What is Dynamic Programming?

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. Dyanmic programming allows you to use more space to take less time.

The following 2 characteristics are required for 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 gett 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

Memoization

Memoization 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 (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 is -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.

We first try finding a top-down solution and then convert it to a bottom-up solution. Bottom-up solutions are not always better than topdown solutions. In an interview situation, although bottom-up solutions often result in more concise code, either approach is appropriate.

The distinction between top-down and bottom-up solutions will be discussed in detail in the upcoming examples. The important point is that top-down = recursive and bottom-up = iterative.

Dynamic Programming Practice

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.

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:

1. The recursive calls must be self-contained. That means no global variables.
2. You cannot do tail recursion. Your solution must compute the results to each subproblem and then combine them afterwards.
3. 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.

Fibonacci Numbers

Given an integer `n`, write a function that will return the nth Fibonacci number. e.g. `fib(0) = 0`, `fib(5) = 5`, `fib(10) = 55`

First Solution

The first step in our FAST method is to find the first possible solution. `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.

```
// We assume n >= 0
let fib = function (n) {
  if (n === 0) return 0;
  if (n === 1) return 1;
  return fib(n - 1) + fib(n - 2);
};
```

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)`

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 + 2^2 + ... 2^n-1` calls, which reduced to `O(2^n)`.

Image of Fib

That 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 (i.e. `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 of `fib(2)` only once.

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-1th 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.

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.

```
let fib = function (n) {
  if (n < 2) return n;
  // Create cache and initialize to -1
  let cache = new Array(n + 1).fill(-1);
  cache[0] = 0;
  cache[1] = 1;
  return fibHelper(n, cache);
};

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

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)`.

```
let fib = function (n) {
  let cache = [0, 1];
  for (let i = 2; i <= n; i++) {
    cache[i] = cache[i - 1] + cache[i - 2];
  }
  return cache[n];
};
```

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.

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.

```
let fib = function (n) {
  if (n < 2) return n;
  let n1 = 1;
  let n2 = 0;
  for (let i = 2; i < n; i++) {
    let n0 = n1 + n2;
    n2 = n1;
    n1 = n0;
  }
  return n1 + n2;
};
```

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.

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 (`makeChangeNaive`). 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?

```
let makeChangeNaive = function (c) {
  if (c === 0) return 0;

  let minCoins = Infinity;
  const COINS = [10, 6, 1];

  for (coin of COINS) {
    if (c - coin >= 0) {
      let currMinCoins = makeChangeNaive(c - coin);
      if (currMinCoins < minCoins) {
        minCoins = currMinCoins;
      }
    }
  }
  return minCoins + 1;
};
```

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)`, 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)`.

Image of Fib

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 a 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(5)` and `makeChange(1)` 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.

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.

```
let makeChangeDP = function (c) {
  let cache = [0];
  return makeChangeHelper(c, cache);
};

let makeChangeHelper = (c, cache) => {
  if (cache[c] >= 0) return cache[c];
  let minCoins = Infinity;
  const COINS = [10, 6, 1];

  for (coin of COINS) {
    if (c - coin >= 0) {
      let currMinCoins = makeChangeHelper(c - coin, cache);
      if (currMinCoins < minCoins) {
        minCoins = currMinCoins;
      }
    }
  }
  cache[c] = minCoins + 1;
  return cache[c];
};
```

Image of Fib

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

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.

```
let makeChangeIterative = function (c) {
  let cache = new Array(c + 1).fill(0);
  const COINS = [10, 6, 1];

  for (let i = 1; i <= c; i++) {
    let minCoins = Infinity;

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

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.

Image of Fib

First Solution

This might be the hardest problem on here to get started with initially. It's 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 really broken down into subproblems. Therefore, in order to do this dynamically, we need to explore further.

Really, what we want to do is 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.

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.

```
let squareSubMatrixNaive = (arr) => {
  let max = 0;
  // compute for each cell, the biggest subarray
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr[0].length; j++) {
      if (arr[i][j] === '1') {
        max = Math.max(max, squareSubMatrixNaiveHelper(arr, i, j));
      }
    }
  }
  return max;
};

let squareSubMatrixNaiveHelper = (arr, i, j) => {
  // Base case at bottom or right of the matrix
  if (i === arr.length || j === arr[0].length) return 0;

  // if the cell is '0', then its not part of a valid submatrix
  if (!arr[i][j] || arr[i][j] === '0') return 0;

  // Find the size of the right, bottom, and bottom right submatrices and
add 1 to the min of those 3 to get the result
  return (
    1 +
    Math.min(
      Math.min(squareSubMatrixNaiveHelper(arr, i + 1, j),
squareSubMatrixNaiveHelper(arr, i, j + 1)),
      squareSubMatrixNaiveHelper(arr, i + 1, j + 1)
    )
  );
};
```

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.

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

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.

```
let squareSubMatrixDP = (arr) => {
  let cache = new Array(arr.length).fill(0).map(() => new
Array(arr[0].length).fill(0));
  let max = 0;
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr[0].length; j++) {
      if (arr[i][j] === '1') {
        max = Math.max(max, squareSubMatrixDPHelper(arr, i, j, cache));
      }
    }
  }
  return max;
};

let squareSubMatrixDPHelper = (arr, i, j, cache) => {
  // Base case at bottom or right of the matrix
  if (i === arr.length || j === arr[0].length) return 0;

  // if the cell is '0', then its not part of a valid submatrix
  if (!arr[i][j] || arr[i][j] === '0') return 0;

  if (cache[i][j] > 0) return cache[i][j];
  // Find the size of the right, bottom, and bottom right submatrices and
add 1 to the min of those 3 to get the result
  cache[i][j] =
    1 +
    Math.min(
      Math.min(squareSubMatrixDPHelper(arr, i + 1, j, cache),
squareSubMatrixDPHelper(arr, i, j + 1, cache)),
      squareSubMatrixDPHelper(arr, i + 1, j + 1, cache)
    );

  return cache[i][j];
};
```

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. We can start with `i = 0` and `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's also possible to keep track of the max as we go.

All we have to do now is iterative 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 store the results in a `n*m` matrix.

```
let squareSubMatrixIterative = (arr) => {
  let max = 0;
  let cache = new Array(arr.length).fill(0).map(() => new
Array(arr[0].length).fill(0));

  for (let i = 0; i < cache.length; i++) {
    for (let j = 0; j < cache[0].length; j++) {
      // if we're in the first row or column, the value is 1 is the cell
is 1 else 0.
      if (i === 0 || j === 0) {
        cache[i][j] = arr[i][j] === '1' ? 1 : 0;
      } else if (arr[i][j] && arr[i][j] === '1') {
        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;
};
```

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).

0-1 Knapsack