiting a set of cities, choosing to visit the cheapest city you haven't been to yet doesn't produce the cheapest overall itinerary.

Validating that a greedy strategy always gets the best answer is tricky. Either prove that the answer produced by the greedy algorithm is as good as an optimal answer, or run through a rigorous set of test cases to convince your interviewer (and yourself) that its correct.

**Trading Apple stocks all day instead.**

First, I wanna know how much money I *could have* made yesterday if I'd been trading Apple stocks all day.

So I grabbed Apple's stock prices from yesterday and put them in an array called stockPrices, where:

* The **indices** are the time (in minutes) past trade opening time, which was 9:30am local time.
* The **values** are the price (in US dollars) of one share of Apple stock at that time.

So if the stock cost $500 at 10:30am, that means stockPrices[60] = 500.

Write an efficient method that takes stockPrices and returns **the best profit I could have made from one purchase and one sale of one share of Apple stock yesterday.**

For example:

int[] stockPrices = { 10, 7, 5, 8, 11, 9 };

// Returns 6 (buying for $5 and selling for $11)

GetMaxProfit(stockPrices);

No "shorting"—you need to buy before you can sell. Also, you can't buy *and* sell in the same time step—at least 1 minute has to pass.

**Gotchas**

**You can't just take the difference between the highest price and the lowest price**, because the highest price might come *before* the lowest price. And you have to buy before you can sell.

What if the price *goes down all day*? In that case, the best profit will be **negative**.

You can do this in O(n)*O*(*n*) time and O(1)*O*(1) space! ↴

**Breakdown**

To start, try writing an example value for stockPrices and finding the maximum profit "by hand." What's your process for figuring out the maximum profit?

The brute force ↴ approach would be to try *every pair of times* (treating the earlier time as the buy time and the later time as the sell time) and see which one is higher.

using System;

public int GetMaxProfit(int[] stockPrices){

int maxProfit = 0;

// Go through every time

for (int outerTime = 0; outerTime < stockPrices.Length; outerTime++) {

// For every time, go through every other time

for (int innerTime = 0; innerTime < stockPrices.Length; innerTime++) {

// For each pair, find the earlier and later times

int earlierTime = Math.Min(outerTime, innerTime);

int laterTime = Math.Max(outerTime, innerTime);

// And use those to find the earlier and later prices

int earlierPrice = stockPrices[earlierTime];

int laterPrice = stockPrices[laterTime];

// See what our profit would be if we bought at the

// min price and sold at the current price

int potentialProfit = laterPrice - earlierPrice;

// Update maxProfit if we can do better

maxProfit = Math.Max(maxProfit, potentialProfit);

}

}

return maxProfit;

}

But that will take O(n^2)*O*(*n*2) time, ↴ since we have two nested loops—for *every* time, we're going through *every other* time. Also, **it's not correct**: we won't ever report a negative profit! Can we do better?

Well, we’re doing a lot of extra work. We’re looking at every pair *twice*. We know we have to buy before we sell, so in our *inner for loop* we could just look at every price **after** the price in our *outer for loop*.

That could look like this:

using System;

public int GetMaxProfit(int[] stockPrices){

int maxProfit = 0;

// Go through every price and time

for (int earlierTime = 0; earlierTime < stockPrices.Length; earlierTime++) {

int earlierPrice = stockPrices[earlierTime];

// And go through all the LATER prices

for (int laterTime = earlierTime + 1; laterTime < stockPrices.Length; laterTime++) {

int laterPrice = stockPrices[laterTime];

// See what our profit would be if we bought at the

// min price and sold at the current price

int potentialProfit = laterPrice - earlierPrice;

// Update maxProfit if we can do better

maxProfit = Math.Max(maxProfit, potentialProfit);

}

}

return maxProfit;

}

**What’s our runtime now?**

Well, our outer for loop goes through *all* the times and prices, but our inner for loop goes through *one fewer price each time*. So our total number of steps is the sum n + (n - 1) + (n - 2) ... + 2 + 1*n*+(*n*−1)+(*n*−2)...+2+1 ↴ , which is *still* O(n^2)*O*(*n*2) time.

We can do better!

If we're going to do better than O(n^2)*O*(*n*2), we're probably going to do it in either O(n\lg{n})*O*(*n*lg*n*) or O(n)*O*(*n*). O(n\lg{n})*O*(*n*lg*n*) comes up in sorting and searching algorithms where we're recursively cutting the array in half. It's not obvious that we can save time by cutting the array in half here. Let's first see how well we can do by looping through the array only *once*.

Since we're trying to loop through the array once, let's use a greedy ↴ approach, where we keep a running maxProfit until we reach the end. We'll start our maxProfit at $0. As we're iterating, how do we know if we've found a new maxProfit?

At each iteration, our maxProfit is either:

1. the same as the maxProfit at the last time step, or
2. the max profit we can get by selling at the currentPrice

How do we know when we have case (2)?

The max profit we can get by selling at the currentPrice is simply the difference between the currentPrice and the minPrice from earlier in the day. If this difference is greater than the current maxProfit, we have a new maxProfit.

So for every price, we’ll need to:

* keep track of the **lowest price we’ve seen so far**
* see if we can get a **better profit**

Here’s one possible solution:

using System;

public int GetMaxProfit(int[] stockPrices){

int minPrice = stockPrices[0];

int maxProfit = 0;

foreach (int currentPrice in stockPrices) {

// Ensure minPrice is the lowest price we've seen so far

minPrice = Math.Min(minPrice, currentPrice);

// See what our profit would be if we bought at the

// min price and sold at the current price

int potentialProfit = currentPrice - minPrice;

// Update maxProfit if we can do better

maxProfit = Math.Max(maxProfit, potentialProfit);

}

return maxProfit;

}

We’re finding the max profit with one pass and constant space!

**Are we done?** Let’s think about some edge cases. What if the price *stays the same*? What if the price *goes down all day*?

If the price doesn't change, the max possible profit is 0. Our method will correctly return that. So we're good.

But if the value *goes down all day*, we’re in trouble. Our method would return 0, but there’s no way we could break even if the price always goes down.

**How can we handle this?**

Well, what are our options? Leaving our method as it is and just returning zero is *not* a reasonable option—we wouldn't know if our best profit was negative or *actually* zero, so we'd be losing information. Two reasonable options could be:

1. **return a negative profit**. “What’s the least badly we could have done?”
2. **throw an exception**. “We should not have purchased stocks yesterday!”

In this case, it’s probably best to go with option (1). The advantages of returning a negative profit are:

* We **more accurately answer the challenge**. If profit is "revenue minus expenses", we’re returning the *best* we could have done.
* It's **less opinionated**. We'll leave decisions up to our method’s users. It would be easy to wrap our method in a helper method to decide if it’s worth making a purchase.
* We allow ourselves to **collect better data**. It *matters* if we would have lost money, and it *matters* how much we would have lost. If we’re trying to get rich, we’ll probably care about those numbers.

**How can we adjust our method to return a negative profit if we can only lose money?** Initializing maxProfit to 0 won’t work...

Well, we started our minPrice at the first price, so let’s start our maxProfit at the *first profit we could get*—if we buy at the first time and sell at the second time.

minPrice = stockPrices[0];

maxProfit = stockPrices[1] - stockPrices[0];

But we have the potential for an IndexOutOfRangeException here, if stockPrices has fewer than 2 prices.

We *do* want to throw an exception in that case, since *profit* requires buying *and* selling, which we can't do with less than 2 prices. So, let's explicitly check for this case and handle it:

if (stockPrices.Length < 2){

throw new ArgumentException("Getting a profit requires at least 2 prices", nameof(stockPrices));

}

int minPrice = stockPrices[0];

int maxProfit = stockPrices[1] - stockPrices[0];

Ok, does that work?

No! **maxProfit is still always 0.** What’s happening?

If the price always goes down, minPrice is always set to the currentPrice. So currentPrice - minPrice comes out to 0, which of course will always be greater than a negative profit.

When we’re calculating the maxProfit, we need to make sure we never have a case where we try **both buying and selling stocks at the currentPrice**.

To make sure we’re always buying at an *earlier* price, never the currentPrice, let’s switch the order around so we calculate maxProfit *before* we update minPrice.

We'll also need to pay special attention to time 0. Make sure we don't try to buy *and* sell at time 0.

**Solution**

We’ll greedily ↴ walk through the array to track the max profit and lowest price so far.

For every price, we check if:

* we can get a better profit by buying at minPrice and selling at the currentPrice
* we have a new minPrice

To start, we initialize:

1. minPrice as the first price of the day
2. maxProfit as the first profit we could get

We decided to return a *negative* profit if the price decreases all day and we can't make any money. We could have thrown an exception instead, but returning the negative profit is cleaner, makes our method less opinionated, and ensures we don't lose information.

using System;

public static int GetMaxProfit(int[] stockPrices){

if (stockPrices.Length < 2) {

throw new ArgumentException("Getting a profit requires at least 2 prices", nameof(stockPrices));

}

// We'll greedily update minPrice and maxProfit, so we initialize

// them to the first price and the first possible profit

int minPrice = stockPrices[0];

int maxProfit = stockPrices[1] - stockPrices[0];

// Start at the second (index 1) time.

// We can't sell at the first time, since we must buy first,

// and we can't buy and sell at the same time!

// If we started at index 0, we'd try to buy \*and\* sell at time 0.

// This would give a profit of 0, which is a problem if our

// maxProfit is supposed to be \*negative\*--we'd return 0.

for (int i = 1; i < stockPrices.Length; i++) {

int currentPrice = stockPrices[i];

// See what our profit would be if we bought at the

// min price and sold at the current price

int potentialProfit = currentPrice - minPrice;

// Update maxProfit if we can do better

maxProfit = Math.Max(maxProfit, potentialProfit);

// Update minPrice so it's always

// the lowest price we've seen so far

minPrice = Math.Min(minPrice, currentPrice);

}

return maxProfit;

}

**Complexity**

O(n)*O*(*n*) time and O(1)*O*(1) space. ↴ We only loop through the array once.

**What We Learned**

This one's a good example of the greedy ↴ approach in action. Greedy approaches are great because they're *fast* (usually just one pass through the input). But they don't work for every problem.

How do you know if a problem will lend itself to a greedy approach? Best bet is to try it out and see if it works. Trying out a greedy approach should be one of the first ways you try to break down a new question.

To try it on a new problem, start by asking yourself:

"Suppose we *could* come up with the answer in one pass through the input, by simply updating the 'best answer so far' as we went. What ***additional values*** would we need to keep updated as we looked at each item in our input, in order to be able to update the **'best answer so far'** in constant time?"

In *this* case:

The "**best answer so far**" is, of course, the max profit that we can get based on the prices we've seen so far.

The "**additional value**" is the minimum price we've seen so far. If we keep that updated, we can use it to calculate the new max profit so far in constant time. The max profit is the larger of:

1. The previous max profit
2. The max profit we can get by selling now (the current price minus the minimum price seen so far)

Try applying this greedy methodology to future questions.

**Given an array of integers, find the highest product you can get from three of the integers.**

**The input arrayOfInts will always have at least three integers.**

Gotchas

Does your method work with negative numbers? If arrayOfInts is [-10, -10, 1, 3, 2][−10,−10,1,3,2] we should return 300300 (which we get by taking -10 \* -10 \* 3−10∗−10∗3).

We can do this in O(n)*O*(*n*) time and O(1)*O*(1) space.

Breakdown

To brute force ↴ an answer we could iterate through arrayOfInts and multiply each integer by each *other* integer, and then multiply that product by each *other other* integer. This would probably involve nesting 3 loops. But that would be an O(n^3)*O*(*n*3) runtime! We can *definitely* do better than that.

Because any integer in the array could potentially be part of the greatest product of three integers, we must at least *look* at *each integer*. So we're doomed to spend at least O(n)*O*(*n*) time.

Sorting the array would let us grab the highest numbers quickly, so it might be a good first step. Sorting takes O(n\lg{n})*O*(*n*lg*n*) time. That's better than the O(n^3)*O*(*n*3) time our brute force approach required, but we can still do better.

Since we know we must spend *at least* O(n)*O*(*n*) time, let's see if we can solve it in *exactly* O(n)*O*(*n*) time.

A great way to get O(n)*O*(*n*) runtime is to use a greedy ↴ approach. How can we keep track of the highestProductOf3 "so far" as we do one walk through the array?

Put differently, for each new current number during our iteration, how do we know if it gives us a new highestProductOf3?

We have a new highestProductOf3 if the current number times two other numbers gives a product that's higher than our current highestProductOf3. What must we keep track of at each step so that we know if the current number times two other numbers gives us a new highestProductOf3?

Our first guess might be:

1. our current highestProductOf3
2. the threeNumbersWhichGiveHighestProduct

But consider this example:

int[] arrayOfInts = { 1, 10, -5, 1, -100 };

Right before we hit -100−100 (so, in our second-to-last iteration), our highestProductOf3 was 1010, and the threeNumbersWhichGiveHighestProduct were [10,1,1][10,1,1]. But once we hit -100−100, suddenly we can take -100 \* -5 \* 10−100∗−5∗10 to get 50005000. So we should have "held on to" that -5−5, even though it wasn't one of the threeNumbersWhichGiveHighestProduct.

We need something a little smarter than threeNumbersWhichGiveHighestProduct. What should we keep track of to make sure we can handle a case like this?

There are at least two great answers:

1. Keep track of the highest2 and lowest2 (most negative) numbers. If the current number times *some combination of those* is higher than the current highestProductOf3, we have a new highestProductOf3!
2. Keep track of the highestProductOf2 and lowestProductOf2 (could be a low negative number). If the current number times one of those is higher than the current highestProductOf3, we have a new highestProductOf3!

We'll go with (2). It ends up being *slightly* cleaner than (1), though they both work just fine.

How do we keep track of the highestProductOf2 and lowestProductOf2 at each iteration? (Hint: we may need to also keep track of *something else*.)

We also keep track of the lowest number and highest number. If the current number times the current highest—*or the current lowest*, if current is negative—is greater than the current highestProductOf2, we have a new highestProductOf2. Same for lowestProductOf2.

So at each iteration we're keeping track of and updating:

* highestProductOf3
* highestProductOf2
* highest
* lowestProductOf2
* lowest

Can you implement this in code? Careful—make sure you update each of these variables in the right order, otherwise you might end up e.g. multiplying the current number by itself to get a new highestProductOf2.

Solution

We use a greedy ↴ approach to solve the problem in one pass. At each iteration we keep track of:

* highestProductOf3
* highestProductOf2
* highest
* lowestProductOf2
* lowest

When we reach the end, the highestProductOf3 is our answer. We maintain the others because they're necessary for keeping the highestProductOf3 up to date as we walk through the array. At each iteration, the highestProductOf3 is the highest of:

1. the current highestProductOf3
2. current \* highestProductOf2
3. current \* lowestProductOf2 (if current and lowestProductOf2 are both low negative numbers, this product is a high positive number).

using System;

public static int HighestProductOf3(int[] arrayOfInts){

if (arrayOfInts.Length < 3) {

throw new ArgumentException("Less than 3 items!", nameof(arrayOfInts));

}

// We're going to start at the 3rd item (at index 2)

// so pre-populate highests and lowests based on the first 2 items.

// We could also start these as null and check below if they're set

// but this is arguably cleaner

int highest = Math.Max(arrayOfInts[0], arrayOfInts[1]);

int lowest = Math.Min(arrayOfInts[0], arrayOfInts[1]);

int highestProductOf2 = arrayOfInts[0] \* arrayOfInts[1];

int lowestProductOf2 = arrayOfInts[0] \* arrayOfInts[1];

// Except this one--we pre-populate it for the first \*3\* items.

// This means in our first pass it'll check against itself, which is fine.

int highestProductOf3 = arrayOfInts[0] \* arrayOfInts[1] \* arrayOfInts[2];

// Walk through items, starting at index 2

for (int i = 2; i < arrayOfInts.Length; i++) {

int current = arrayOfInts[i];

// Do we have a new highest product of 3?

// It's either the current highest,

// or the current times the highest product of two

// or the current times the lowest product of two

highestProductOf3 = Math.Max(Math.Max(

highestProductOf3,

current \* highestProductOf2),

current \* lowestProductOf2);

// Do we have a new highest product of two?

highestProductOf2 = Math.Max(Math.Max(

highestProductOf2,

current \* highest),

current \* lowest);

// Do we have a new lowest product of two?

lowestProductOf2 = Math.Min(Math.Min(

lowestProductOf2,

current \* highest),

current \* lowest);

// Do we have a new highest?

highest = Math.Max(highest, current);

// Do we have a new lowest?

lowest = Math.Min(lowest, current);

}

return highestProductOf3;

}

Complexity

O(n)*O*(*n*) time and O(1)*O*(1) additional space.

Bonus

1. What if we wanted the highest product of 4 items?
2. What if we wanted the highest product of k*k* items?
3. If our highest product is really big, it could overflow. ↴ How should we protect against this?

What We Learned

Greedy ↴ algorithms in action again!

That's not a coincidence—to illustrate how one pattern can be used to break down several different questions, we're showing this one pattern in action on several different questions.

Usually it takes seeing an algorithmic idea from a few different angles for it to really make intuitive sense.

Our goal here is to teach you the right *way of thinking* to be able to break down problems you haven't seen before. Greedy algorithm design is a big part of that *way of thinking*.

For this one, we built up our greedy algorithm exactly the same way we did for the [Apple stocks](https://www.interviewcake.com/question/stock-price) question. By asking ourselves:

"Suppose we *could* come up with the answer in one pass through the input, by simply updating the 'best answer so far' as we went. What *additional values* would we need to keep updated as we looked at each item in our set, in order to be able to update the 'best answer so far' in constant time?"

For the Apple stocks question, the only "additional value" we needed was the min price so far.

For this one, we needed *four* things in order to calculate the new highestProductOf3 at each step:

* highestProductOf2
* highest
* lowestProductOf2
* lowest

**cake shop**

My cake shop is so popular, I'm adding some tables and hiring wait staff so folks can have a cute sit-down cake-eating experience.

I have two registers: one for take-out orders, and the other for the other folks eating inside the cafe. All the customer orders get combined into one list for the kitchen, where they should be handled first-come, first-served.

Recently, some customers have been complaining that people who placed orders after them are getting their food first. Yikes—that's not good for business!

To investigate their claims, one afternoon I sat behind the registers with my laptop and recorded:

* The take-out orders as they were entered into the system and given to the kitchen. (takeOutOrders)
* The dine-in orders as they were entered into the system and given to the kitchen. (dineInOrders)
* Each customer order (from either register) as it was finished by the kitchen. (servedOrders)

Given all three arrays, write a method to check that my service is first-come, first-served. All food should come out in the same order customers requested it.

We'll represent each customer order as a unique integer.

As an example,

Take Out Orders: [1, 3, 5]

Dine In Orders: [2, 4, 6]

Served Orders: [1, 2, 4, 6, 5, 3]

would *not* be first-come, first-served, since order 3 was requested before order 5 but order 5 was served first.

But,

Take Out Orders: [17, 8, 24]

Dine In Orders: [12, 19, 2]

Served Orders: [17, 8, 12, 19, 24, 2]

*would* be first-come, first-served.

Note: Order numbers are arbitrary. They do not have to be in increasing order.

Gotchas

Watch out for index out of bounds errors! Will your method ever try to grab the 0th item from an empty array, or the n^{th}*nth* item from an array with n*n* elements (where the last index would be n-1*n*−1)?

We can do this in O(n)*O*(*n*) time and O(1)*O*(1) additional space.

Did you come up with a recursive solution? Keep in mind that you may be incurring a hidden space cost (probably O(n)*O*(*n*)) in the call stack! ↴ You can avoid this using an iterative approach.

Breakdown

How can we re-phrase this problem in terms of smaller subproblems?

Breaking the problem into smaller subproblems will clearly involve reducing the size of at least one of our lists of customer order numbers. So to start, let's try taking the first customer order out of servedOrders.

What should be true of this customer order number if my service is first-come, first-served?

If my cake cafe is first-come, first-served, then the first customer order finished (first item in servedOrders) has to either be the first take-out order entered into the system (takeOutOrders[0]) or the first dine-in order entered into the system (dineInOrders[0]).

Once we can check the *first* customer order, how can we verify the remaining ones?

Let's "throw out" the first customer order from servedOrders as well as the customer order it matched with from the beginning of takeOutOrders or dineInOrders. That customer order is now "accounted for."

Now we're left with a smaller version of the original problem, which we can solve using the same approach! So we keep doing this over and over until we exhaust servedOrders. If we get to the end and every customer order "checks out," we return true.

How do we implement this in code?

Now that we have a problem that's the same as the original problem except smaller, our first thought might be to use recursion. All we need is a base case. ↴ What's our base case?

We stop when we run out of customer orders in our servedOrders. So that's our base case: when we've checked all customer orders in servedOrders, we return true because we know all of the customer orders have been "accounted for."

public static int[] RemoveFirstOrder(int[] orders){

int[] result = new int[orders.Length - 1];

if (result.Length > 0)

Array.Copy(orders, 1, result, 0, result.Length);

return result;

}

public static bool IsFirstComeFirstServed(int[] takeOutOrders, int[] dineInOrders, int[] servedOrders){

// Base case

if (servedOrders.Length == 0)

return true;

// If the first order in servedOrders is the same as the

// first order in takeOutOrders

// (making sure first that we have an order in takeOutOrders)

if (takeOutOrders.Length > 0 && takeOutOrders[0] == servedOrders[0]) {

// Take the first order off takeOutOrders and servedOrders and recurse

return IsFirstComeFirstServed(RemoveFirstOrder(takeOutOrders), dineInOrders, RemoveFirstOrder(servedOrders));

}

// If the first order in servedOrders is the same as the first

// in dineInOrders

if (dineInOrders.Length > 0 && dineInOrders[0] == servedOrders[0]) {

// Take the first order off dineInOrders and servedOrders and recurse

return IsFirstComeFirstServed(takeOutOrders, RemoveFirstOrder(dineInOrders), RemoveFirstOrder(servedOrders));

}

// First order in servedOrders doesn't match the first in

// takeOutOrders or dineInOrders, so we know it's not first-come, first-served

return false;

}

C#

This solution will work. But we can do better.

Before we talk about optimization, note that our inputs are probably pretty small. This method will take hardly any time or space, even if it *could be* more efficient. In industry, especially at small startups that want to move quickly, optimizing this might be considered a "premature optimization." Great engineers have both the *skill* to see how to optimize their code and the *wisdom* to know when those optimizations aren't worth it. At this point in the interview I recommend saying, "I think we can optimize this a bit further, although given the nature of the input this probably won't be very resource-intensive anyway...should we talk about optimizations?"

Suppose we *do* want to optimize further. What are the time and space costs to beat? This method will take O(n^2)*O*(*n*2) time and O(n^2)*O*(*n*2) additional space.

Whaaaaat? Yeah. Take a look at this snippet:

public int[] RemoveFirstOrder(int[] orders){

int[] result = new int[orders.Length - 1];

if (result.Length > 0)

Array.Copy(orders, 1, result, 0, result.Length);

return result;

}

return IsFirstComeFirstServed(RemoveFirstOrder(takeOutOrders), dineInOrders, RemoveFirstOrder(servedOrders));

C#

In particular this expression:

int[] result = new int[orders.Length - 1];

if (result.Length > 0)

Array.Copy(orders, 1, result, 0, result.Length);

return result;

C#

That's a slice, ↴ and it costs O(m)*O*(*m*) time and space, where m*m* is the size of the resulting array. That's going to determine our overall time and space cost here—the rest of the work we're doing is constant space and time.

In our recursing we'll build up n*n* frames on the call stack. ↴ Each of those frames will hold a *different slice* of our original servedOrders (and takeOutOrders and dineInOrders, though we only slice one of them in each recursive call).

So, what's the total time and space cost of all our slices?

If servedOrders has n*n* items to start, taking our first slice takes n-1*n*−1 time and space (plus half that, since we're also slicing one of our halves—but that's just a constant multiplier so we can ignore it). In our second recursive call, slicing takes n-2*n*−2 time and space. Etc.

So our total time and space cost for slicing comes to:

(n - 1) + (n - 2) + ... + 2 + 1(*n*−1)+(*n*−2)+...+2+1

This is a common series ↴ that turns out to be O(n^2)*O*(*n*2).

We can do better than this O(n^2)*O*(*n*2) time and space cost. One way we could to that is to avoid slicing and instead keep track of indices in the array:

public static bool IsFirstComeFirstServed(int[] takeOutOrders, int[] dineInOrders,

int[] servedOrders, int servedOrdersIndex = 0,

int takeOutOrdersIndex = 0, int dineInOrdersIndex = 0){

if (servedOrdersIndex == servedOrders.Length) {

// Base case we've hit the end of servedOrders

return true;

}

if (takeOutOrdersIndex < takeOutOrders.Length

&& takeOutOrders[takeOutOrdersIndex] == servedOrders[servedOrdersIndex]) {

// If we still have orders in takeOutOrders

// and the current order in takeOutOrders is the same

// as the current order in servedOrders

takeOutOrdersIndex++;

}

else if (dineInOrdersIndex < dineInOrders.Length

&& dineInOrders[dineInOrdersIndex] == servedOrders[servedOrdersIndex]) {

// If we still have orders in dineInOrders

// and the current order in dineInOrders is the same

// as the current order in servedOrders

dineInOrdersIndex++;

}

else {

// If the current order in servedOrders doesn't match

// the current order in takeOutOrders or dineInOrders, then we're not

// serving in first-come, first-served order

return false;

}

// The current order in servedOrders has now been "accounted for"

// so move on to the next one

servedOrdersIndex++;

return IsFirstComeFirstServed(takeOutOrders, dineInOrders, servedOrders,

servedOrdersIndex, takeOutOrdersIndex, dineInOrdersIndex);

}

C#

So now we're down to O(n)*O*(*n*) time, but we're still taking O(n)*O*(*n*) space in the call stack ↴ because of our recursion. We can rewrite this as an iterative method to get that memory cost down to O(1)*O*(1).

What's happening in each iteration of our recursive method? Sometimes we're taking a customer order out of takeOutOrders and sometimes we're taking a customer order out of dineInOrders, but we're *always* taking a customer order out of servedOrders.

So what if instead of taking customer orders out of servedOrders 1-by-1, we *iterated over them*?

That should work. But are we missing any edge cases?

What if there are *extra* orders in takeOutOrders or dineInOrders that don't appear in servedOrders? Would our kitchen be first-come, first-served then?

Maybe.

It's possible that our data doesn't include everything from the afternoon service. Maybe we stopped recording data before every order that went into the kitchen was served. It would be reasonable to say that our kitchen is still first-come, first-served, since we don't have any evidence otherwise.

On the other hand, if our input is supposed to cover the entire service, then any orders that went into the kitchen but weren't served should be investigated. We don't want to accept people's money but not serve them!

When in doubt, ask! This is a *great* question to talk through with your interviewer and shows that you're able to think through edge cases.

Both options are reasonable. In this writeup, we'll enforce that *every* order that goes into the kitchen (appearing in takeOutOrders or dineInOrders) should come out of the kitchen (appearing in servedOrders). How can we check that?

To check that we've served every order that was placed, we'll validate that when we finish iterating through servedOrders, we've exhausted takeOutOrders and dineInOrders.

Solution

We walk through servedOrders, seeing if each customer order *so far* matches a customer order from one of the two registers. To check this, we:

1. Keep pointers to the current index in takeOutOrders, dineInOrders, and servedOrders.
2. Walk through servedOrders from beginning to end.
3. If the current order in servedOrders is the same as the current customer order in takeOutOrders, increment takeOutOrdersIndex and keep going. This can be thought of as "checking off" the current customer order in takeOutOrders and servedOrders, reducing the problem to the remaining customer orders in the arrays.
4. Same as above with dineInOrders.
5. If the current order isn't the same as the customer order at the front of takeOutOrders or dineInOrders, we know something's gone wrong and we're not serving food first-come, first-served.
6. If we make it all the way to the end of servedOrders, we'll check that we've reached the end of takeOutOrders and dineInOrders. If every customer order checks out, that means we're serving food first-come, first-served.

public static bool IsFirstComeFirstServed(int[] takeOutOrders, int[] dineInOrders, int[] servedOrders){

int takeOutOrdersIndex = 0;

int dineInOrdersIndex = 0;

foreach (var order in servedOrders) {

if (takeOutOrdersIndex < takeOutOrders.Length && order == takeOutOrders[takeOutOrdersIndex]) {

// If we still have orders in takeOutOrders

// and the current order in takeOutOrders is the same

// as the current order in servedOrders

takeOutOrdersIndex++;

}

else if (dineInOrdersIndex < dineInOrders.Length && order == dineInOrders[dineInOrdersIndex]) {

// If we still have orders in dineInOrders

// and the current order in dineInOrders is the same

// as the current order in servedOrders

dineInOrdersIndex++;

}

else {

// If the current order in servedOrders doesn't match the current

// order in takeOutOrders or dineInOrders, then we're not serving first-come,

// first-served

return false;

}

}

// Check for any extra orders at the end of takeOutOrders or dineInOrders

if (dineInOrdersIndex != dineInOrders.Length || takeOutOrdersIndex != takeOutOrders.Length) {

return false;

}

// All orders in servedOrders have been "accounted for"

// so we're serving first-come, first-served!

return true;

}

Complexity

O(n)*O*(*n*) time and O(1)*O*(1) additional space.

Bonus

1. This assumes each customer order in servedOrders is unique. How can we adapt this to handle arrays of customer orders with *potential repeats*?
2. Our implementation returns true when all the items in dineInOrders and takeOutOrders are first-come first-served in servedOrders and false otherwise. That said, it'd be reasonable to throw an exception if some orders that went into the kitchen were never served, or orders were served but not paid for at either register. How could we check for those cases?
3. Our solution iterates through the customer orders from front to back. Would our algorithm work if we iterated from the back towards the front? Which approach is cleaner?

What We Learned

If you read the whole breakdown section, you might have noticed that our recursive function cost us extra space. If you missed that part, go back and take a look.

Be careful of the hidden space costs from a recursive function's call stack! If you have a solution that's recursive, see if you can save space by using an iterative algorithm instead.

**6. Dynamic programming and recursion**

**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 sequence (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 n*n*th Fibonacci number, we could use this simple recursive algorithm:

public int Fib(int n){

if (n == 0 || n == 1)

return n;

return Fib(n - 1) + Fib(n - 2);

}

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 method ends up recursively calling Fib(2) *three times*. So the problem of finding the n*n*th Fibonacci number has overlapping subproblems.

**Memoization**

Memoization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a dictionary).

For example, a simple recursive method for computing the n*n*th Fibonacci number:

public int Fib(int n){

// Edge case

if (n < 0)

throw new ArgumentOutOfRangeException(nameof(n), "Index was negative.");

// Base cases

if (n == 0 || n == 1)

return n;

Console.WriteLine($"Computing Fib({n})");

return Fib(n - 1) + Fib(n - 2);

}

Will run on the same inputs multiple times:

PS C:\> Fib(5)

Computing Fib(5)

Computing Fib(4)

Computing Fib(3)

Computing Fib(2)

Computing Fib(2)

Computing Fib(3)

Computing Fib(2)

5

We can imagine the recursive calls of this method as a tree, where the two children of a node are the two recursive calls it makes. We can see that the tree quickly branches out of control:

To avoid the duplicate work caused by the branching, we can wrap the method in a class with an instance variable, \_memo, that maps inputs to outputs. Then we simply

1. check \_memo to see if we can avoid computing the answer for any given input, and
2. save the results of any calculations to \_memo.

public class Fibber{

private Dictionary<int, int> \_memo = new Dictionary<int, int>();

public int Fib(int n) {

// Edge case

if (n < 0)

throw new ArgumentOutOfRangeException(nameof(n), "Index was negative");

// Base cases

if (n == 0 || n == 1)

return n;

// See if we've already calculated this

if (\_memo.ContainsKey(n)) {

Console.WriteLine($"Grabbing \_memo[{n}]");

return \_memo[n];

}

Console.WriteLine($"Computing Fib({n})");

int result = Fib(n - 1) + Fib(n - 2);

// Memoize

\_memo[n] = result;

return result;

}

}

We save a bunch of calls by checking the memo:

PS C:\> new Fibber().Fib(5)

Computing Fib(5)

Computing Fib(4)

Computing Fib(3)

Computing Fib(2)

Grabbing \_memo[2]

Grabbing \_memo[3]

5

Now in our recurrence tree, no node appears more than twice:

Memoization 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 [going bottom-up](https://www.interviewcake.com/concept/bottom-up), which is usually cleaner and often more efficient.

**Bottom-Up Algorithms**

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..n1..*n*, we could use this cute, top-down, recursive one-liner:

public int Product1ToN(int n){

// We assume n >= 1

return (n > 1) ? (n \* Product1ToN(n - 1)) : 1;

}

This approach has a problem: it builds up a call stack of size O(n)*O*(*n*), which makes our total memory cost O(n)*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:

public int Product1ToN(int n){

// We assume n >= 1

int result = 1;

for (int num = 1; num <= n; num++)

result \*= num;

return result;

}

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

*Some* compilers and interpreters will do what's called tail call optimization (TCO), where it can optimize *some* recursive methods 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 multiplying the numbers 1..n1..*n*, above). The other common strategy for dynamic programming problems is [memoization](https://www.interviewcake.com/concept/memoization).

**Problems**

**Write a recursive method for generating all permutations of an input string. Return them as a set.**

Don't worry about time or space complexity—if we wanted efficiency we'd write an iterative version.

To start, assume every character in the input string is unique.

Your method can have loops—it just needs to also be recursive.

**Gotchas**

Make sure you have a base case! ↴ Otherwise your method may never terminate!

**Breakdown**

Let's break the problem into subproblems. How could we re-phrase the problem of getting all permutations for "cats" in terms of a smaller but similar subproblem?

**Let's make our subproblem be getting all permutations for all characters except the last one**. If we had all permutations for "cat," how could we use that to generate all permutations for "cats"?

We could put the "s" in each possible position for each possible permutation of "cat"!

These are our permutations of "cat":

cat cta atc act tac tca

For each of them, we add "s" in each possible position. So for "cat":

cat scat csat cast cats

And for "cta":

cta scta csta ctsa ctas

And so on.

Now that we can break the problem into subproblems, we just need a base case and we have a recursive algorithm!

**Solution**

If we're making all permutations for "cat," we take all permutations for "ca" and then put "t" in each possible position in each of those permutations. We use this approach recursively:

using System.Collections.Generic;

using System.Linq;

public static ISet<string> GetPermutations(string inputString){

// Base case

if (inputString.Length <= 1)

return new HashSet<string>() { inputString };

var allCharsExceptLast = inputString.Substring(0, inputString.Length - 1);

char lastChar = inputString[inputString.Length - 1];

// Recursive call: get all possible permutations for all chars except last

var permutationsOfAllCharsExceptLast = GetPermutations(allCharsExceptLast);

// Put the last char in all possible positions for each of the above permutations

var permutations = new HashSet<string>();

foreach (var permutationOfAllCharsExceptLast in permutationsOfAllCharsExceptLast) {

for (int position = 0; position <= allCharsExceptLast.Length; position++) {

var permutation = permutationOfAllCharsExceptLast.Substring(0, position)

+ lastChar + permutationOfAllCharsExceptLast.Substring(position);

permutations.Add(permutation);

}

}

return permutations;

}

**Bonus**

How does the problem change if the string can have duplicate characters?

What if we wanted to bring down the time and/or space costs?

**What We Learned**

This is one where *playing with a sample input* is huge. Sometimes it helps to think of algorithm design as a two-part process: *first* figure out how you would solve the problem "by hand," as though the input was a stack of paper on a desk in front of you. *Then* translate that process into code.

**Write a method Fib() that takes an integer n*n* and returns the n*n*th Fibonacci ↴ number.**

Let's say our Fibonacci series is 0-indexed and starts with 0. So:

Fib(0); // => 0

Fib(1); // => 1

Fib(2); // => 1

Fib(3); // => 2

Fib(4); // => 3

…

**Gotchas**

Our solution runs in n*n* time.

There's a clever, more mathy solution that runs in O(\lg{n})*O*(lg*n*) time, but we'll leave that one as a bonus.

If you wrote a recursive method, think carefully about what it does. It might do repeat work, like computing Fib(2) multiple times!

We can do this in O(1)*O*(1) space. If you wrote a recursive method, there might be a hidden space cost in the call stack! ↴

**Breakdown**

The n*n*th Fibonacci number is defined in terms of the two *previous* Fibonacci numbers, so this seems to lend itself to recursion.

Fib(n) = Fib(n - 1) + Fib(n - 2);

Can you write up a recursive solution?

As with any recursive method, we just need a base case and a recursive case:

1. **Base case:** n*n* is 0 or 1. Return n*n*.
2. **Recursive case:** Return Fib(n - 1) + Fib(n - 2).

public static int Fib(int n){

if (n == 0 || n == 1)

return n;

return Fib(n - 1) + Fib(n - 2);

}

Okay, this'll work! What's our time complexity?

It's not super obvious. We might guess n*n*, but that's not quite right. Can you see why?

Each call to Fib() makes *two more calls*. Let's look at a specific example. Let's say n=5*n*=5. **If we call Fib(5), how many calls do we make in total?**

Try drawing it out as a tree where each call has two child calls, unless it's a base case.

Here's what the tree looks like:

We can notice this is a binary tree ↴ whose height is n*n*, which means the total number of nodes is O(2^n)*O*(2*n*).

So our total runtime is O(2^n)*O*(2*n*). That's an "exponential time cost," since the n*n* is *in an exponent*. Exponential costs are *terrible*. This is way worse than O(n^2)*O*(*n*2) or even O(n^{100})*O*(*n*100).

Our recurrence tree above essentially gets twice as big each time we add 1 to n*n*. So as n*n* gets really big, our runtime quickly spirals out of control.

The craziness of our time cost comes from the fact that we're doing so much repeat work. How can we avoid doing this repeat work?

We can memoize! ↴

Let's wrap Fib() in a class with an instance variable where we store the answer for any n*n* that we compute:

using System;

using System.Collections.Generic;

class Fibber{

private Dictionary<int, int> \_memo = new Dictionary<int, int>();

public int Fib(int n) {

// Edge case: negative index

if (n < 0)

throw new ArgumentOutOfRangeException(nameof(n), "Index was negative.");

// Base case: 0 or 1

if (n == 0 || n == 1)

return n;

// See if we've already calculated this

int result;

if (!\_memo.TryGetValue(n, out result)) {

// Not yet, so compute it

result = Fib(n - 1) + Fib(n - 2);

// Memoize

\_memo.Add(n, result);

}

return result;

}

}

What's our time cost now?

Our recurrence tree will look like this:

The computer will build up a call stack with Fib(5), Fib(4), Fib(3), Fib(2), Fib(1). Then we'll start returning, and on the way back up our tree we'll be able to compute each node's 2nd call to Fib() in constant time by just looking in the memo. n*n* time in total.

What about space? \_memo takes up n*n* space. Plus we're still building up a call stack that'll occupy n*n* space. Can we avoid one or both of these space expenses?

Look again at that tree. Notice that to calculate Fib(5) we worked "down" to Fib(4), Fib(3), Fib(2), etc.

What if instead we *started* with Fib(0) and Fib(1) and worked "up" to n*n*?

**Solution**

We use a bottom-up ↴ approach, starting with the 0th Fibonacci number and iteratively computing subsequent numbers until we get to n*n*.

using System;

public static int Fib(int n){

// Edge cases:

if (n < 0)

throw new ArgumentException("Index was negative. No such thing as a negative index in a series.");

if (n == 0 || n == 1)

return n;

// We'll be building the fibonacci series from the bottom up.

// So we'll need to track the previous 2 numbers at each step.

int prevPrev = 0; // 0th fibonacci

int prev = 1; // 1st fibonacci

int current = 0; // Declare and initialize current

for (int i = 1; i < n; i++) {

// Iteration 1: current = 2nd fibonacci

// Iteration 2: current = 3rd fibonacci

// Iteration 3: current = 4th fibonacci

// To get nth fibonacci ... do n-1 iterations.

current = prev + prevPrev;

prevPrev = prev;

prev = current;

}

return current;

}

**Complexity**

O(n)*O*(*n*) time and O(1)*O*(1) space.

**Bonus**

* If you're good with matrix multiplication you can bring the time cost down even further, to O(lg(n))*O*(*lg*(*n*)). Can you figure out how?

**What We Learned**

This one's a good illustration of the tradeoff we sometimes have between code cleanliness and efficiency.

We could use a cute, recursive method to solve the problem. But that would cost O(2^n)*O*(2*n*) time as opposed to n*n* time in our final bottom-up solution. Massive difference!

In general, whenever you have a recursive solution to a problem, think about what's *actually happening on the call stack*. An iterative solution might be more efficient.

**Your quirky boss collects rare, old coins...**

They found out you're a programmer and asked you to solve something they've been wondering for a long time.

Write a method that, given:

1. an amount of money
2. an array of coin denominations

computes the number of ways to make the amount of money with coins of the available denominations.

**Example:** for amount=44 (44¢) and denominations=[1,2,3][1,2,3] (11¢, 22¢ and 33¢), your program would output 44—the number of ways to make 44¢ with those denominations:

1. 1¢, 1¢, 1¢, 1¢
2. 1¢, 1¢, 2¢
3. 1¢, 3¢
4. 2¢, 2¢

**Gotchas**

What if there's *no way* to make the amount with the denominations? Does your method have reasonable behavior?

We can do this in O(n\*m)*O*(*n*∗*m*) time and O(n)*O*(*n*) space, where n*n* is the amount of money and m*m* is the number of denominations.

A simple recursive approach works, but you'll find that your method gets called more than once with the same inputs. We can do better.

We could avoid the duplicate method calls by memoizing, ↴ but there's a cleaner bottom-up ↴ approach.

**Breakdown**

We need to find some way to break this problem down into subproblems.

Here's one way: for **each denomination**, we can use it once, or twice, or...as many times as it takes to reach or overshoot the amount with coins of that denomination alone.

For each of those choices of how many times to include coins of each denomination, we're left with the subproblem of seeing how many ways we can get the remaining amount from the remaining denominations.

Here's that approach in pseudocode:

public static int NumberOfWays(amount, denominations){

answer = 0;

foreach (denomination in denominations) {

foreach (numTimesToUseDenomination in possibleNumTimesToUseDenomWithoutOvershootingAmt) {

answer += NumberOfWays(amountRemaining, otherDenominations);

}

}

return answer;

}

The answer for some of those subproblems will of course be 0. For example, there's no way to get 1¢ with only 2¢ coins.

As a recursive method, we could formalize this as:

using System;

using System.Linq;

public static int ChangePossibilitiesTopDown(

int amountLeft, int[] denominations, int currentIndex = 0){

// Base cases:

// We hit the amount spot on. Yes!

if (amountLeft == 0)

return 1;

// We overshot the amount left (used too many coins)

if (amountLeft < 0)

return 0;

// We're out of denominations

if (currentIndex == denominations.Length)

return 0;

// Print out actual part of array

Console.Write($"checking ways to make {amountLeft} with ");

Console.WriteLine($"[{string.Join(", ",

denominations.Skip(currentIndex).Take(

denominations.Length - currentIndex))}]");

// Choose a current coin

int currentCoin = denominations[currentIndex];

// See how many possibilities we can get

// for each number of times to use currentCoin

int numPossibilities = 0;

while (amountLeft >= 0) {

numPossibilities += ChangePossibilitiesTopDown(amountLeft,

denominations, currentIndex + 1);

amountLeft -= currentCoin;

}

return numPossibilities;

}

But there's a problem—we'll often **duplicate** the work of checking remaining change possibilities. Note the duplicate calls with the input 4, [1,2,3]:

PS C:\> ChangePossibilitiesTopDown(4, [1, 2, 3])

checking ways to make 4 with [1, 2, 3]

checking ways to make 4 with [2, 3]

checking ways to make 4 with [3]

checking ways to make 2 with [3]

checking ways to make 3 with [2, 3]

checking ways to make 3 with [3]

checking ways to make 1 with [3]

checking ways to make 2 with [2, 3]

checking ways to make 2 with [3]

checking ways to make 1 with [2, 3]

checking ways to make 1 with [3]

4

For example, we check ways to make 2 with [3] *twice*.

We can do better. How do we avoid this duplicate work and bring down the time cost?

One way is to **memoize**. ↴

Here's what the memoization might look like:

using System;

using System.Collections.Generic;

using System.Linq;

public class Change{

private Dictionary<string, int> \_memo = new Dictionary<string, int>();

public int ChangePossibilitiesTopDown(

int amountLeft, int[] denominations, int currentIndex = 0) {

// Check our memo and short-circuit if we've already solved this one

string memoKey = $"{amountLeft}, {currentIndex}";

if (\_memo.ContainsKey(memoKey)) {

Console.WriteLine($"grabbing memo [{memoKey}]");

return \_memo[memoKey];

}

// Base cases:

// We hit the amount spot on. Yes!

if (amountLeft == 0)

return 1;

// We overshot the amount left (used too many coins)

if (amountLeft < 0)

return 0;

// We're out of denominations

if (currentIndex == denominations.Length)

return 0;

// Print out actual part of array

Console.Write($"checking ways to make {amountLeft} with ");

Console.WriteLine($"[{string.Join(", ", denominations.Skip(currentIndex).Take(

denominations.Length - currentIndex))}]");

// Choose a current coin

int currentCoin = denominations[currentIndex];

// See how many possibilities we can get

// for each number of times to use currentCoin

int numPossibilities = 0;

while (amountLeft >= 0) {

numPossibilities += ChangePossibilitiesTopDown(

amountLeft, denominations, currentIndex + 1);

amountLeft -= currentCoin;

}

// Save the answer in our memo, so we don't compute it again

\_memo.Add(memoKey, numPossibilities);

return numPossibilities;

}

}

And now our checking has no duplication:

PS C:\> var change = new Change();

PS C:\> change.ChangePossibilitiesTopDown(4, [1, 2, 3]);

checking ways to make 4 with [1, 2, 3]

checking ways to make 4 with [2, 3]

checking ways to make 4 with [3]

checking ways to make 2 with [3]

checking ways to make 3 with [2, 3]

checking ways to make 3 with [3]

checking ways to make 1 with [3]

checking ways to make 2 with [2, 3]

grabbing memo [2, 2]

checking ways to make 1 with [2, 3]

grabbing memo [1, 2]

4

This answer is quite good. It certainly solves our duplicate work problem. It takes O(n\*m)*O*(*n*∗*m*) time and O(n\*m)*O*(*n*∗*m*) space, where n*n* is the size of amount and m is the number of items in denominations. (Except we'd need to remove the line where we print "checking ways to make..." because making all those subarrays will take O(m^2)*O*(*m*2) space!)

However, we can do better. Because our method is recursive it will build up a **large call stack** ↴ of size O(m)*O*(*m*). Of course, this cost is eclipsed by the memory cost of \_memo, which is O(n\*m)*O*(*n*∗*m*). But it's still best to avoid building up a large stack like this, because it can cause a **stack overflow** (yes, that means recursion is *usually* better to avoid for methods that might have arbitrarily large inputs).

It turns out we can get O(n)*O*(*n*) additional space.

A great way to avoid recursion is to go **bottom-up**. ↴

Our recursive approach was top-down because it started with the final value for amount and recursively broke the problem down into subproblems with smaller values for amount. What if instead we tried to **compute the answer for small values of amount first**, and use those answers to iteratively compute the answer for higher values until arriving at the final amount?

We can **start by making an array waysOfDoingNCents**, where the index is the amount and the value at each index is the number of ways of getting that amount.

This array will take O(n)*O*(*n*) space, where n*n* is the size of amount.

To further simplify the problem, we can work with only the first coin in denominations, then add in the second coin, then the third, etc.

What would waysOfDoingNCents look like for just our first coin: 1¢? Let's call this waysOfDoingNCents1.

int[] waysOfDoingNCents1 = new[]

{

1, // 0c: no coins

1, // 1c: 1 1c coin

1, // 2c: 2 1c coins

1, // 3c: 3 1c coins

1, // 4c: 4 1c coins

1, // 5c: 5 1c coins

};

Now what if we add a 2¢ coin?

int[] waysOfDoingNCents1And2 = new[]{

1, // 0c: no change

1, // 1c: no change

1 + 1, // 2c: new [(2)]

1 + 1, // 3c: new [(2, 1)]

1 + 2, // 4c: new [(2, 1, 1), (2, 2)]

1 + 2, // 5c: new [(2, 1, 1, 1), (2, 2, 1)]

};

How do we formalize this process of going from waysOfDoingNCents1 to waysOfDoingNCents1And2?

Let's **suppose we're partway through already** (this is a classic dynamic programming approach). Say we're trying to calculate waysOfDoingNCents1And2[5]. Because we're going bottom-up, we know we already have:

1. waysOfDoingNCents1And2 for amounts less than 55
2. a fully-populated waysOfDoingNCents1

So how many *new* ways should we add to waysOfDoingNCents1[5] to get waysOfDoingNCents1And2[5]?

Well, if there are *any* new ways to get 5¢ now that we have 2¢ coins, those new ways must involve at least one 2¢ coin. So if we presuppose that we'll use one 2¢ coin, that leaves us with 5-2=35−2=3 left to come up with. We already know how many ways we can get 3¢ with 1¢ and 2¢ coins: waysOfDoingNCents1And2[3], which is 22.

So we can see that:

waysOfDoingNCents1And2[5] = waysOfDoingNCents1[5] + waysOfDoingNCents1And2[5 - 2]

**Why don't we also need to check waysOfDoingNCents1And2[5 - 2 - 2] (two 2¢ coins)?** Because we already checked waysOfDoingNCents1And2[1] when calculating waysOfDoingNCents1And2[3]. We'd be counting some arrangements multiple times. In other words, waysOfDoingNCents1And2[k] already includes the full count of possibilities for getting k*k*, including possibilities that use 2¢ any number of times. We're only interested in how many *more* possibilities we might get when we go from k*k* to k+2*k*+2 and thus have the ability to add one *more* 2¢ coin to each of the possibilities we have for k*k*.

**Solution**

We use a bottom-up ↴ algorithm to build up a table waysOfDoingNCents such that waysOfDoingNCents[k] is how many ways we can get to k cents using our denominations. We start with the base case that there's **one way to create the amount zero**, and progressively add each of our denominations.

The number of new ways we can make a higherAmount when we account for a new coin is simply waysOfDoingNCents[higherAmount - coin], where we know that value already includes combinations involving coin (because we went bottom-up, we know smaller values have already been calculated).

public static int ChangePossibilitiesBottomUp(int amount, int[] denominations){

// Array of zeros from 0..amount

int[] waysOfDoingNCents = new int[amount + 1];

waysOfDoingNCents[0] = 1;

foreach (int coin in denominations) {

for (int higherAmount = coin; higherAmount <= amount; higherAmount++) {

int higherAmountRemainder = higherAmount - coin;

waysOfDoingNCents[higherAmount] +=

waysOfDoingNCents[higherAmountRemainder];

}

}

return waysOfDoingNCents[amount];

}

Here's how waysOfDoingNCents would look in successive iterations of our method for amount=55 and denominations=[1,3,5][1,3,5].

===========

key:

a = higherAmount

r = higherAmountRemainder

===========

for coin = 1:

[1, 1, 0, 0, 0, 0]

[1, 1, 1, 0, 0, 0]

[1, 1, 1, 1, 0, 0]

[1, 1, 1, 1, 1, 0]

[1, 1, 1, 1, 1, 1]

for coin = 3:

[1, 1, 1, 2, 1, 1]

[1, 1, 1, 2, 2, 1]

[1, 1, 1, 2, 2, 2]

for coin = 5:

[1, 1, 1, 2, 2, 3]

final answer: 3

**Complexity**

O(n\*m)*O*(*n*∗*m*) time and O(n)*O*(*n*) additional space, where n*n* is the amount of money and m*m* is the number of potential denominations.

**What We Learned**

This question is in a broad class called "dynamic programming." We have a bunch more [dynamic programming questions](https://www.interviewcake.com/concept/bottom-up#related_questions) we'll go over later.

Dynamic programming is *kind of* like the next step up from greedy. ↴ You're taking that idea of "keeping track of what we need in order to update the best answer so far," and applying it to situations where the new best answer so far might not *just* have to do with the previous answer, but some *other* earlier answer as well.

So as you can see in this problem, we kept track of *all* of our previous answers to smaller versions of the problem (called "subproblems") in a big array called waysOfDoingNCents.

Again, same *idea* of keeping track of what we need in order to update the answer as we go, like we did when storing the max product of 2, min product of 2, etc in the [highest product of 3](https://www.interviewcake.com/question/highest-product-of-3) question. Except now the thing we need to keep track of is *all* our previous answers, which we're keeping in an array.

We built that array bottom-up, but we also talked about how we could do it top-down and memoize. Going bottom-up is cleaner and usually more efficient, but often it's easier to think of the top-down version first and try to adapt from there.

Dynamic programming is a weak point for lots of candidates. If this one was tricky for you, don't fret. We have more coming later.

**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:

public class CakeType{

public readonly int Weight;

public readonly long Value;

public CakeType(int weight, long value) {

Weight = weight;

Value = value;

}

}

C#

For example:

// Weighs 7 kilograms and has a value of 160 shillings

var cakeType = new CakeType(7, 160);

// Weighs 3 kilograms and has a value of 90 shillings

var yetAnotherCakeType = 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:

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

int capacity = 20;

// Returns 555 (6 of the middle type of cake and 1 of the last type of cake)

MaxDuffelBagValue(cakeTypes, capacity);

**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 method work if the duffel bag's weight capacity is 0 kg?

Does your method 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)*O*(*n*∗*k*) time and O(k)*O*(*k*) space, where n*n* is the number of types of cakes and k*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:

var cakeType = new CakeType(1, 30);

var anotherCakeType = 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:

var cakeType = new CakeType(1, 30);

var anotherCakeType = 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 (200/50200/50), but the first type of cake is worth 30 shillings/kg (30/130/1)!

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:

var cakeType = new CakeType(3, 40);

var anotherCakeType = 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:

using System;

public static long MaxDuffelBagValueWithCapacity1(CakeType[] cakeTypes){

long maxValueAtCapacity1 = 0;

foreach (var cakeType in 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:

long maxValueAtCapacity3 = 0;

foreach (var cakeType in 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:

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:

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

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.

var 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:

// 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):

foreach (var cakeType in 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:

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++) {

foreach (var cakeType in 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?

// 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*?

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

int remainingCapacityAfterTakingCake = currentCapacity - cakeType.Weight;

So maxValueUsingCake is:

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

int remainingCapacityAfterTakingCake = currentCapacity - cakeType.Weight;

long maxValueUsingCake = cakeType.Value + maxValuesAtCapacities[remainingCapacityAfterTakingCake];

We can squish this into one line:

long maxValueUsingCake = cakeType.Value + maxValuesAtCapacities[currentCapacity - cakeType.Weight];

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

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 currentMaxValue 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 currentMaxValue, we'll update currentMaxValue!

currentMaxValue = Math.Max(maxValueUsingCake, currentMaxValue);

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

We save each currentMaxValue in the maxValuesAtCapacities array. We'll also need to make sure we set currentMaxValue 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 method so far:

using System;

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++) {

// Set a variable to hold the max monetary value so far for currentCapacity

long currentMaxValue = 0;

foreach (var cakeType in cakeTypes) {

// If the current cake weighs as much or less than the current weight capacity

// it's possible taking the cake would get a better value

if (cakeType.Weight <= currentCapacity) {

// 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 maxValuesAtCapacities.

long maxValueUsingCake = cakeType.Value

+ maxValuesAtCapacities[currentCapacity - cakeType.Weight];

// Now we see if it's worth taking the cake.

// How does the value with the cake compare to the currentMaxValue?

currentMaxValue = Math.Max(maxValueUsingCake, currentMaxValue);

}

}

// Add each capacity's max value to our array so we can use them

// when calculating all the remaining capacities

maxValuesAtCapacities[currentCapacity] = currentMaxValue;

}

}

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

Our final answer is maxValuesAtCapacities[weightCapacity]!

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 a couple choices. We could return:

1. **The highest possible long.** In C#, that'd be long.MaxValue.
2. **Raise an exception** indicating the answer is infinity.

What are the advantages and disadvantages of each option?

For the **first option** the advantage is the highest possible long will *behave* like infinity in a few ways. For example, it'll be greater than any other integer. But it's a still a *specific* number, which can be an advantage or disadvantage—we might want our result to always be the same *type*, but representing infinity as a specific number is "lossy"—it won't be clear if we're talking about an actual value or the special case of infinity.

The **second 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 method.

Either option *could* be reasonable. We'll go with the second one here.

**Solution**

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

We use a bottom-up ↴ approach to find the max value at our duffel bag's weightCapacity by finding the max value at *every* capacity from 0 to weightCapacity.

We allocate an array maxValuesAtCapacities 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 maxValuesAtCapacities

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

using System;

public class InfinityException : Exception{

public InfinityException() :

base("Max value is infinity!") { }

}

public class CakeType{

public readonly int Weight;

public readonly long Value;

public CakeType(int weight, int value) {

Weight = weight;

Value = value;

}

}

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++) {

// Set a variable to hold the max monetary value so far for currentCapacity

long currentMaxValue = 0;

foreach (var cakeType in cakeTypes) {

// If a cake weighs 0 and has a positive value the value of our duffel bag is infinite!

if (cakeType.Weight == 0 && cakeType.Value != 0)

throw new InfinityException();

// If the current cake weighs as much or less than the current weight capacity

// it's possible taking the cake would get a better value

if (cakeType.Weight <= currentCapacity) {

// 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 maxValuesAtCapacities.

long maxValueUsingCake = cakeType.Value

+ maxValuesAtCapacities[currentCapacity - cakeType.Weight];

// Now we see if it's worth taking the cake. how does the

// value with the cake compare to the currentMaxValue?

currentMaxValue = Math.Max(maxValueUsingCake, currentMaxValue);

}

}

// Add each capacity's max value to our array so we can use them

// when calculating all the remaining capacities

maxValuesAtCapacities[currentCapacity] = currentMaxValue;

}

return maxValuesAtCapacities[weightCapacity];

}

**Complexity**

O(n\*k)*O*(*n*∗*k*) time, and O(k)*O*(*k*) space, where n*n* is number of types of cake and k*k* is the capacity of the duffel bag. We loop through each cake (n*n* cakes) for every capacity (k*k* capacities), so our runtime is O(n\*k)*O*(*n*∗*k*), and maintaining the array of k+1*k*+1 capacities gives us the O(k)*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)*O*(*n*∗*k*) growing out of control quickly if n*n* or k*k* is large.

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})*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 object 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).

**7. Queues and stacks**

**Bracket Validator**

**You're working with an intern that keeps coming to you with JavaScript code that won't run because the braces, brackets, and parentheses are off. To save you both some time, you decide to write a braces/brackets/parentheses validator.**

Let's say:

* '(', '{', '[' are called "*openers*."
* ')', '}', ']' are called "*closers*."

Write an efficient method that tells us whether or not an input string's openers and closers are properly nested.

Examples:

* "{ [ ] ( ) }" should return **true**
* "{ [ ( ] ) }" should return **false**
* "{ [ }" should return **false**

**Gotchas**

**Simply making sure each opener has a corresponding closer is not enough**—we must also confirm that they are correctly *ordered*.

For example, "{ [ ( ] ) }" should return false, even though each opener can be matched to a closer.

We can do this in O(n)*O*(*n*) time and space. One iteration is all we need!

**Breakdown**

We can use a greedy ↴ approach to walk through our string character by character, making sure the string validates "so far" until we reach the end.

What do we do when we find an opener or closer?

Well, we'll need to keep track of our openers so that we can confirm they get closed properly. What data structure should we use to store them? **When choosing a data structure, we should start by deciding on the properties we want**. In this case, we should figure out how we will want to *retrieve* our openers from the data structure! So next we need to know: what will we do when we find a closer?

Suppose we're in the middle of walking through our string, and we find our first closer:

[ { ( ) ] . . . .

^

How do we know whether or not that closer in that position is valid?

**A closer is valid if and only if it's the closer for the most recently seen, unclosed opener**. In this case, '(' was seen most recently, so we know our closing ')' is valid.

So we want to store our openers in such a way that we can **get the most recently added one quickly**, and we can **remove the most recently added one quickly** (when it gets closed). Does this sound familiar?

What we need is a stack! ↴

**Solution**

We iterate through our string, making sure that:

1. **each closer corresponds to the most recently seen, unclosed opener**
2. **every opener and closer is in a pair**

We use a stack ↴ to keep track of the most recently seen, unclosed opener. And if the stack is ever empty when we come to a closer, we know that closer doesn't have an opener.

So as we iterate:

* If we see an opener, we push it onto the stack.
* If we see a closer, we check to see if it is the closer for the opener at the top of the stack. If it is, we pop from the stack. If it isn't, or if the stack is empty, we return false.

If we finish iterating and our stack is empty, we know every opener was properly closed.

using System.Collections.Generic;

public static bool IsValid(String code)

{

var openersToClosers = new Dictionary<char, char>

{

{ '(', ')' },

{ '[', ']' },

{ '{', '}' }

};

var openers = new HashSet<char>(openersToClosers.Keys);

var closers = new HashSet<char>(openersToClosers.Values);

var openersStack = new Stack<char>();

foreach (char c in code)

{

if (openers.Contains(c))

{

openersStack.Push(c);

}

else if (closers.Contains(c))

{

if (openersStack.Count == 0)

{

return false;

}

else

{

char lastUnclosedOpener = openersStack.Pop();

// If this closer doesn't correspond to the most recently

// seen unclosed opener, short-circuit, returning false

if (openersToClosers[lastUnclosedOpener] != c)

{

return false;

}

}

}

}

return openersStack.Count == 0;

}



**Complexity**

O(n)*O*(*n*) time (one iteration through the string), and O(n)*O*(*n*) space (in the worst case, all of our characters are openers, so we push them all onto the stack).

**Bonus**

In Ruby, sometimes expressions are surrounded by vertical bars, "|like this|". Extend your validator to validate vertical bars. Careful: there's no difference between the "opener" and "closer" in this case—they're the same character!

**What We Learned**

The trick was to use a stack. ↴

It might have been difficult to have that insight, because you might not use stacks that much.

Two common uses for stacks are:

1. **parsing** (like in this problem)
2. **tree or graph traversal** (like depth-first traversal)

So remember, if you're doing either of those things, try using a stack