# Dynamic Programming for Interviews

## Using the FAST Method to Easily Solve Dynamic Programming Coding Interview Questions

www.byte-by-byte.com
**Sam Gavis-Hughson of Byte By Byte LLC**

bytebybyte

# Dynamic Programming for Interviews

Using the FAST method to easily solve dynamic programming coding interview questions

By Sam Gavis-Hughson
Creator of www.byte-by-byte.com

# Dedication

This book is dedicated to my ever-supportive parents. I literally wouldn't be here without you guys.

# A quick note on sharing

Thank you so much for downloading this ebook. I've put many hours and hundreds of dollars into putting together the best free resource that is available on dynamic programming.

I'd love for you to share this book with anyone who you think might enjoy it! In return, I ask that you only share the book through the official link (www.dynamicprogrammingbook.com) or by using one of the social media links at the bottom of each page. Please DO NOT share the PDF file directly.

# Contents

(clickable)


Dedication ..... 3
Introduction ..... 5
Prereqs ..... 6
Dynamic Programming Overview ..... 7
Dynamic Programming Practice ..... 9
Fibonacci Numbers ..... 12
Making Change ..... 17
Square Submatrix ..... 23
0-1 Knapsack ..... 29
Target Sum ..... 35
Closing thoughts ..... 40
Disclaimer ..... 42

# Introduction

When I ask people what they struggle with the most when preparing for interviews, I always hear the same answer: Dynamic programming.

This answer doesn't surprise me at all. Dynamic programming is somewhat unintuitive. There's an aura around it as something to be feared.

However, there is no logical reason why it should be this way. Dynamic programming can actually be easy and fun if you take the time to learn how to properly approach these problems.

This book attempts to capture the two most important concepts for dynamic programming interview problems:

1. Have a system to solve every problem
2. All dynamic programming problems are very similar

In order for you to understand how to approach any dynamic programming problem, this book introduces you to the FAST method. This four step process makes it very easy for you to clearly and consistently think about dynamic programming. This information will be reinforced through a review of five different sample problems that commonly occur in interviews.

Fully understanding the FAST method and how to apply it in our examples will prepare you for any dynamic programming problems that you might face in an interview situation. My hope is that when you reach the end of this book, you will no longer fear dynamic programming and will realize that it can actually be fun.

# Prereqs

## Programming fundamentals

This book assumes a solid level of coding ability. I have attempted to make the intentions clear in the code. It is likely difficult to follow if you don't have a good foundation.

## Recursion

Recursion is the key to understanding dynamic programming. Dynamic programming involves breaking problems down into smaller subproblems. The first step of the FAST method requires you to find a recursive solution to the problem and I will provide minimal guidance on this point. If you struggle with recursion, I recommend that you work on it before attempting to do dynamic programming.

# Dynamic Programming Overview

## What is Dynamic Programming?

It is important to have a good understanding of dynamic programming before we go any further. Generally speaking, dynamic programming is the technique of storing repeated computations in memory, rather than recomputing them every time you need them. The ultimate goal of this process is to improve runtime. Dynamic programming allows you to use more space to take less time.

The following two characteristics are required of all problems that can be optimized using dynamic programming: **Optimal substructure** and **overlapping subproblems**.

### Optimal Substructure

Optimal substructure requires that you can solve a problem based on the solutions of subproblems. For example, if you want to calculate the 5th Fibonacci number, it can be solved by computing `fib(5) = fib(4) + fib(3)`. It is not necessary to know any more information other than the solutions of those two subproblems, in order to get the solution.

A useful way to think about optimal substructure is whether a problem can be easily solved recursively. Recursive solutions inherently solve a problem by breaking it down into smaller subproblems. If you can solve a problem recursively, it most likely has an optimal substructure.

## Overlapping Subproblems

Overlapping subproblems means that when you split your problem into subproblems, you sometimes get the same subproblem multiple times. With the Fibonacci example, if we want to compute `fib(5)`, we need to compute `fib(4)` and `fib(3)`. However, to compute `fib(4)`, we need to compute `fib(3)` again. This is a wasted effort, since we've already computed the value of `fib(3)`.

Dynamic programming relies on overlapping subproblems, because it uses memory to save the values that have already been computed to avoid computing them again. The more overlap there is, the more computational time is saved.

# Key terms

There are several terms that are used frequently when discussing dynamic programming, those will be presented here.

## Memoization

Memoization (sounds like memorization) is the technique of writing a function that remembers the results of previous computations. This allows us to capitalize on overlapping subproblems.

To use memoization, a function can use a data structure (like an array or `HashMap`) to store the values it has previously computed and then look them up when it gets called. With the Fibonacci example, there could be an array where index `i == -1`, if we haven't computed the value or `i == fib`$_i$, if we have computed the value. Therefore, if we call `fib(3)` and `array[3] != -1`, we can return `array[3]` rather than recomputing the value.

## Top-down and bottom-up

Top-down and bottom-up refer to two general approaches to dynamic programming. A top-down solution starts with the final result and recursively breaks it down into subproblems. The bottom-up method does the opposite. It takes an iterative approach to solve the subproblems first and then works up to the desired solution.

This book works through problems by first finding a top-down solution and then converting it into a bottom-up solution. Bottom-up solutions are not always better than top-down solutions. The goal of the book is to demonstrate that both solutions are equally valid and that one solution can be determined from the other. In an interview situation, although bottom-up solutions often result in more concise code, either approach is appropriate. I recommend that you use whatever solution makes the most sense to you.

The distinction between top-down and bottom-up solutions will be discussed in detail in the upcoming examples. Therefore, if it not yet clear to you, it is not necessary to be concerned. The important point is that top-down = recursive and bottom-up = iterative.

# Dynamic Programming Practice

As with many things, talk is cheap. Now that we've discussed some of the higher-level ideas, let's dig into the specifics.

## The FAST method

The most successful interviewees are those who have developed a repeatable strategy to succeed. This is especially true for dynamic programming. This is the reason for the development of the FAST method.

There are four steps in the FAST method:

1. **F**irst solution
2. **A**nalyze the first solution
3. Identify the **S**ubproblems
4. **T**urn the solution around

By following these four steps, it is easy to come up with an optimal dynamic solution for almost any problem.

Find full code for all of the problems
in this book and other resources at
www.byte-by-byte.com/dpbook-resources.

## First solution

This is an important step for any interview question but is particularly important for dynamic programming. This step finds the first possible solution. This solution will be brute force and recursive. The goal is to solve the problem without concern for efficiency. It means that if you need to find the biggest/smallest/longest/shortest something, you should write code that goes through every possibility and then compares them all to find the best one.

Your solution must also meet these restrictions:

- The recursive calls must be self-contained. That means no global variables.
- You cannot do tail recursion. Your solution must compute the results to each subproblem and then combine them afterwards.
- Do not pass in unnecessary variables. Eg. If you can count the depth of your recursion as you return, don't pass a count variable into your recursive function.

Once you've gone through a couple problems, you will likely see how this solution looks almost the same every time.

## Analyze the first solution

In this step, we will analyze the first solution that you came up with. This involves determining the time and space complexity of your first solution and asking whether there is obvious room for improvement.

As part of the analytical process, we need to ask whether the first solution fits our rules for problems with dynamic solutions:

- Does it have an optimal substructure? Since our solution's recursive, then there is a strong likelihood that it meets this criteria. If we are recursively solving subproblems of the same problem, then we know that our substructure is optimal, otherwise our algorithm wouldn't work.
- Are there overlapping subproblems? This can be more difficult to determine because it doesn't always present itself with small examples. It may be necessary to try a medium-sized test case. This will enable you to see if you end up calling the same function with the same input multiple times.

## Find the Subproblems

If our solution can be made dynamic, the exact subproblems to memoize must be codified. This step requires us to discover the high-level meaning of the subproblems. This will make it easier to understand the solution later. Our recursive solution can be made dynamic by caching the values. This top-down solution facilitates a better understanding of the subproblems which is useful for the next step.

## Turn the solution around

We now have a top-down solution. This is fine and it would be possible to stop here. However, sometimes it is better to flip it around and to get a bottom up solution instead. Since we understand our subproblems, we will do that. This involves writing a completely different function (without modifying the existing code). This will iteratively compute the results of successive subproblems, until our desired result is reached.

The following sample problems use the FAST method to arrive at the solutions. By going through the examples, it will help you to understand how each of these steps is applied.

# Fibonacci Numbers

Given an integer `n`, write a function that will return the `n`th Fibonacci number.

eg.

```
fib(0) = 0
fib(1) = 1
fib(5) = 5
fib(10) = 55
```

## First solution

The first step in our FAST method is to find the first possible solution. With this type of problem, we can find a solution by focusing on the definition of a Fibonacci number. If you're not sure, you can ask your interviewer. However, it is likely that you remember the following from grade school:

```
fib(n) = fib(n - 1) + fib(n - 2)
fib(0) = 0
fib(1) = 1
```

Using these three expressions, it's possible to calculate any Fibonacci number.

When looking at these expressions, we can see that we've defined a recursive function. There are two base cases: `fib(0)` and `fib(1)`. There is also the recursive call: `fib(n) = fib(n - 1) + fib(n - 2)`. It is, therefore, possible to code up our first solution (*fig. 1*).

```
// Compute the nth Fibonacci number
// We assume that n >= 0 and that int is
// sufficient to hold the result
public int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2);
}
```

*Fig 1. Naive recursive Fibonacci solution*

## Analyze the first solution

The first iteration of our code has a short and sweet solution. It could even be reduced to two lines by combining the if statements.

But how does our solution stack up time-wise? Let's look at the execution of `fib(4)` (*fig. 2*).

Our graph shows that this solution isn't ideal. While `fib(0)` and `fib(1)` are constant time operations, `fib(2)` is called multiple times, each of which involves further recursive calls.

We can quickly compute the runtime, since we know that the depth of the recursive call (the height of the tree) will be `n`. This is because we recursively call our function with `n-1` each time. We also know that each recursive call results in two more calls, until we reach the base case. We can, therefore, say that we are making `1 + 2 + 4 + 8 + … + 2^n-1` function calls or `2^0 + 2^1 + … + 2^n-1` calls, which reduces to `O(2^n)`.

This is a terrible runtime!

Since we're using recursion, we can determine whether this problem is a potential candidate for dynamic programming. By looking at the solution, we can see that it has:

1. **Optimal substructure.** Our solution is recursive. Once we've solved one of the subproblems (ie. `fib(2)`), we can use these solutions to solve for greater values of `n`.

2. **Overlapping subproblems.** Since `fib(2)` gets called multiple times, it would be much more efficient if we were able to compute the solution to `fib(2)` only once.

Our problem has an optimal substructure and overlapping subproblems. Therefore, we know that we can improve our problem by using dynamic programming. We are now ready to move to the next step of our FAST method.

```
// Compute the nth Fibonacci number
public int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2);
}
```

*Fig 1. Repeated*

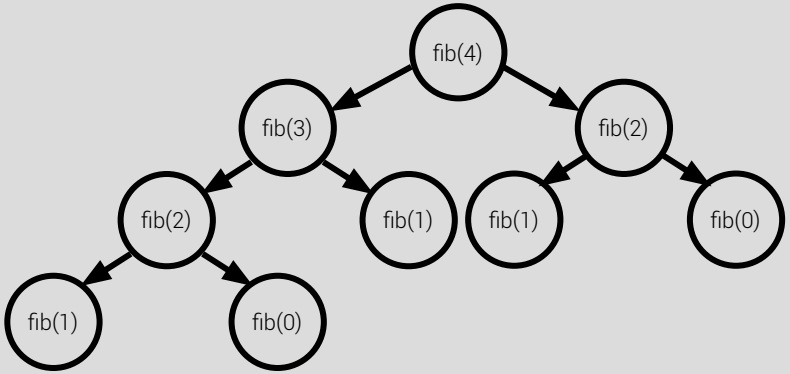



*Fig 2. Execution tree for* `fib(4)`

## Find the subproblems

In each recursive call, we break our problem into two subproblems. We then combine those two partial results to get our final result. In this case, for a function call of `fib(n)`, our two subproblems are `fib(n-1)` and `fib(n-2)`.

`fib(n-1)` and `fib(n-2)` are the two subproblems whose results we need in order to solve our problem. They are also the same subproblems that are being called multiple times with the same input. Therefore, we should cache the results of these subproblems.

However, before caching the values, it is essential to be clear on the meaning of the subproblems. In this case, it is simple - the value of `fib(n-1)` is just the `n-1`th Fibonacci number. With future problems, however, understanding the meaning of the subproblem is more complicated, as well as more important.

Understanding the subproblems allows us to add a cache to our original solution and to obtain a top-down dynamic programming solution (***figs. 3, 4***).

Our runtime now scales linearly, rather than exponentially. For `fib(n)`, we compute the Fibonacci value for each value from `1` to `n` exactly once. This gives us a runtime of `O(n)`. In terms of the space complexity, we have to use `O(n)` space to store the cache. However, since we are making a recursive call that goes `n` deep and uses `O(n)` stack space already, it does not affect our asymptotic space complexity.

```
// Compute the nth Fibonacci number recursively.
// Optimized by caching subproblem results
public int fib(int n) {
  if (n < 2) return n;
  // Create cache and initialize to -1
  int[] cache = new int[n+1];
  for (int i = 0; i < cache.length; i++) {
      cache[i] = -1;
  }
  // Fill initial values in cache
  cache[0] = 0;
  cache[1] = 1;
  return fib(n, cache);
}

// Overloaded private method
private int fib(int n, int[] cache) {
  // If value is set in cache, return
  if (cache[n] >= 0) return cache[n];
  // Compute and add to cache before returning
  cache[n] = fib(n-1, cache) + fib(n-2, cache);
  return cache[n];
}
```

*Fig 3. Top-down dynamic Fibonacci solution*

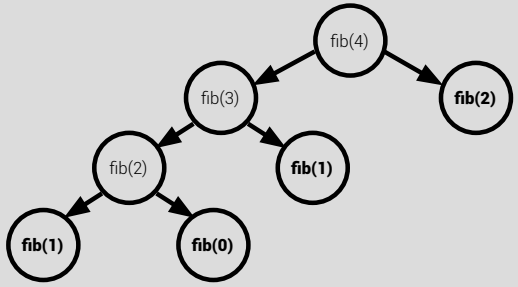



*Fig 4. Execution tree for fib(4). Cache hits are bolded*

## Turn the solution around

Since we now have a top-down solution, it is possible to reverse the process and solve it from the bottom up. This can be done by starting with the base cases and building up the solution from there by computing the results of each subsequent subproblem, until we reach our result.

In this problem, our base cases are `fib(0) = 0` and `fib(1) = 1`. From these two values, we can compute the next largest Fibonacci number, `fib(2) = fib(0) + fib(1)`. Once we have the value of `fib(2)`, we can calculate `fib(3)` etc. As we successively compute each Fibonacci number, the previous values are saved and referred to as necessary, eventually reaching `fib(n)`.

Our code for this process is fairly straightforward (***fig 5***).

This process yields a bottom-up solution. Since we iterate through all of the numbers from `0` to `n` once, our time complexity will be `O(n)` and our space will also be `O(n)`, since we create a 1D array from `0` to `n`. This makes our current solution comparable to the top-down solution, although without recursion. This code is likely easier to understand.

Because we understood the meaning of our subproblems and how to combine them into subsequently larger solutions, it was easy to write this code. A full understanding of the problem means that converting from top-down to bottom-up doesn't have to be a difficult task.

```
// Compute the nth Fibonacci number iteratively
public int fib(int n) {
  if (n == 0) return 0;

  // Initialize cache
  int[] cache = new int[n+1];
  cache[1] = 1;

  // Fill cache iteratively
  for (int i = 2; i <= n; i++) {
      cache[i] = cache[i-1] + cache[i-2];
  }

  return cache[n];
}
```

*Fig 5. Bottom-up dynamic Fibonacci solution*

Many problems would actually be solved by this point. However, in this case it is possible to improve our solution further. During the computation process, we only refer to the most recent two subproblems (`cache[i-1]` and `cache[i-2]`) to compute the value of the current subproblem. Therefore, `cache[0]` through `cache[i-3]` are unnecessary and do not need to be kept in memory.

We can, therefore, improve the space complexity of our solution to `O(1)` by only caching the most recent two values.

While this additional optimization is not applicable to all problems, it is useful to look for the opportunity to use it, wherever possible.

## Conclusion

This problem is a perfect introduction to dynamic programming. It lends itself to recursion and has clear subproblems that make it easy to cache for a dynamic solution. Although other problems can be more complicated, the concepts in this problem easily carry over.

```
// Compute the nth Fibonacci number iteratively
// with constant space. We only need to save
// the two most recently computed values
public int fib(int n) {
  if (n < 2) return n;
  int n1 = 1, n2 = 0;
  for (int i = 2; i < n; i++) {
    int n0 = n1 + n2;
    n2 = n1;
    n1 = n0;
  }

  return n1 + n2;
}
```

*Fig 6. Optimized bottom-up dynamic Fibonacci solution*

# Making Change

Given an integer representing a given amount of change, write a function to compute the total number of coins required to make that amount of change. You can assume that there is always a 1¢ coin.

eg. (assuming American coins: 1, 5, 10, and 25 cents)

```
makeChange(1) = 1 (1)
makeChange(6) = 2 (5 + 1)
makeChange(49) = 7 (25 + 10 + 10 + 1 + 1 + 1 + 1)
```

## First solution

We will start by finding the first solution to this problem. For any problem where you are asked to find the most/least/largest/smallest etc, an excellent technique is to compare every possible combination. Although it will be inefficient, efficiency is not the most important current consideration and a solution of that nature is easy to make dynamic.

We can easily write a recursive function to find every possible combination of coins (***fig. 7***). At each recursive step, the solution can be broken into subproblems. If a 25¢ coin is selected, how many coins will be required to compose the remaining quantity?

```
// Brute force solution. Go through every
// combination of coins that sum up to c to
// find the minimum number
private int[] coins = new int[]{10, 6, 1};
public int makeChange(int c) {
  if (c == 0) return 0;
  int minCoins = Integer.MAX_VALUE;
  // Try removing each coin from the total and
  // see how many more coins are required
  for (int coin : coins) {
    // Skip a coin if it's value is greater
    // than the amount remaining
    if (c - coin >= 0) {
     int currMinCoins = makeChange(c - coin);
      if (currMinCoins < minCoins)
        minCoins = currMinCoins;
    }
  }
  // Add back the coin removed recursively
  return minCoins + 1;
}
```

*Fig 7. Naive Making Change solution*

A common, efficient solution to this problem is to use a greedy algorithm. In this situation, you repeatedly select the largest coin that isn't larger than the remaining amount of change (more info here). However, this doesn't work for arbitrary combinations of coins (consider 1¢, 6¢, and 10¢ coins and using a greedy algorithm to compute `makeChange(12)`). Therefore, we will instead focus on a generalized solution.

## Analyze the first solution

It is not surprising that the first iteration of our code is relatively inefficient. Since we are looking at every possible combination of coins to find the best one, we have to look at many different combinations.

As we can see from the execution of `makeChange(12)` (*fig. 8*), our tree will have a maximum height of `c` and branch `n` different ways at each level, where `n` is the number of different coins. This means that our big O time complexity will be `O(c^n)`.

Since the runtime is very poor and our solution is recursive, let's consider whether we can use dynamic programming.

1. **Optimal substructure.** As we've seen, recursion is a pretty good heuristic in this case. It is also possible to use the commutative and associative properties of addition to see that by finding the minimum number of coins for subproblem, it can be combined into the larger problem.
2. **Overlapping subproblems.** This property is often easiest to visualize by drawing a diagram and seeing if there is overlap between the multiple branches of the tree. While not shown in this diagram, we know that `makeChange(11)` will be broken into `makeChange(1)`, `makeChange(5)`, and `makeChange(10)`. `makeChange(1)` and `makeChange(5)` are called in other branches, so we know there's overlap.

With those properties, we can continue using the FAST method to find a dynamic programming solution for this problem.

```
// Brute force solution. Go through every
// combination of coins that sum up to c to
// find the minimum number
private int[] coins = new int[]{10, 6, 1};
public int makeChange(int c) {
  if (c == 0) return 0;
  int minCoins = Integer.MAX_VALUE;
  // Try removing each coin from the total and
  // see how many more coins are required
  for (int coin : coins) {
    // Skip a coin if it's value is greater
    // than the amount remaining
    if (c - coin >= 0) {
     int currMinCoins = makeChange(c - coin);
      if (currMinCoins < minCoins)
        minCoins = currMinCoins;
    }
  }
  // Add back the coin removed recursively
  return minCoins + 1;
}
```

*Fig 7. Repeated*

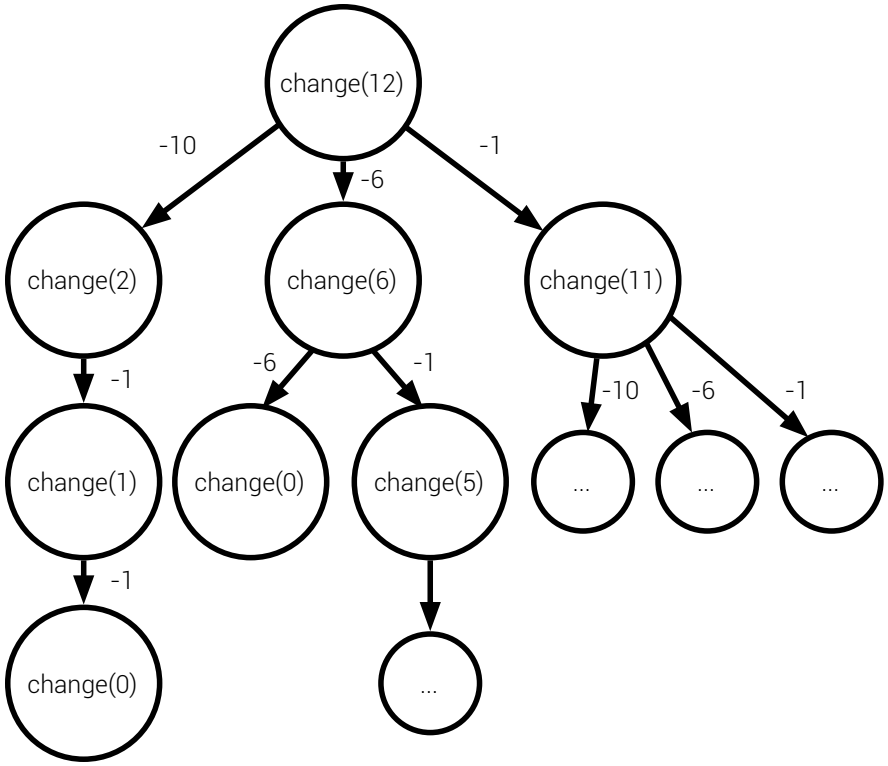



*Fig 8. Tree of recursive calls for* `makeChange(12)` *with* `coins={10, 6, 1}`

## Find the subproblems

Each function calls itself recursively once for each coin. These recursive calls are the subproblems because they break down our original input into smaller components and calculate those respective values.

The meaning of these subproblems is relatively easy to understand because it is identical to the meaning of the original problem. `makeChange(c)` for any value of `c` simply returns the minimum number of coins required to make `c` cents. Therefore, in our solution, we know that `makeChange(c - coin)` is simply the minimum number of coins to make `c-coin` cents.

Based on this understanding, we can turn our solution into a top-down dynamic solution. We can cache the results as they are computed. That means that we will cache the minimum number of coins needed to make various smaller amounts of change.

Like the Fibonacci problem, our code doesn't actually have to change very much. It's only necessary to overload our function with another that can initialize the cache. Then we update the original function in order to check the cache before doing the computation and saving the result to the cache afterwards (*fig. 9*).

The execution tree on the next page shows that the changes significantly improve the solution (*fig. 10*). We are only using `O(c)` space, even with the recursive stack.

The time complexity is a bit more complicated, but it can be estimated. There will be at most `c` calls that don't hit the cache. The number that do hit the cache is proportional to the number of coins (the branching factor). Therefore, we can estimate our complexity at `O(c * n)`.

```
// Top down dynamic solution. Cache the values
// as we compute them
private int[] coins = new int[]{10, 6, 1};

public int makeChange(int c) {
  // Initialize cache with values as -1
  int[] cache = new int[c + 1];
  for (int i = 1; i < c + 1; i++)
  cache[i] = -1;
  return makeChange(c, cache);
}

// Overloaded recursive function
private int makeChange(int c, int[] cache) {
  // Return the value if it's in the cache
  if (cache[c] >= 0) return cache[c];

  int minCoins = Integer.MAX_VALUE;

  // Find the best coin
  for (int coin : coins) {
    if (c - coin >= 0) {
      int currMinCoins =
        makeChange(c - coin, cache);
        if (currMinCoins < minCoins)
          minCoins = currMinCoins;
    }
  }

  // Save the value into the cache
  cache[c] = minCoins + 1;
  return cache[c];
}
```

*Fig 9. Top-down dynamic Making Change solution*

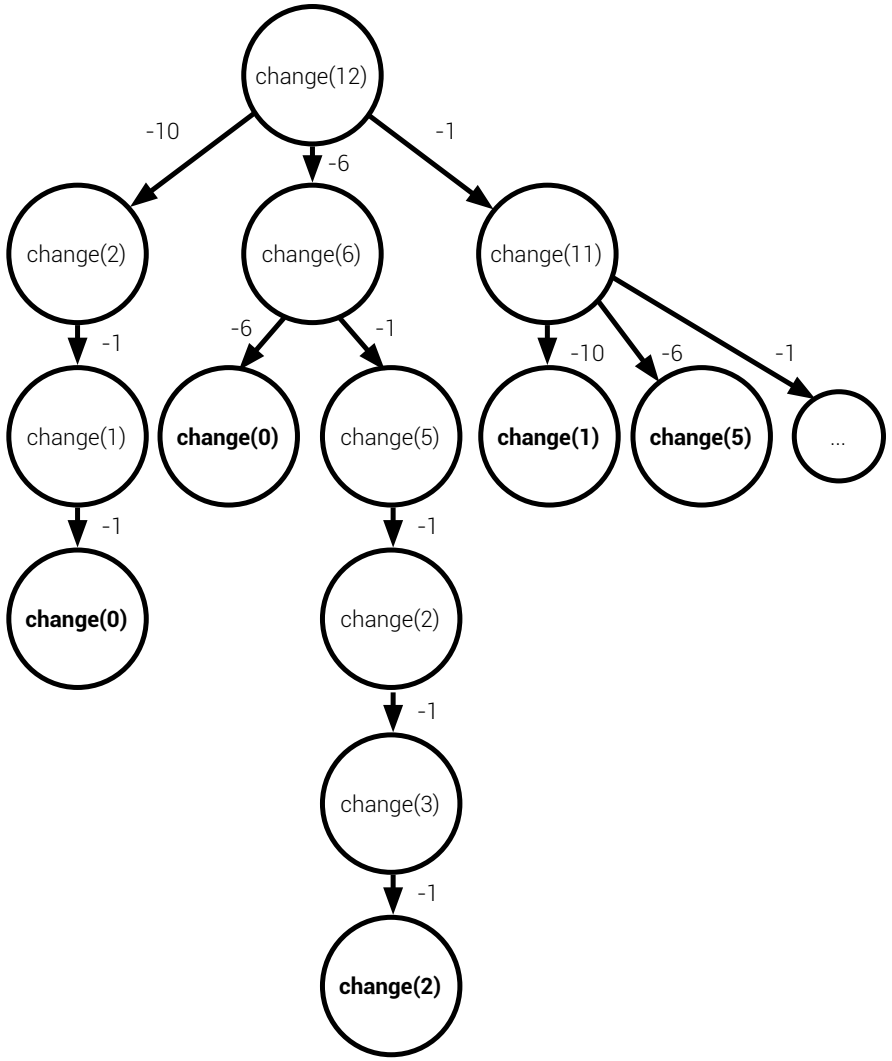



*Fig 10. Tree of recursive calls for* `makeChange(12)` *with* `coins={10, 6, 1}`. *Cache hits are bolded*

## Turn the solution around

Once the top-down solution is completed, it's possible to flip it around. We do this by solving the same subproblems in reverse order. Rather than starting with our result in mind, we start with no change and work our way up until we reach the solution.

The next step is to determine the subproblems that must be solved, in order to solve successive subproblems. If we want to compute `makeChange(c)`, then we will have `n` different subproblems. If our coins are `{10, 6, 1}`, we need to have the solutions for `makeChange(c - 10)`, `makeChange(c - 6)`, and `makeChange(c - 1)`.

Once `makeChange()` is solved for `0` through `c - 1`, it will be easy to compute the value of `makeChange(c)`. This is done by using the first value, `0` as our base case. We can then compute the remaining values from the previously computed values.

By looking at the code on the next page we can see that we are using `O(c)` space and `O(c * n)` time. This is comparable to the top-down solution.

Although it is otherwise comparable, this code is much simpler than the top-down solution. It's much easier to read and test. However, the understanding developed from working through this problem makes it easier to understand what this code is really doing.

```
// Bottom up dynamic programming solution.
// Iteratively compute number of coins for
// larger and larger amounts of change
public int makeChange(int c) {
  int[] cache = new int[c + 1];
  for (int i = 1; i <= c; i++) {
    int minCoins = Integer.MAX_VALUE;

    // Try removing each coin from the total
    // and see which requires the fewest
    // extra coins
    for (int coin : coins) {
      if (i - coin >= 0) {
        int currCoins = cache[i-coin] + 1;
        if (currCoins < minCoins) {
          minCoins = currCoins;
        }
      }
    }
    cache[i] = minCoins;
  }

  return cache[c];
}
```

*Fig 11. Bottom-up dynamic Making Change solution*

## Conclusion

Although Making Change is slightly more complicated than where we started with the Fibonacci problem, it is still an excellent example of how to use dynamic programming.

Both of these problems have had very straightforward subproblems. Their meaning is very clear and doesn't require much thought. However, in the remaining three problems, defining the subproblems correctly will become much more important.

# Square Submatrix

Given a 2D boolean array, find the largest square subarray of true values. The return value should be the side length of the largest square subarray subarray.

eg.

`arr =`

| | | | |
|---|---|---|---|
| False | True | False | False |
| True | True | True | True |
| False | True | True | False |

`squareSubmatrix(arr) = 2`

## First solution

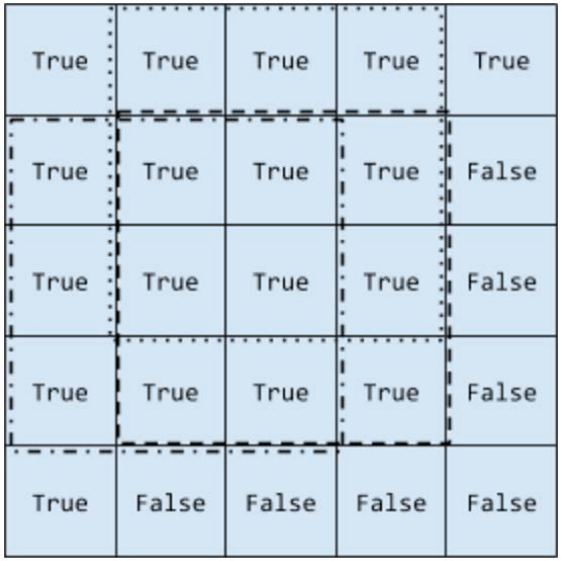
This may be the hardest problem in this ebook to get started with initially. It is easy to find a brute force solution that checks every possible square subarray to see if it contains all true values. However, that solution isn't easily broken down into subproblems. Therefore, in order to do this dynamically, we need to explore this further.

Really, what we want to do is to iterate through our array in order to find the biggest square subarray that contains each cell. Since we do not want to keep looking at the same subarray, we want to ask this question: What is the biggest square subarray for which the current cell is the upper left-hand corner?

While it may seem to be okay to do this iteratively, reframing the problem in this way allows us to think of it recursively in terms of subproblems.

Consider the figure on the next page (***fig. 12***). If our current cell is `arr[0][0]`, we find that the cells `arr[0][1]`, `arr[1][0]`, and `arr[1][1]` are each the upper left-hand corner of their own respective `3x3` subarrays. With that knowledge, we can see our current cell, `arr[0][0]`, is the only cell missing in a subarray of the next size larger.

We can generalize based on this realization. If a given cell is `true`, then it is the upper lefthand corner of the minimum size of the three subarrays to the bottom, right, and bottom-right. It is, therefore, possible to implement this recursively. This can be done by iterating through each cell and recursively finding the size of the largest square subarrays to the bottom, right, and bottom-right, and combine it to get our solution (***fig. 13***).



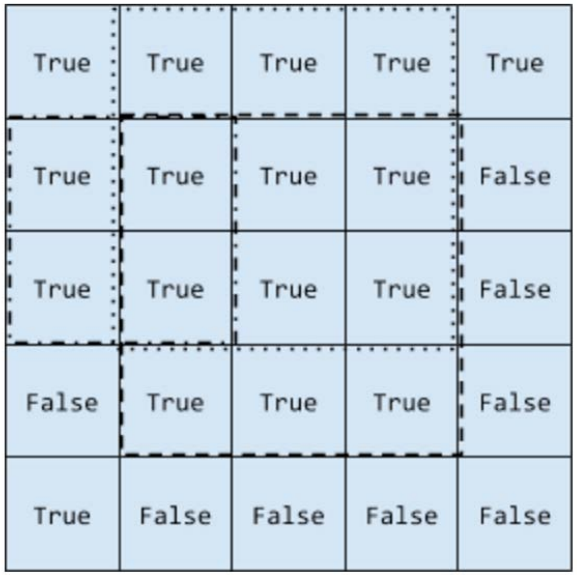




*Fig 12. Considering* `arr[0][0]` *is* `true`*, what is the largest submatrix we can make with the child submatrix?*

```
// Brute force solution. From each cell
// find the biggest square submatrix for which
// it is the upper left-hand corner
public int squareSubmatrix(boolean[][] arr) {
  int max = 0;
 // Compute for each cell the biggest subarray
  for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[0].length; j++) {
      if (arr[i][j]) max =
     Math.max(max,squareSubmatrix(arr,i,j));
    }
  }
  return max;
}

// Overloaded recursive function
private int squareSubmatrix(boolean[][] arr,
                            int i, int j) {
 // Base case at bottom or right of the matrix
  if (i == arr.length || j == arr[0].length)
    return 0;
  // If the cell is false then it's not part
  // of a valid submatrix
  if (!arr[i][j]) return 0;
  // Find the size of the right, bottom, and
  // bottom right submatrices and add 1 to the
  // minimum of those 3 to get the result
  return 1 + Math.min(Math.min(
                squareSubmatrix(arr, i+1, j),
               squareSubmatrix(arr, i, j+1)),
             squareSubmatrix(arr, i+1, j+1));
}
```

*Fig 13. Naive Square Submatrix solution*

This solution may seem to be complicated or that you need to just see it. Although this is generally true, the technique of using geometric properties occurs frequently. It is possible to use it, if needed.

## Analyze the first solution

By now, you're probably getting used to the routine of how the FAST method can be used. The current time solution is very poor and it is helpful to understand how bad it is.

In order to evaluate the runtime of this code, we can examine our recursion at a high level. With each turn, we make three recursive calls. Therefore, we branch by `3` each time and get a runtime of `3 * 3 * 3 * … or 3^x`. In this case, `x` is the depth of our recursion. Since in each turn, we either go down or left in our matrix, we can find that the maximum depth of our recursion is `n + m` for an `n` by `m` matrix. In this solution, we also have to do our recursive call for each of the `n * m` cells. We, therefore, get a runtime of `O(n * m * 3 ^ (n + m))`.

In terms of space complexity, our solution is simple because the only space we use is the recursive stack. Therefore, we get a space complexity of `O(n + m)`.

Since that solution was so bad, let's see if it can be improved with dynamic programming.

1. **Optimal substructure.** Although our original solution of just looking at every possibility didn't match this criteria, our new recursive solution does. By finding the maximum size of three smaller subarrays, we can find the maximum size of the larger subarray.
2. **Overlapping subproblems.** Without the recursive tree, it is more difficult to see this. However, if we think about how we execute our code, it is clear that we definitely have overlapping subproblems. Since we have to iterate over our array and continuously repeat our recursion, we are guaranteed to recompute subproblems that we have already computed.

We now know that we can definitely benefit from dynamic programming.

## Find the subproblems

Although we have already discussed this, we need to explicitly define our subproblems. Based on our definition of our recursion, we know that for any values of `i` and `j`, the function returns the largest square subarray of all true values with `arr[i][j]` as the upper left-hand corner. This is arbitrarily true for any values of `i` and `j`.

A closer look at the recursive function indicates that the function is being called recursively with different values of `i` and `j` and the same value of `arr`. Therefore, we know that our result is dependent on the values of `i` and `j`, but not `arr`, since `arr` stays the same for the duration of any given execution.

With this knowledge of the subproblems, we can cache our results and to look them up by `i` and `j`. To do this, it makes sense for us to use a 2D array as our cache. We can again easily modify our old solution by simply checking the cache before performing a computation and putting any computed values into the cache at the end (***fig. 14***).

Adding a cache allows us to significantly improve our time complexity. Now we only need to recursively compute each value once. Therefore, we visit each cell a constant number of times. We now get a time complexity of `O(n * m)` and a space complexity of `O(n * m)`, because we have to store the cache.

```
// Top down dynamic programming solution. Cache
// the values to avoid repeating computations
public int squareSubmatrix(boolean[][] arr) {
  // Initialize cache. Don't need to initialize
  // to -1 because the only cells that will be
  // 0 are ones that are false and we want to
  // skip those ones anyway
  int[][] cache =
    new int[arr.length][arr[0].length];
  int max = 0;
  for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[0].length; j++) {
      if (arr[i][j]) max = Math.max(max,
          squareSubmatrix(arr, i, j, cache));
    }
  }
  return max;
}
// Overloaded recursive function
private int squareSubmatrix(boolean[][] arr,
              int i, int j, int[][] cache) {
  if (i == arr.length || j == arr[0].length)
    return 0;
  if (!arr[i][j]) return 0;

  // If the value is set in the cache return
  // it. Otherwise compute and save to cache
  if (cache[i][j] > 0) return cache[i][j];
  cache[i][j] = 1 + Math.min(Math.min(
      squareSubmatrix(arr, i+1, j, cache),
      squareSubmatrix(arr, i, j+1, cache)),
    squareSubmatrix(arr, i+1, j+1, cache));
  return cache[i][j];
}
```

*Fig 14. Top-down dynamic Square Submatrix solution*

## Turn the solution around

Since the top-down solution is now complete, we can flip the solution on it's head. Remember that our subproblems were the largest square submatrix with the upper left-hand corner at a given location `(i, j)`. This can be solved easily by starting with the smallest subproblems.

However, it is first necessary to slightly modify our subproblems. This is likely to happen whenever we recurse through an array/matrix. Since recursion recurses to the end before working its way up, it traverses the array backwards. It starts solving subproblems with the bottom right-hand corner of the array.

Although it is fine for recursion, it makes iteration confusing. Therefore, we can reverse the subproblem so that rather than being the upper left-hand corner of a subarray, each subproblem taking `(i, j)` represents the bottom right-hand corner of the largest subarray.

This makes it easy to build up our solution (***fig. 15***). We can start with `i=0`, `j=0` and simply solve each successive subproblem, until we get to `i=n`, `j=m`. We then have a cache that is full of the max sizes of different bottom left-hand corners. This allows us to iterate over and pick the largest. It is also possible to keep track of the maximum as we go, to avoid having to iterate over the whole thing again.

All we have to do now is iterate over the entire matrix once and do a constant-time operation for each cell. This gives us a runtime complexity of `O(n * m)`. We also get a space complexity of `O(n * m)`, because we have to store the results in an `nxm` matrix.

```
// Bottom up solution. Start from the
// upper left corner and compute each larger
// submatrix
public int squareSubmatrix(boolean[][] arr) {
  int max = 0;
  // Initialize cache
  int[][] cache =
    new int[arr.length][arr[0].length];
  // Iterate over matrix to compute each value
  for (int i = 0; i < cache.length; i++) {
    for (int j = 0; j < cache[0].length; j++) {
      // If we're in the first row/column then
      // the value is just 1 if that cell is
      // true and 0 otherwise. In other rows and
      // columns need to look up and to the left
       if (i == 0 || j == 0) {
         cache[i][j] = arr[i][j] ? 1 : 0;
       } else if (arr[i][j]) {
         cache[i][j] =
           Math.min(Math.min(cache[i][j-1],
                             cache[i-1][j]),
                    cache[i-1][j-1]) + 1;
       }
      if (cache[i][j] > max) max = cache[i][j];
    }
  }
  return max;
}
```

*Fig 15. Bottom-up dynamic Square Submatrix solution*

## Conclusion

This problem is difficult because it can be challenging to break it into subproblems. However, with some creativity, it is straightforward after you codify it into a recursive function. This is also the first problem where the output is dependent on multiple inputs (`i` and `j`). We will see that come up again in the next two problems.

# 0-1 Knapsack

Imagine that you have a knapsack which can carry a certain maximum amount of weight and you have a set of items with their own weight and a monetary value. You are going to to sell your items in the market but you can only carry what fits in the knapsack. How do you maximize the amount of money that you can earn?

Or more formally...

Given a list of items with values and weights, as well as a max weight, find the maximum value you can generate from items, where the sum of the weights is less than or equal to the max.
eg.

```
items = {(w:2, v:6), (w:2, v:10), (w:3, v:12)}
max weight = 5
knapsack(items, max weight) = 22
```

## First solution

The first step is to try to find a brute force solution to this problem. One way of solving this type of problem where you're trying to find the most of something, is to find every possible combination of items to find the one with the maximum value and is under the max weight.

This can be done with a straightforward recursive function. We will be slightly clever and instead of looking at every possible combination, we limit our focus to combinations that are less than the maximum weight.

This can be done by recursively iterating over all the items. If the item isn't too heavy to fill the remaining space in the bag, we recursively compute the max value of both, including and not including the current item. If it is too heavy, our only option is to not include it and to continue on to the next item (***fig. 16***).

```
public class Item {
    int weight;
    int value;
}
// Naive brute force solution. Recursively
// include or exclude each item to try every
// possible combination
public int knapsack(Item[] items, int W) {
    return knapsack(items, W, 0);
}
// Overloaded recursive function
private int knapsack(Item[] items, int W, int
i)
  {
  // If we've gone through all the items,
  // return
  if (i == items.length) return 0;
  // If the item is too big to fill the
  // remaining space, skip it
  if (W - items[i].weight < 0)
    return knapsack(items, W, i+1);

  // Find the maximum of including and not
  // including the current item
  return Math.max(
    knapsack(items, W - items[i].weight, i+1)
                + items[i].value,
    knapsack(items, W, i+1));
}
```

*Fig 16. Naive 0-1 Knapsack solution*

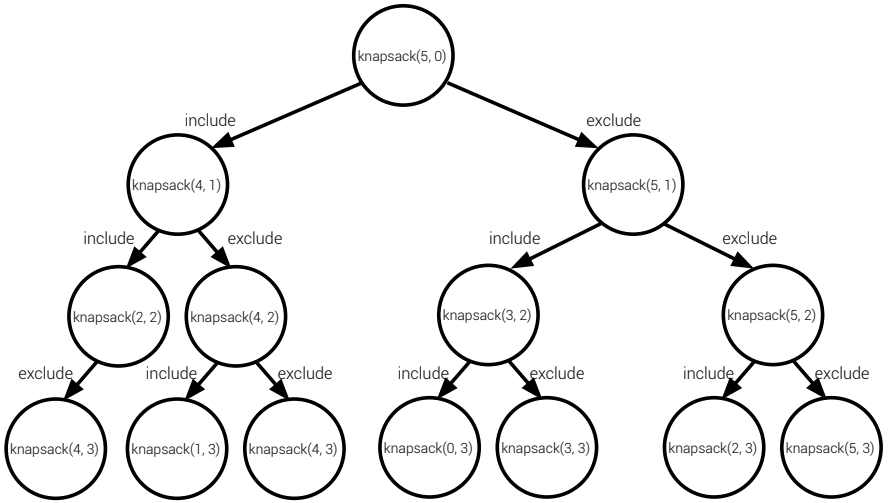



*Fig 17. Tree of recursive calls for* `knapsack(W, i)` *where* `W = 5, i = 0` *and* `items={(w:2, v:6), (w:2, v:10), (w:3, v:12)}`

## Analyze the first solution

Our code looks at every possible combination, which is not very efficient. The question that must be asked is: how bad is it?

Each item can be included it or not included. This causes the recursion to branch in two different ways. Like several other problems that have been discussed, we get `2 * 2 * … * 2` or `2^n`, where `n` is the depth of our recursion. In this case, our recursion iterates over the items array. Therefore, `n` is the number of items, which gives us a time complexity of `O(2^n)`.

The code performs reasonably well in terms of space complexity. The only extra space that we are using is the recursive stack, which is at most, height `O(n)`.

Since there is recursion, this problem is a likely candidate to be improved by dynamic programming.

1. **Optimal substructure.** From our recursive solution, we know that this problem can be solved by combining the results of subproblems.
2. **Overlapping subproblems.** The diagram (*fig. 17*) shows that there is at least one overlapping subproblem. With these types of examples, it is sometimes not obvious how many subproblems actually overlap. However, if even one subproblem overlaps, we know that there will be more with bigger inputs and deeper levels of recursion.

Since there is an optimal substructure and overlapping subproblems, our solution will be made dynamic.

## Find the subproblems

This is a problem where the subproblems are not so straightforward. We need to look at the recursion in order to understand the real meaning of the recursive calls.

Our function includes a list of items, a weight and an index. We are always passing the same list of items and not modifying it in any way. Therefore, we know that for a given execution of this code, the solutions to our subproblems are dependent solely on the weight and index. Therefore, this discussion focuses on those two variables.

With each recursive call, we either subtract our current item's weight from the total weight or ignore it. Therefore, the weight that we're passing into our subproblem is the remaining available weight for additional items. The index increases with each call. Therefore, we know that any recursive call will only include the items from that index to the end of the list.

Therefore, our recursive call is asking this question: What is the maximum value of the items from `i` to `items.length` that weigh less than the given weight? It means that the initial call of `knapsack(items, W, 0)` is exactly the same, but includes all of the items.

Our understanding of the subproblems enables us to make our original solution dynamic (*fig. 18*). This is done by caching the values as we have done before. However, since it is possible that there will be considerable empty space in our cache, we can implement it using a HashMap of HashMaps (works the same as a 2D array, which is an array of arrays).

Looking at the code on a high level, the space complexity is `O(n * W)`, where `n` is the number of items. This is because our worst case is that we will need to store the subproblem for every `weight < W` and every item index. In terms of time complexity, we only have to compute our value once for any combination of `i` and `W`. This gives us a similar time complexity of `O(n * W)`.

```
// Top down dynamic programming solution.
// Cache values in a HashMap - the cache may
// be sparse
public int knapsack(Item[] items, int W) {
  // Map: i -> W -> value
  Map<Integer, Map<Integer, Integer>> cache =
    new HashMap<Integer,
                Map<Integer, Integer>>();
  return knapsack(items, W, 0, cache);
}

// Overloaded recursive function
private int knapsack(Item[] items, int W, int
i, Map<Integer, Map<Integer, Integer>> cache)
{
  if (i == items.length) return 0;
  // Check if the value is in the cache
  if (!cache.containsKey(i))
 cache.put(i,new HashMap<Integer,Integer>());
  Integer cached = cache.get(i).get(W);
  if (cached != null) return cached;
  // Compute the item and add it to the cache
  int toReturn;
  if (W - items[i].weight < 0) {
    toReturn = knapsack(items, W, i+1, cache);
  } else {
    toReturn =
      Math.max(knapsack(items,
                        W - items[i].weight,
                        i+1, cache)
                   + items[i].value,
               knapsack(items, W, i+1, cache));
  }
  cache.get(i).put(W, toReturn);
  return toReturn;
}
```

*Fig 18. Top-down dynamic 0-1 Knapsack solution*

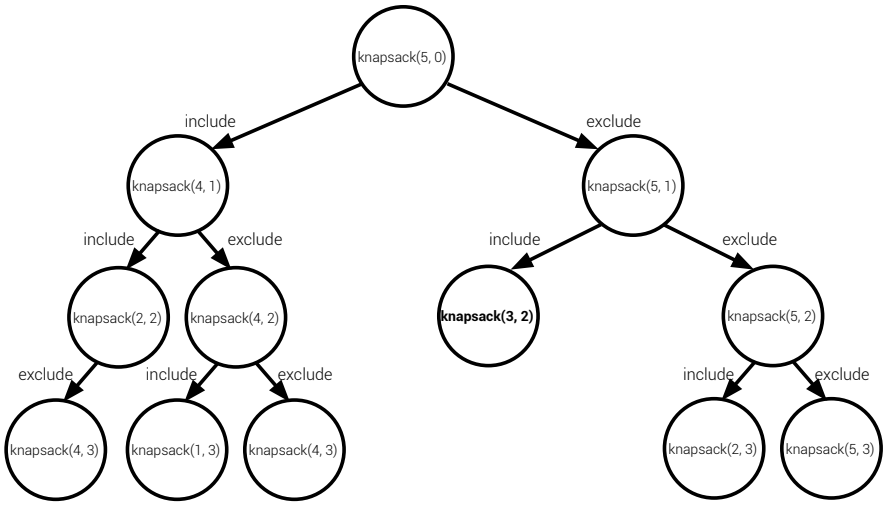



*Fig 19. Tree of recursive calls for* `knapsack(W, i)` *where* `W = 5, i = 0` *and* `items={(w:2, v:6), (w:2, v:10), (w:3, v:12)}`. *Cache hits are bolded*

## Turn around the solution

Since we took the time to fully understand the meaning of the subproblems, it's straightforward to flip our solution.

Similar to the Square Submatrix problem, our recursive solution iterates from the end of the array to the front. Therefore, we can start by flipping our subproblem to make it easier to deal with. We simply say that rather than `knapsack(items, W, i)` being the max value of the items including and after `i`, it will be the max value of the items up to, but not including `i` (or the first `i` items).

It is now easy to reason through how to solve our problem iteratively (*fig. 20*). For any pair where `i` or `W` is `0`, we get a max value of `0`, because if `i == 0` there are no items included. If `W == 0`, then no items may be included, because they will overflow the capacity.

When `i` and `W` do not equal `0`, we can determine our solution by looking at the previously computed subproblems. `i` tells us which item to consider and `W` tells us the remaining weight that we have available to us. For each cell, we need to decide whether we get a greater value at that weight by including or excluding item `i - 1`.

We can get the value of not including the item by looking at the solution for `W = W` and `i = i - 1`, since the value is the same as it was before. We can get the value of including the item by looking at the max value we could get for `W = W - item.weight`. This tells us the maximum value for the first `i` items, while still leaving sufficient empty space to include the current item.

This solution will be `O(n * W)` for both space and time complexity, which is identical in space and time complexity to our previous solution. We create an `nxW` array and iterate over the entire array.

```
// Iterative bottom up solution.
public int knapsack(Item[] items, int W) {
  // Initialize cache
  int[][] cache =
    new int[items.length + 1][W + 1];
  // For each item and weight, compute the max
  // value of the items up to that item that
  // doesn't go over W weight
  for (int i = 1; i <= items.length; i++) {
    for (int j = 0; j <= W; j++) {
      if (items[i-1].weight > j) {
        cache[i][j] = cache[i-1][j];
      } else {
        cache[i][j] = Math.max(cache[i-1][j],
            cache[i-1][j-items[i-1].weight] +
          items[i-1].value);
      }
    }
  }

  return cache[items.length][W];
}
```

*Fig 20. Bottom-up dynamic 0-1 Knapsack solution*

This problem can also be optimized in terms of space in the same way as with the Fibonacci problem. A close examination of our solution indicates that cache values are only looked up where `i = i - 1`. Therefore, we can reduce our solution to only using a 1D array (***fig. 21***).

This optimization does have limitations. With the original bottom-up solution, we were able to go through the array and figure out which items are included in the optimal knapsack. When we compress it into a single array, we are no longer able to do that. However, if you're just trying to solve for the maximum value, it isn't necessary to know the specific items.

## Conclusion

The 0-1 Knapsack problem is the quintessential example of a dynamic programming problem. A good understanding of how this problem works and how to formulate the subproblems makes the rest of dynamic programming relatively easy to understand.

Many dynamic programming problems are only variations of the 0-1 Knapsack problem. As you go through the next problem, pay attention to the similarities and if you can reframe the problem in terms of the 0-1 Knapsack problem.

```
// Optimized bottom up solution with 1D cache.
// Same as before but only save the cache of
// i-1 and not all values of i.
public int knapsack(Item[] items, int W) {
  int[] cache = new int[W + 1];
  for (Item i : items) {
    int[] newCache = new int[W + 1];
    for (int j = 0; j <= W; j++) {
     if (i.weight > j) newCache[j] = cache[j];
      else newCache[j] = Math.max(cache[j],
                          cache[j - i.weight] +
                             i.value);
    }
    cache = newCache;
  }

  return cache[W];
}
```

*Fig 21. Optimized bottom-up dynamic 0-1 Knapsack solution*

# Target Sum

Given an array of integers, `nums` and a target value `T`, find the number of ways that you can add and subtract the values in nums to add up to `T`.

eg.

```
nums = {1, 1, 1, 1, 1}
T = 3

1 + 1 + 1 + 1 - 1
1 + 1 + 1 - 1 + 1
1 + 1 - 1 + 1 + 1
1 - 1 + 1 + 1 + 1
-1 + 1 + 1 + 1 + 1

targetSum(nums, T) = 5
```

## First solution

To start our solution to this problem, we can consider all of the possible combinations of adding and subtracting items and then count the number of those combinations that add up to our target.

This can be done recursively by iterating through the array and recursively adding and subtracting each value from the total. Once we reach the end of the array, we check if the total is equal to the target sum. If it is equal to the target sum, then we have found one valid combination of adding and subtracting (***fig. 22***).

```
// Naive brute force solution. Find every
// combo
public int targetSum(int[] nums, int T) {
  return targetSum(nums, T, 0, 0);
}

// Overloaded recursive function
private int targetSum(int[] nums, int T, int
i,
                      int sum) {
  // When we've gone through every item, see
  // if we've reached our target sum
  if (i == nums.length) {
    return sum == T ? 1 : 0;
  }

  // Combine the possibilites by adding and
  // subtracting the current value
 return targetSum(nums, T, i+1, sum + nums[i])
 + targetSum(nums, T, i+1, sum - nums[i]);
}
```

*Fig 22. Naive Target Sum solution*

## Analyze the first solution

Similar to what we saw in several of the previous problems, our brute force solution is not very efficient. The recursive tree branches by `2` each time, which yields a runtime of `2 * 2 * … * 2` or `O(2^n)`, where `n` is the number of numbers. Our space complexity is the depth of the recursion. In this case, it is the length of our input, or `O(n)`.

Based on all the problems we've seen so far, this looks like a good candidate for dynamic programming. However, before making a decision, we need to look at the properties.

1. **Optimal substructure.** This problem, like the others, is solved recursively. This means that we're breaking it It is broken down into discrete subproblems with the results combined to obtain the solution.
2. **Overlapping subproblems.** The example tree (*fig. 23*) shows that there are multiple overlapping problems, even for this relatively simple example.

Hopefully at this point you're seeing the similarities between this and the 0-1 Knapsack problem are apparent. Adding or subtracting a number is similar to including or excluding an item in the knapsack. In this case, the total number of combinations is tracked instead of the items in the knapsack. It would require a minimal effort to modify the 0-1 Knapsack code to track, for example, the number of different combinations that are less than a certain weight.

```
// Naive brute force solution. Find every
// combo
public int targetSum(int[] nums, int T) {
  return targetSum(nums, T, 0, 0);
}

// Overloaded recursive function
private int targetSum(int[] nums, int T, int
i,
                      int sum) {
  // When we've gone through every item, see
  // if we've reached our target sum
  if (i == nums.length) {
    return sum == T ? 1 : 0;
  }

  // Combine the possibilites by adding and
  // subtracting the current value
 return targetSum(nums, T, i+1, sum + nums[i])
 + targetSum(nums, T, i+1, sum - nums[i]);
}
```

*Fig 22. Repeated*

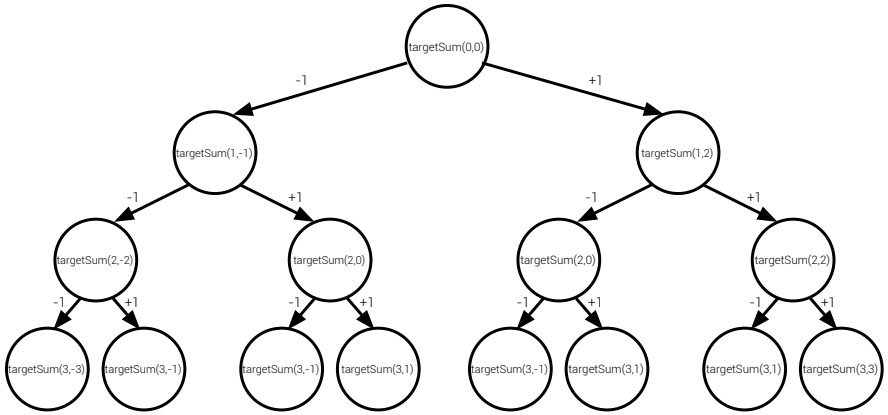



*Fig 23. Tree of recursive calls for* `targetSum(i, sum)` *where* `nums = {1, 1, 1}`

## Find the subproblems

In this problem, the subproblems are a little bit more complex. Values are passed into the recursion that don't actually change during the the problem execution. We can ignore `nums` and `T`, since those values don't actually change. We focus on only `i` and `sum`.

We're recursively calling `targetSum(nums, T, i+1, sum + nums[i])` and `targetSum(nums, T, i+1, sum - nums[i])`. This provides the basis for determining what the subproblems are. We know because we increment `i` every call that we are only looking at the values `>= i`, and `sum` is a running sum of numbers that we've added and subtracted from index `0` to `i`.

We can, therefore, define our subproblem as follows: `targetSum(nums, T, i, sum)` is the number of possible combinations of adding and subtracting the numbers at or after index `i`, where the sum of those numbers plus the sum equals `T`. It can also be stated as follows: The number of combinations where the sum equals `T - sum`.

Once the subproblem has been codified, the next step is to modify the original solution to cache these values (***fig. 24***). Similar to a 0-1 Knapsack, a HashMap of HashMaps will be used to ensure space efficiency.

In order to get the space complexity, we need to determine the range of the sum. This is because we are caching solutions for subproblems based on `i` and `sum`. An examination of our code shows that the sum ranges from `-sum(nums)` (if every value is subtracted) to `+sum(nums)` (if every value is added). This yields a space complexity of `O(i * sum(nums)`). The time complexity is exactly the same, since we only have to compute each value once.

```
// Top down dynamic programming solution. Like
// 0-1 Knapsack, we use a HashMap to save
// space
public int targetSum(int[] nums, int T) {
  // Map: i -> sum -> value
  Map<Integer, Map<Integer, Integer>> cache =
  new HashMap<Integer,Map<Integer,Integer>>();
  return targetSum(nums, T, 0, 0, cache);
}

// Overloaded recursive function
private int targetSum(
  int[] nums, int T, int i, int sum,
  Map<Integer, Map<Integer, Integer>> cache)
{
  if (i == nums.length) {
    return sum == T ? 1 : 0;
  }

  // Check the cache and return if we get a
  // hit
  if (!cache.containsKey(i)) cache.put(i,
      new HashMap<Integer, Integer>());
  Integer cached = cache.get(i).get(sum);
  if (cached != null) return cached;

  // If we didn't hit in the cache, compute
  // the value and store to cache
  int toReturn =
    targetSum(nums,T,i+1,sum+nums[i],cache) +
    targetSum(nums,T,i+1,sum-nums[i],cache);
  cache.get(i).put(sum, toReturn);
  return toReturn;
}
```

*Fig 24. Top-down dynamic Target Sum solution*

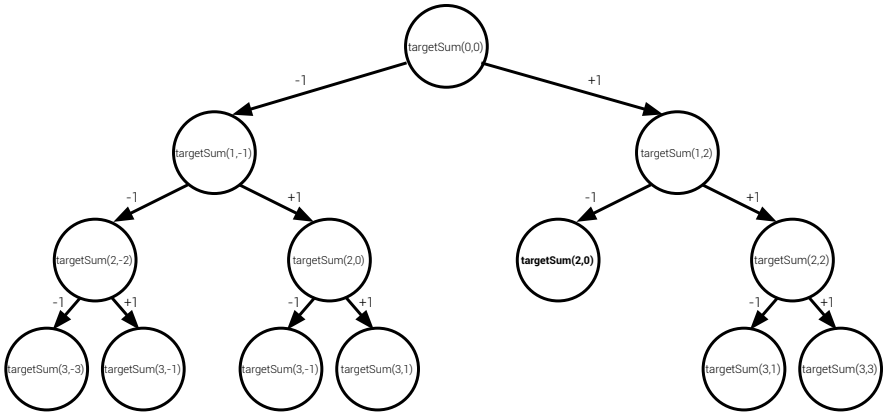



*Fig 25. Tree of recursive calls for* `targetSum(i, sum)` *where* `nums = {1, 1, 1}`. *Cache hits are bolded*

## Turn around the solution

The subproblems will again be reversed so that we can iterate through the array in forwards order. Our subproblems, therefore, become: `targetSum(nums, T, i, sum)` is the number of possible combinations of adding and subtracting the numbers before index `i` (or the first `i` numbers).

Now we can build up the subproblems. We need to be careful with how the subproblem solutions are stored. This is because some of the values can be negative and our cache array can't have negative indices. Therefore, the sum dimension of the cache will be size `2 * sum(nums) + 1`. The values will offset the values by sum. The `0` index actually represents `-sum(nums)`.

We can avoid the need for excessive bounds checking with a slight change in our approach. Instead of looking at the prior solutions to the subproblems, we will look at the subproblems that have already been solved and add those values to the next iteration.

Assume that there is a solution for the subproblem where `i = 3` and `sum = 10`. If the value at the next index is `4`, we know that we need to add our current value to the values at `i = 4`, `sum = 10 +/- 4`, or `i = 4` and `sum = 14` and `i = 4` and `sum = 6`. If this is done for all the values where `i = 3`, then we will have solved for the entire column where `i = 4`.

Our solution (*fig. 26*) gives us the same time and space complexity as the top-down solution. We are filling in a cache of size `i * sum(nums)` and each step takes constant time. Therefore, our time and space complexity are both `O(i * sum(n))`.

```
public int targetSum(int[] nums, int T) {
  int sum = 0;
  // Our cache has to range from -sum(nums) to
  // sum(nums), so we offset everything by sum
  for (int num : nums) sum += num;
  int[][] cache =
    new int[nums.length + 1][2*sum + 1];
  if (sum == 0) return 0;
  // Initialize i=0, T=0
  cache[0][sum] = 1;
  // Iterate over previous row and update the
  // current row
  for (int i = 1; i <= nums.length; i++) {
    for (int j = 0; j < 2 * sum + 1; j++) {
      int prev = cache[i-1][j];
      if (prev != 0) {
        cache[i][j - nums[i-1]] += prev;
        cache[i][j + nums[i-1]] += prev;
      }
    }
  }
  return cache[nums.length][sum + T];
}
```

*Fig 26. Bottom-up dynamic Target Sum solution*

## Conclusion

I hope you now have a better understanding of how to deal with these types of problems. The increased complexity of our problem requires a solution that can be trickier and more susceptible to errors. If our values are not offset by `sum(nums)` in the iterative solution, we will likely get an `IndexOutOfBounds` exception.

This discussion clearly shows how this problem closely mirrors the 0-1 Knapsack problem. This problem was intentionally chosen to because it is similar enough to be easy to spot, but different enough so that is possible to account for the differences. A clear understand of how this works, makes all dynamic programming much easier.

# Closing thoughts

Your main take away from this book should be that dynamic programming and coding interviews don't have to be hard. We worry about them excessively because we feel that there is so much pressure on us.

It's easy to feel like your life is over if you don't get a job at Google or Facebook. But that's the same as saying that if we don't get into Harvard, it's not worth going to college. It's clearly not true. However, many people hold themselves to ridiculously high standards that cannot possibly be achieved.

These misconceptions can hinder us when we are practicing for interviews. These feelings of inadequacy prevent us from succeeding and are something that we must deal with and change.

If interview prep is approached with structure and if we really understand what we're doing, we don't have to kill ourselves with excessive studying. We can follow a clear path forward to reach our ultimate goal of getting a great job. We do not need to be constantly worrying about whether we've memorized enough practice questions.

It is my hope that this book showed you how to take a structured approach which can make it easy to solve a wide array of dynamic programming problems.

With the FAST method in your toolkit, dynamic programming is one less thing you have to worry about. You can do this. You don't need to spend any more time here. You should turn your attention elsewhere. Take this same focus to other areas of your study and don't stress out.

You're going to do great!

## About the Author

Sam Gavis-Hughson is a former software engineer and founder of www.byte-by-byte.com. He has worked at several established tech companies, including HubSpot and Yext. His primary focus with Byte by Byte is to help current students and recent college graduates to find their dream jobs as software engineers. Between Yext and Byte by Byte, Sam has conducted over 50 interviews and mock interviews.

## About Byte by Byte

Byte by Byte is a site targeting current students and recent college graduates to help them find their dream jobs. Their mission is to help people succeed at interviewing by focusing on a structured approach to studying, as well as solving individual interview problems. They believe that interview prep doesn't need to be scary or time consuming. Anyone can be successful.

To learn more about Sam and Byte by Byte, visit www.byte-by-byte.com/about.

# Disclaimer