Chapter 23
Dynamic Programming

23.1 Foundations of Dynamic Programming

※ 23.1.1 *Introduction to Dynamic Programming*

Dynamic programming is an algorithmic strategy that breaks a problem down into smaller subproblems and *stores the results of repeated subproblems* so that they are not recomputed if they are ever needed again. To understand the motivation behind dynamic programming, let's return to the example we used to introduce recursion back in chapter 5: suppose you are waiting in a very long line, and you want to know what position you are in. Since you cannot leave the line without losing your position, you make the assumption that the person in front of you already knows their position, so you ask them for their position and add one to get your own position. The person in front then makes the same assumption and asks the same question to the person in front of them. This process continues until the first person is reached, who knows that they are first in line. This person then tells the person behind them that they are second in line, who then tells the person behind them that they are third in line, and so on. Eventually, the person directly in front of you learns their position in line, and they turn around and tell you your own position. This real-life example demonstrates the recursive process, and is summarized using the following code:

```
1  int32_t get_line_position(int32_t person) {
2    if (person == 1) return 1;
3    else return 1 + get_line_position(person - 1);
4  } // get_line_position()
```

Now, let's suppose that the person behind you asks you for their position in line a few minutes later. If this were real life, you would immediately tell that person their position, since you already know your own position in line from asking before! However, notice that the recursive function above doesn't do this. Unlike you, the function doesn't remember that it has already made a recursive call to get its position in line, so it ends up making the same recursive call all over again! This is equivalent to you asking the person in front of you for your position *again*, even though you already did so a few minutes earlier. Assuming that the line hasn't changed, you end up going through the line again for no reason, wasting time to obtain a position number that you already knew from a previous recursive call.[1]

[1] In real life your position in line does eventually change, so this isn't the best example to use — in this case, just assume that the line doesn't move forward to better reflect actual recursive problems, where making a recursive call on the same input will always produce the same solution.

There are many different problems that exhibit this type of behavior, where a recursive call may be called multiple times on the exact same input throughout the lifetime of solving the problem. This is inefficient, since you are essentially computing the same thing multiple times (and each recursive call is often quite expensive). This is where dynamic programming comes into play! The idea behind dynamic programming is to *store the solutions of each recursive call so that they can be reused whenever the same recursive call is needed later down the line*. In the context of the previous example, this is equivalent to "remembering" your position in line so that you don't have to re-ask for it if you ever need it again.

※ 23.1.2 *Fibonacci Numbers*

To begin our exploration of dynamic programming, let's first take a look at one of the most common problems involving recursion: computing Fibonacci numbers.

Example 23.1 Given a positive integer `n`, write a function `fib(n)` that returns the n^{th} number in the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., where `fib(0) = 0`, `fib(1) = 1`, and every subsequent number is the sum of the previous two numbers in the sequence.

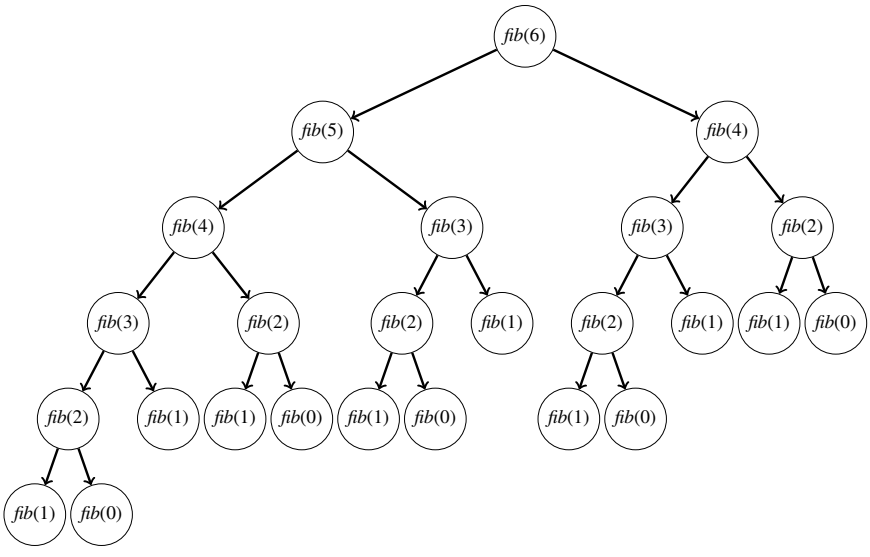
A recurrence relation for this problem can be derived in a straightforward manner: given any integer `n`, we can obtain the n^{th} Fibonacci number by simply summing up `fib(n-1)` and `fib(n-2)`.

$$fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n > 1 \end{cases}$$

Converting this recurrence relation to code, we get:

```
1  uint64_t fib(int32_t n) {
2    if (n == 0) return 0;
3    if (n == 1) return 1;
4    return fib(n - 1) + fib(n - 2);
5  } // fib()
```

If we run this function using small values of `n`, we are able to get the correct solution in a reasonable amount of time. However, if we were to try it with a larger number (such as `n = 50`), the function would take forever to run. To understand why this happens, let's take a look at exactly what is happening when `fib()` is called. The following tree depicts every recursive call that is made when `fib()` is invoked with `n = 6`:



At each node of the tree, we make two recursive calls and add up their return values, where each addition requires a constant amount of work. Since there are a total of $O(2^n)$ nodes in the tree, the total work required to solve for `fib(n)` can thus be represented as $O(2^n)$. This is why the function takes forever to run on larger inputs![2] Additionally, since `fib()` is not tail recursive and has a recursion depth of n, the auxiliary space required by this recursive implementation is $\Theta(n)$.

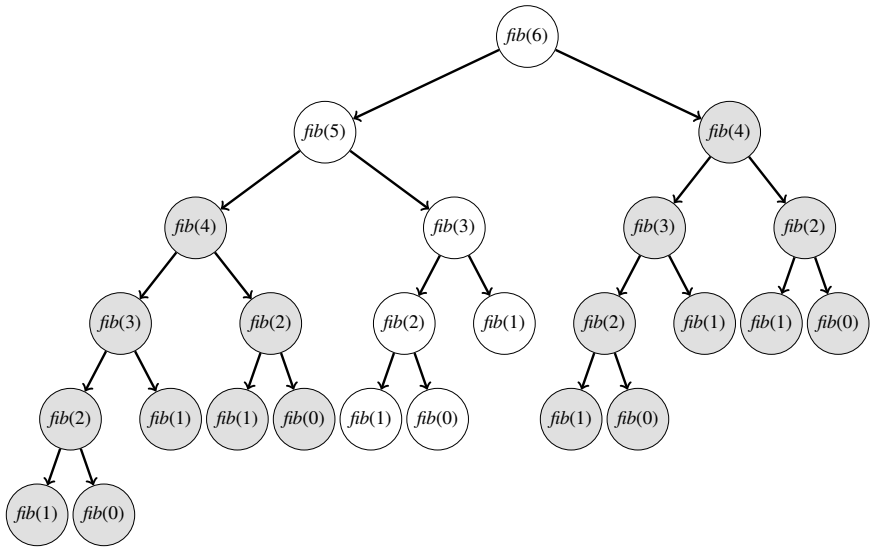
To return the Fibonacci number for a large value of n in a reasonable amount of time, we will need to reduce our algorithm's time complexity. To do so, we can either reduce the number of recursive calls we make, or we can reduce the amount of work that is done at each recursive call. Since each recursive call only performs a single addition operation, we cannot further reduce the work that is done at each recursive call. Therefore, our only option is to reduce the number of recursive calls that we make.

[2]This actually isn't the tightest possible bound for the this recursive `fib(n)` function, as the Fibonacci sequence has a closed form solution of

$$fib(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

which is $\Theta((1+\sqrt{5})/2)^n) \approx \Theta(1.618^n)$. Visually, you can see the actual time complexity is slightly better than $O(2^n)$ since one side of the tree is always larger than the other because the two recursive calls are not the same size. Also yes, you could technically write a $\Theta(1)$ solution using this closed-form solution, but that is beside the point since this isn't something we would reasonably expect you to do in this class.

Upon closer inspection of the recursion tree, there is a major issue with our naïve recursive approach: several identical recursive calls are completed *more than once*. For example, a single call to `fib(6)` ends up calling `fib(4)` twice before it returns.

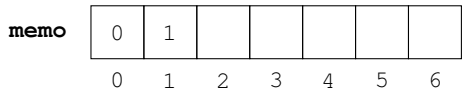


Similarly, `fib(3)` is called three times, and `fib(2)` five times. However, these additional calls are pointless, since you already know the solutions of these recursive calls. For larger values of n, this replicated work can end up being quite wasteful! It would be nice if our program could remember which recursive calls it has already made so that it never duplicates any work.

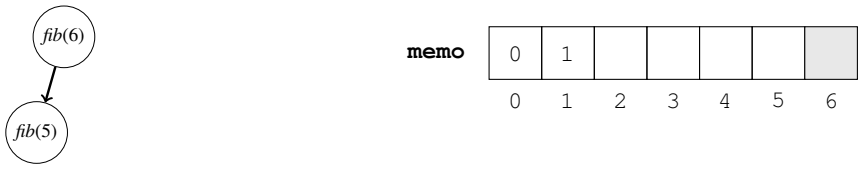
This is where the strategy of dynamic programming comes into play. When solving a problem using dynamic programming, you want to store the solutions to all the subproblems you encounter in a fast-access data structure such as a vector or a hash table. This container is known as a **memo**, which maps the *input arguments* of each subproblem to the solution of that subproblem. That way, if you ever encounter the same subproblem more than once, you can quickly retrieve its solution *in constant time* by using its input arguments to reference your memo. For example, the solution of `fib(4)` is 3, so your dynamic programming memo should map the input $n = 4$ to the output 3 (i.e., `memo[4] = 3`).

Typically, the dimensions of a memo should correspond to the number of arguments that are required to uniquely identify each subproblem or recursive call. In the Fibonacci example, each recursive call only accepts a single argument n, so our memo will be one-dimensional based on the value of n. However, there will be problems where a recurrence relation may involve multiple inputs within a single recursive call — for example, given the recurrence $F(m,n) = F(m-1,n) + F(m,n-1)$, your memo would need to be two-dimensional, one dimension for m and the other for n. That way, your memo will be large enough to support all possible input combinations up to m and n. We will look at some examples that require multidimensional memos later in this chapter.

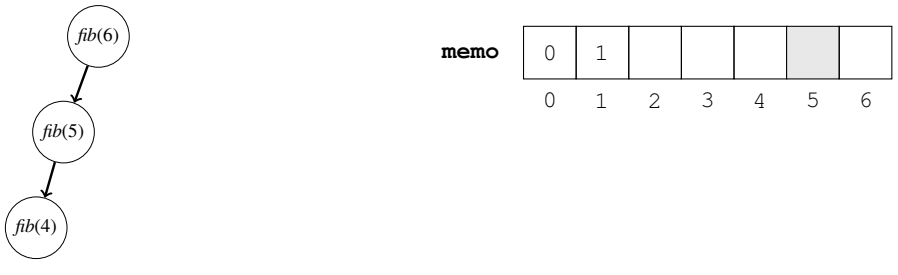
First, let's analyze the dynamic programming solution for the Fibonacci problem, with initial input size $n = 6$. To start, we want to initialize a memo that can be used to store the solutions to all subproblems we encounter, from `fib(0)` to `fib(n)`. This can be done using a vector of size 7, where each index `i` stores the value of `fib(i)`. We know that `fib(0)` and `fib(1)` are equal to 0 and 1, so we can fill out these values in the memo immediately (or directly return them in our recursive function).



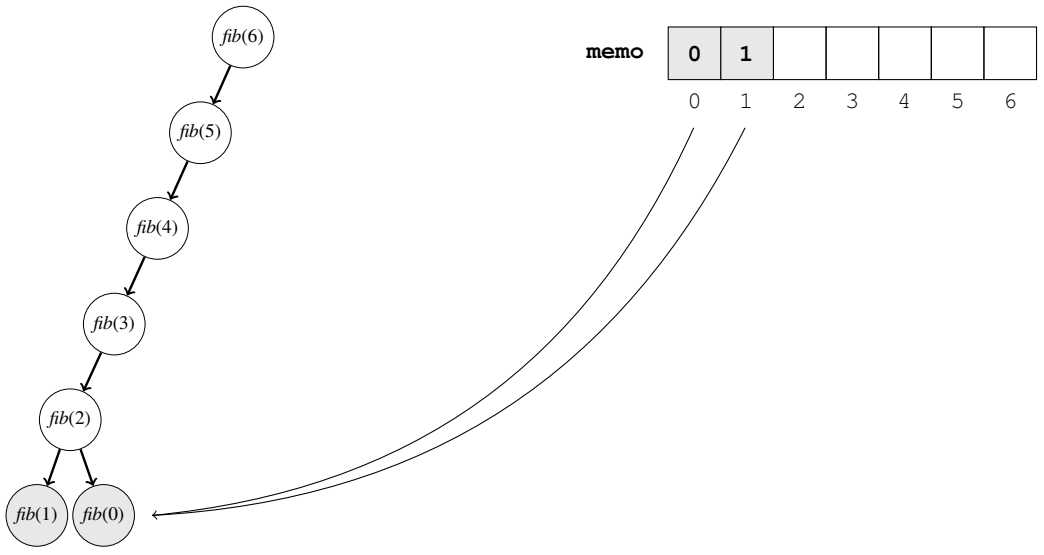
We start by making a recursive call to `fib(6)`. `memo[6]` is currently uninitialized, which indicates that we have never called `fib(6)` before, so we make a recursive call to `fib(6)`. This recursive call then makes a call to `fib(5)`.



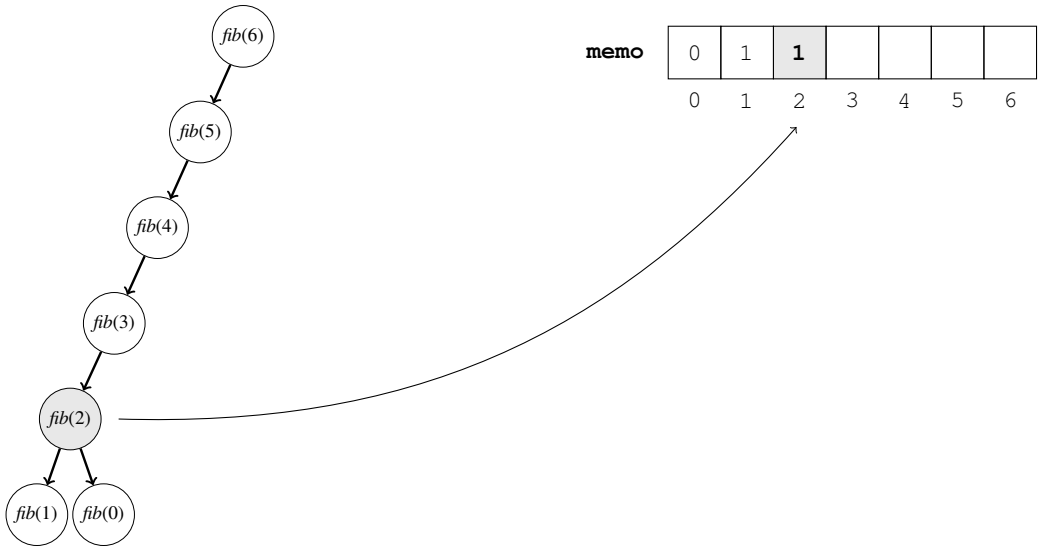
`memo[5]` is uninitialized, so we haven't called `fib(5)` before. We make a recursive call to `fib(5)`, which calls `fib(4)`.



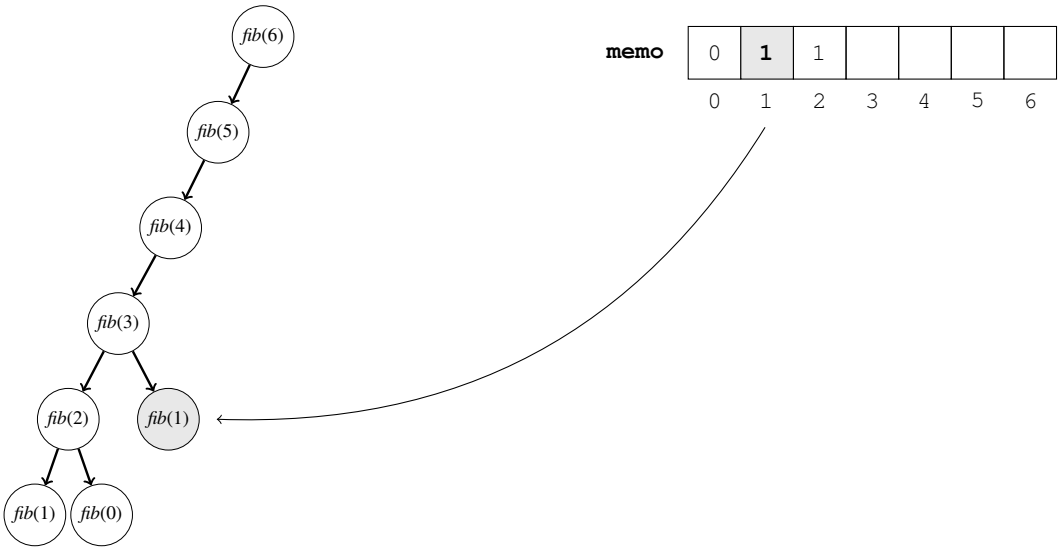
This process continues until we hit the base cases of `fib(1)` and `fib(0)`. These subproblems can be solved trivially, so we simply return 1 and 0, respectively.



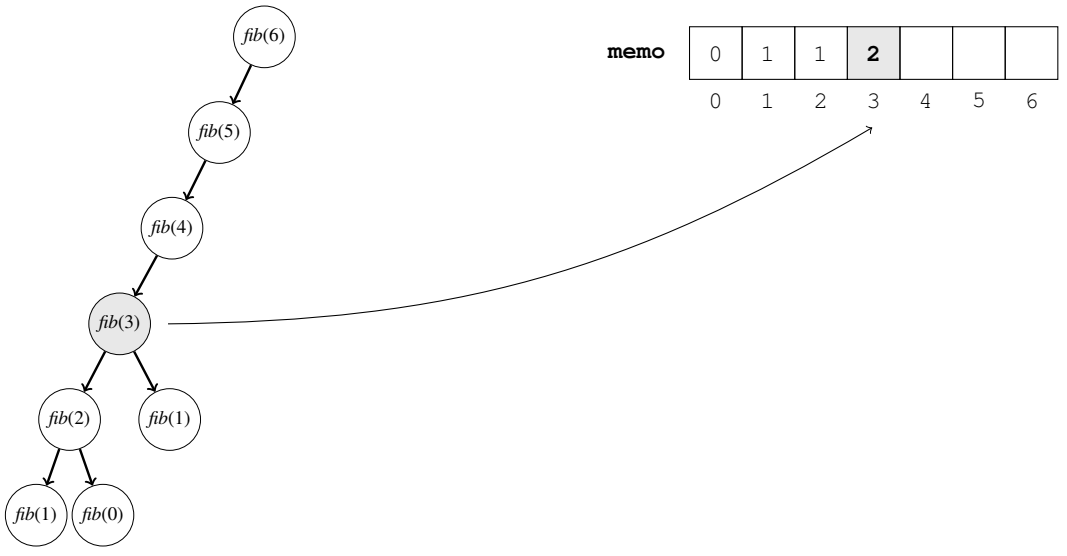
The recursion unrolls back to the `fib(2)` stack frame, which sums up the results of `fib(1)` and `fib(0)` to get a value of $1 + 0 = 1$. Thus, `fib(2)` has a return value of 1, which we store at position 2 of the memo.



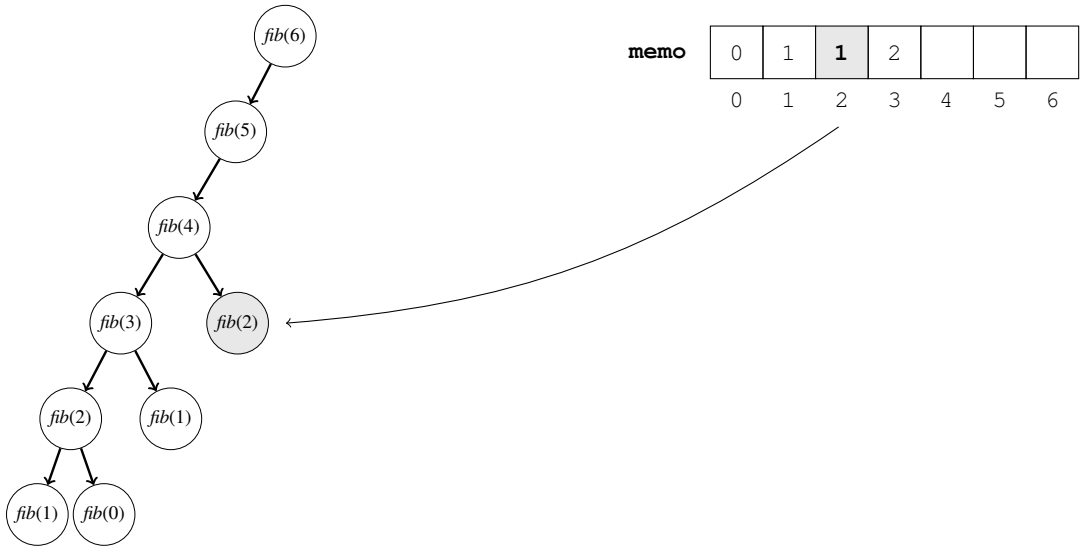
The recursion unrolls back to `fib(3)`, which then makes a recursive call to `fib(1)`. This is a base case, so we return 1.



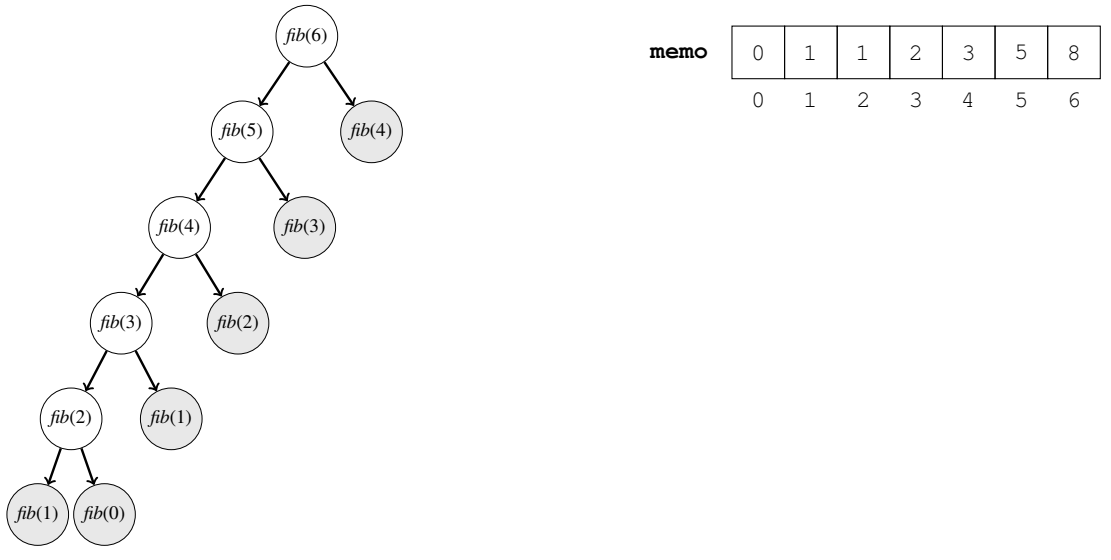
`fib(3)` then sums up its two values to get $1 + 1 = 2$. Before `fib(3)` returns 2, it stores its solution at position 3 of the memo.



The recursion unrolls back to `fib(4)`, which then makes a recursive call to `fib(2)`. However, `fib(2)` has already been computed before, so we can just take its value from the memo.



This process repeats until we reach our initial stack frame of `fib(6)`, which returns 8.



The shaded nodes of the previous tree represent recursive calls we were able to trivially solve, either from returning a base case or from pulling a solution directly from the memo. By getting rid of duplicate recursive calls, we were able to bring the time complexity down from exponential to linear. The code for this dynamic programming implementation is shown below. Since we already know that `fib(0)` and `fib(1)` return 0 and 1, we can initialize them in the memo immediately. We then call a helper function that performs the work of solving each subproblem and storing its solution in the memo.

```
uint64_t fib(int32_t n) {
  std::vector<uint64_t> memo(n + 1);
  return fib_helper(n, memo);
} // fib()

uint64_t fib_helper(int32_t n, std::vector<uint64_t>& memo) {
  if (n == 0 || n == 1) {
    return n;
  } // if
  if (memo[n] != 0) { // already solved before, fetch from memo
    return memo[n];
  } // else if
  // solve and store in memo
  return memo[n] = fib_helper(n - 1, memo) + fib_helper(n - 2, memo);
} // fib_helper()
```

※ 23.1.3 *Top-Down and Bottom-Up Dynamic Programming*

This process of storing the results of recursive calls so that they can be used later is formally known as **memoization**. Memoization can often be implemented without significant changes to an existing recursive solution. To memoize a recursive function:

- On function entry:
  1. Query the memo using the function input(s) to determine if a recursive call has been made on that input before.
  2. If the input has been called before, retrieve the result from the memo instead of recomputing with a recursive call.
- On function exit:
  1. Save the result of the recursive call in the memo, in the position corresponding to the function input(s).

Thus, memoization solutions often exhibit a structure similar to a naïve recursive approach, as shown below in pseudocode:

```
function dp_helper(dp_state, memo):
  if dp_state is a base case:
    return base case solution
  if dp_state in memo:
    return memo[dp_state]
  calculate solution for dp_state using recursive calls
  save the solution for dp_state in memo
  return solution for dp_state

function solve_original_problem(input):
  initialize memo
  return dp(starting_state, memo)
```

So far, we have discussed the recursive memoization strategy for solving dynamic programming problems. However, this is not the sole approach: you can avoid recursion by precomputing the entire memo at the beginning of the algorithm! For example, if you are asked to compute `fib(6)`, you could simply precompute all Fibonacci numbers up to the sixth Fibonacci number and then return its value. This method achieves the same linear time complexity as the previous approach, and it is implemented *iteratively* rather than recursively. The code is shown below:

```
uint64_t fib(int32_t n) {
  std::vector<uint64_t> memo(n + 1);
  memo[0] = 0;
  memo[1] = 1;
  for (size_t i = 2; i <= n; ++i) {
    memo[i] = memo[i - 1] + memo[i - 2];
  } // for i
  return memo[n];
} // fib()
```

This process of starting at the smallest possible subproblem and building upward toward a solution is known as **tabulation**. Both memoization and tabulation are valid approaches for solving a dynamic programming problem! The main difference between the two approaches is in how the subproblems are computed along the way. With memoization, you start with the inputs of the original problem and make your way down to the smallest subproblems using recursive calls. With tabulation, you start from the smallest subproblems and use their solutions to build upward toward the original problem you are trying to solve.[3] In both cases, the solutions of intermediate subproblems are stored in a memo so that they can be referenced later.

[3]Note: if you use tabulation, you must make sure you are solving the subproblems in the correct order, so that each subproblem's dependencies are all solved before the subproblem itself. This may require you to iterate over the memo in a specific way, which may not always be obvious!

Formally, dynamic programming that involves recursion and memoization falls into the category of **top-down dynamic programming**, while dynamic programming that involves iteration and tabulation falls into the category of **bottom-up dynamic programming**. The top-down approach starts with the original problem and divides the input into smaller subproblems that are solved recursively, while the bottom-up approach starts by solving the smallest subproblems and uses these solutions to solve for larger subproblems, eventually reaching the solution of the original problem. If a problem can be solved using top-down dynamic programming, it can also be solved using bottom-up dynamic programming (and vice versa), and choosing one over the other is often a matter of preference.

There are a few minor differences between the two approaches. Top-down dynamic programming only computes subproblems that are needed, since a result is only stored in the memo if a recursive call has already been made on its input. On the other hand, bottom-up dynamic programming precomputes *all* possible subproblems, so it may solve for subproblems that are never used. That being said, this does not mean that top-down is always better. For instance, a top-down approach is more likely to stack overflow for large inputs compared to a bottom-up approach, since it relies on recursion rather than iteration. A top-down approach may also prevent you from reusing positions of the memo for multiple subproblems to reduce memory usage (which we will discuss in section 23.8). Furthermore, one implementation may be easier or cleaner to write than the other depending on the type of problem you are trying to solve.

The performance of these two approaches also depends on the problem at hand. Bottom-up implementations are more cache-friendly and perform slightly better than top-down in many cases, since querying a precomputed table is often faster than making a recursive call and conditionally checking a memo for every subproblem encountered.[4] However, if the number of relevant subproblems only comprise a small portion of the overall output space, top-down may be faster than bottom-up if the cost of precomputing all possible subproblems exceeds the cost of making recursive calls only on the subproblems that matter.

※ 23.1.4 *Binomial Coefficients*

As another example, let's look at the *binomial coefficient* problem, which depends on two inputs in its recurrence relation rather than one. In mathematics, we define the binomial coefficient $\binom{n}{k}$ as the number of ways to choose k unordered outcomes out of n possibilities, defined as follows for non-negative integers n and k:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Example 23.2 Given two non-negative integers n and k, where $n \geq k$, implement a function `bi_coeff(n, k)` that returns the binomial coefficient $\binom{n}{k}$.

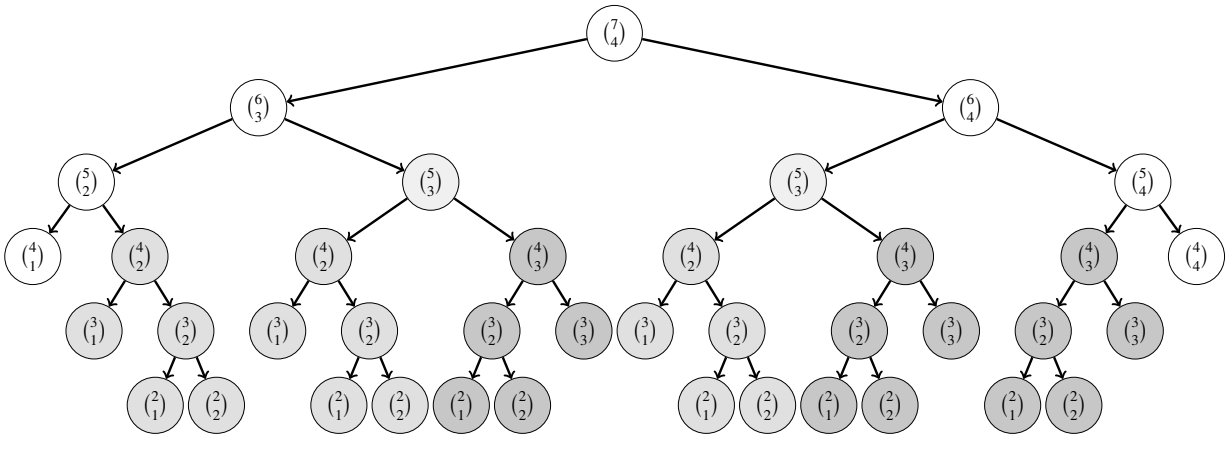
One way to approach this problem is to calculate $n!$, $k!$, and $(n-k)!$, and then use these values to solve for $\binom{n}{k}$. However, this doesn't always work, since factorials can get quite large: integer overflow becomes an issue when $n > 12$ for 32-bit integers, and $n > 20$ for 64-bit integers. Instead, a better way would be to use the *recursive* definition of the binomial coefficient formula, shown below (this should have been covered in EECS 203, but we won't expect you to derive anything like this on your own):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$
$$\binom{n}{0} = 1$$
$$\binom{n}{n} = 1$$

Converting this to a recursive function, we would get the following:

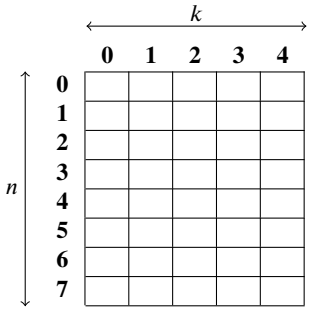
```
1  uint64_t bi_coeff(int32_t n, int32_t k) {
2    if (k == 0 || k == n) {
3      return 1;
4    } // if
5    return bi_coeff(n - 1, k - 1) + bi_coeff(n - 1, k);
6  } // bi_coeff()
```

However, this implementation is not fully efficient since there are duplicate subproblems. This is shown below for $\binom{7}{4}$:

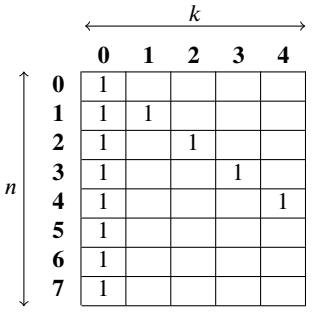


[4]Note: we are talking about actual runtime here, and *not* time complexity.

To avoid repeating recursive calls that have already been made, we can use dynamic programming and store the solutions of these subproblems in a memo. In this case, each subproblem is identified using two input values, n and k, so our memo will need to be two-dimensional with size $\Theta(nk)$ to support all possible subproblems. An example of a suitable memo is shown below: here, the solution to $\binom{n}{k}$ for any valid n and k is stored in row n, column k.



We know that $\binom{n}{0}$ and $\binom{n}{n}$ are both equal to 1 from the base case, so we can fill out their corresponding cells immediately (or return 1 in our recursive helper function if any of these base cases are reached):



If we use a top-down dynamic programming approach, we would take our original recursive function and add a memo to keep track of subproblems we have encountered before. A top-down solution is shown below:

```
1  uint64_t bi_coeff(int32_t n, int32_t k) {
2    std::vector<std::vector<uint64_t>> memo(n + 1, std::vector<uint64_t>(k + 1));
3    return bi_coeff_helper(n, k, memo);
4  } // bi_coeff()
5
6  uint64_t bi_coeff_helper(int32_t n, int32_t k, std::vector<std::vector<uint64_t>>& memo) {
7    if (k == 0 || n == k) {
8      return 1;
9    } // if
10   if (memo[n][k] != 0) { // recursive call made on (n, k) already
11     return memo[n][k];
12   } // if
13   // solve and store in memo
14   return memo[n][k] = bi_coeff_helper(n - 1, k - 1, memo) + bi_coeff_helper(n - 1, k, memo);
15 } // bi_coeff_helper()
```

If we use a bottom-up dynamic programming approach, we would loop over our memo and precompute all subproblems up to our original input. To implement this, we begin by looking at the smaller subproblems (e.g., the base cases), and then solving for the larger subproblems using the solutions we compute from the smaller subproblems. For example, we know that $\binom{1}{0}$ and $\binom{1}{1}$ are both equal to 1. Thus, we can solve for $\binom{2}{1} = \binom{1}{0} + \binom{1}{1} = 1 + 1 = 2$. Then, we can use the solution of $\binom{2}{1}$ to solve for $\binom{3}{1}$ and $\binom{3}{2}$, which we can use to solve for $\binom{4}{1}$, $\binom{4}{2}$, $\binom{4}{3}$, and so on, until we eventually obtain the solution for $\binom{n}{k}$. The code for a bottom-up solution is shown below.

```
1  uint64_t bi_coeff(int32_t n, int32_t k) {
2    std::vector<std::vector<uint64_t>>
3      memo(n + 1, std::vector<uint64_t>(k + 1));
4    for (int32_t curr_n = 0; curr_n <= n; ++curr_n) {
5      for (int32_t curr_k = 0; curr_k <= std::min(curr_n, k); ++curr_k) {
6        if (curr_n == curr_k || curr_k == 0) {
7          memo[curr_n][curr_k] = 1;
8        } // if
9        else {
10         memo[curr_n][curr_k] = memo[curr_n - 1][curr_k - 1] + memo[curr_n - 1][curr_k];
11       } // else
12     } // for curr_k
13   } // for curr_n
14   return memo[n][k];
15 } // bi_coeff()
```

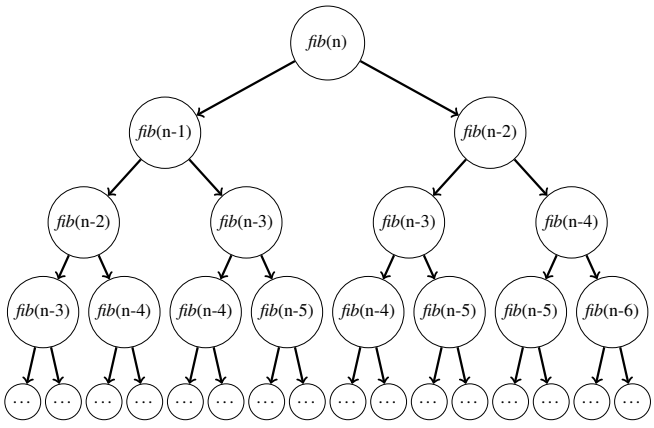
By storing the solutions of repeated subproblems in a memo, we only need to solve each subproblem once, regardless of how many times it is needed. Since there are only $\Theta(nk)$ subproblems we may encounter given values n and k, and each subproblem takes a constant amount of time to solve, the worst-case time complexity of a dynamic programming approach is $\Theta(nk)$. Furthermore, since we used a memo with dimensions $(n+1)\times(k+1)$, the auxiliary space used by our solution is also $\Theta(nk)$.

23.2 Dynamic Programming Implementation Strategies

In general, dynamic programming is useful for problems that exhibit the following characteristics:

1. **Optimal substructure**. A problem with an optimal substructure is one whose solution can be constructed using the optimal solutions of its smaller subproblems. If a problem has optimal substructure, the correct solution for some input size n can be computed by simply referencing the solutions of subproblems with input size less than n.
2. **Overlapping subproblems**. A problem exhibits the property of overlapping subproblems if multiple instances of the same subproblem are encountered while recursively decomposing the original problem down to its base cases.

For example, the Fibonacci problem exhibits both optimal substructure and overlapping subproblems. If we know the values of $fib(n-2)$ and $fib(n-1)$ for any n, we can immediately use their solutions to compute the value of $fib(n)$. In addition, if we recursively break any input down to its base case, we can see that multiple identical subproblems appear more than once.



The existence of overlapping subproblems is important to have! A common misconception is that dynamic programming can be used to improve the performance of any recusive algorithm. This is not true: for dynamic programming to be beneficial, there must exist overlapping subproblems that recur multiple times over the course of solving a problem. If the subproblems can be solved independently without any overlap, there is no need to store their solutions in a memo since you will never need them more than once (for these problems, an approach such as divide-and-conquer may be more applicable). Only when there exist *dependent subproblems* does a memo become useful in significantly improving the performance of an algorithm.[5]

Identifying and solving a dynamic programming problem can be challenging at times, but this is a topic that can be mastered with practice. At the end of the day, the goal of dynamic programming is the same regardless of what type of problem you are working with: you want to (1) break a larger problem into a collection of smaller subproblems, (2) solve each of these smaller subproblems *once* and store their solutions in a memo, and (3) use these solutions to compute the solution of the original problem, either in a top-down (recursive) or bottom-up (iterative) manner. The procedure for solving a dynamic programming problem can be roughly summarized using the following steps:

1. Verify that the problem you are solving can be broken up into smaller subproblems.

To determine if dynamic programming can be efficiently used to solve a problem, first ask yourself if the problem you are trying to solve can be *expressed using the solutions of smaller instances of the same problem*. That is, if you had the solutions of the problem on smaller input, can you use those solutions to help you calculate the solution on a larger input? If you can, then the problem has an optimal substructure, and dynamic programming may be viable approach.

2. Identify the problem variables that each subproblem depends on.

Next, determine the input variables that uniquely identify each subproblem; this will allow you to identify the number of subproblems you may need to solve, and therefore the size of your memo. For example, the subproblems of the Fibonacci problem depend on the value of n, while the subproblems of the binomial coefficient problem depend on the values of n and k. Therefore, there are potentially $\Theta(n)$ subproblems that you need to solve for the Fibonacci problem, and $\Theta(nk)$ subproblems that you need to solve for the binomial coefficient problem. You only want to focus on the variables that *differ* among the subproblems you may encounter — if a variable never changes across all subproblems, then it doesn't affect the number of subproblems you may need to solve.

[5]The term "dependent" here indicates that the process of solving one subproblem may depend on a solution that was already solved by another subproblem, and *not* that the two subproblems share resources or need to be solved together.

3. Express the problem in the form of a recurrence relation.

Use the information from the previous two steps to express the problem in the form of a recurrence relation, including all the base cases. *This is an important step that you should complete before you begin writing any code, since it will make implementation significantly easier!* Once you have identified how the subproblems relate to each other, the recurrence relation should come naturally: simply write an expression that describes how several smaller subproblems can be combined to compute the solution of the larger problem you are trying to solve.

For example, we know that $fib(0) = 0$, $fib(1) = 1$, and that $fib(n-2)$ and $fib(n-1)$ can be summed to obtain the value of $fib(n)$. Therefore, we would express the Fibonacci problem using the following recurrence relation:

$$fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n > 1 \end{cases}$$

For the binomial coefficient problem, we know that $C(n,0)$ and $C(n,n)$ are both equal to 1, and that $C(n,k)$ is the sum of $C(n-1,k-1)+C(n-1,k)$. Thus, we would express the binomial coefficient problem using the following recurrence relation:

$$C(n,k) = \begin{cases} 1, & \text{if } k = 0 \\ 1, & \text{if } n = k \\ C(n-1,k-1) + C(n-1,k), & \text{if } n > k \end{cases}$$

4. Identify the overlapping subproblems.

For dynamic programming to be useful, we want there to be overlapping subproblems. Therefore, it is always good to check your recurrence relation to see if duplicate subproblems actually exist (a good way to do this is to draw out a recurrence tree, like the one on the previous page, and look for subproblems that appear multiple times). If they don't exist, then dynamic programming may not be the best approach to use, and an alternative approach such as divide-and-conquer should be considered instead.

5. Convert the recurrence relation into code, using either a top-down (recursive) or bottom-up (iterative) approach.

Once you have identified the recurrence relation, convert it into code using either a top-down or bottom-up approach. If you decide to use a top-down approach, start by writing a straightforward recursive solution that solves for relevant subproblems using recursive calls. Then, add memoization to prevent the same subproblem from being computed twice — this is done by storing the solutions of recursive calls in a memo so that they can be easily retrieved later when they are needed again. If you decide to use a bottom-up approach, start with the base cases of the problem and use the recursive relationship of the problem to build upward toward the original input size you are trying to solve. Each intermediary subproblem you encounter is stored in a memo so that it can be quickly referenced later while solving for larger subproblems.

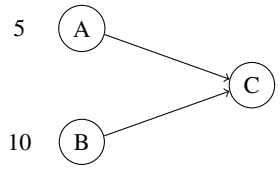
Regardless of how you approach a dynamic programming problem, the core component of any solution revolves around the memo, which stores the solutions of overlapping subproblems so that they don't have to be solved more than once. By using a bit of additional memory to store these subproblems, dynamic programming can greatly reduce the runtime of a recursive function! Over the next few sections, we will look at a few examples of problems that can be solved using dynamic programming.

23.3 Common Dynamic Programming Patterns

※ 23.3.1 *Counting Distinct Ways*

There are several common categories of problems that can be efficiently solved using dynamic programming. One common problem you might see involves *counting the number of distinct ways* to get to some target value or state, given a set of rules that you must follow to reach that target. These problem types often exhibit both the optimal substructure and overlapping subproblems that make dynamic programming useful.

To see why, consider the following example. Suppose you want to get to some state C, which is accessible from states A and B. There are 5 distinct ways to get to state A, and 10 distinct ways to get to state B. Knowing this, how many distinct ways are there to get to state C?

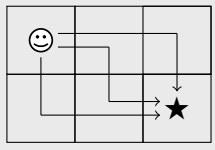


The answer is pretty straightforward: since there are 5 ways to get to state A and 10 ways to get to state B, the total number of ways to get to state C must be $5 + 10 = 15$, since C can only be reached from A or B. This simple example can actually be generalized to any state, allowing us to conceptualize a recurrence relation for any problem that fits this pattern. *To solve problems that ask you to count the distinct number of ways to reach some target state, you should sum up all the possible ways to reach states that directly precede it*. That is, if you wanted to count the number of ways to reach some target state C, and you can get to C from either A or B, then the number of ways to get to C is equal to the sum of the number of ways to get to A and B. We will look at a few examples of this pattern in this section.

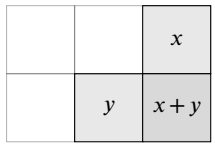
Example 23.3 You are located at the top-left corner of a $m \times n$ grid, and you want to reach the bottom-right corner. However, you are only allowed to move *down* or *right* at any point in time. Write a function that takes in m and n and returns the total number of *unique* paths you can take to reach the bottom-right corner of the grid.

Example: Given the following 2×3 grid, there are three unique paths that you can take to reach the bottom-right corner:

1. down, right, right
2. right, down, right
3. right, right, down



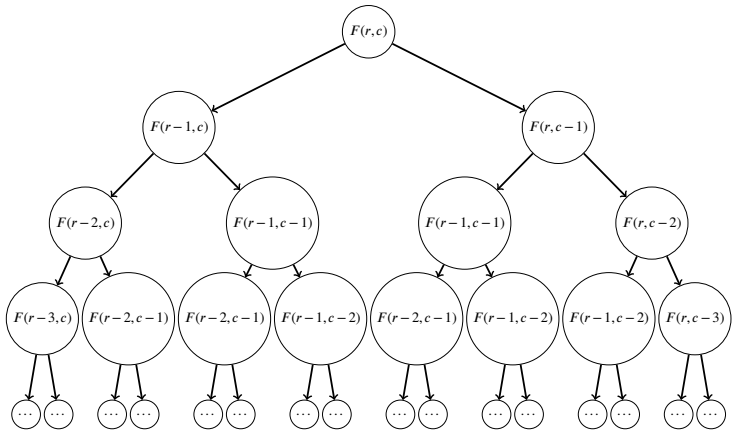
The key detail to notice here is that there are only two ways for you to reach the target square: either by moving down from the square directly above, or by moving right from the square directly to the left. Thus, if we know there are x ways to reach the square directly above, and y ways to reach the square directly to the left, then there must be $x + y$ ways to reach the target square.



Thus, given any square at (row r, column c) of the grid, we can calculate the number of ways to reach this square by recursively computing the number of ways to get to (row $r-1$, column c) and (row r, column $c-1$), and then summing up these two values. The base case is our starting position at (row 1, column 1), which we can trivially reach in exactly one way. There are also zero ways to reach any out-of-bounds square. This gives us the following recurrence relation $F(r,c)$, which represents the total number of ways to reach the square in (row r, column c):

$$F(r,c) = \begin{cases} 1, & \text{if } r = 1 \text{ and } c = 1 \\ 0, & \text{if } r < 1 \text{ or } c < 1 \text{ (out-of-bounds)} \\ F(r-1,c) + F(r,c-1), & \text{otherwise} \end{cases}$$

If we draw out the recursion tree, we would see that there are overlapping subproblems:



Thus, dynamic programming can be used to solve this problem. Each subproblem is identified using two variables, the row and column, so we will initialize a $m \times n$ memo of integers to store the solutions of subproblems we encounter (where the solution to $F(r,c)$ for any (r, c) is stored at position `memo[r][c]`). If we use a top-down approach to solve this problem, we would make recursive calls on $(r-1,c)$ and $(r,c-1)$ per the recurrence relation above, but we will check the memo before every recursive call to determine if the call has been made before. If the call has been made before, we just fetch its solution from the memo; otherwise, we make the recursive call and write the solution to the memo.

The code for a top-down solution is shown below:[6]

```
int32_t count_paths(int32_t m, int32_t n) {
  std::vector<std::vector<int32_t>> memo(m + 1, std::vector<int32_t>(n + 1));
  return count_paths_helper(m, n, memo);
} // count_paths()

int32_t count_paths_helper(int32_t r, int32_t c, std::vector<std::vector<int32_t>>& memo) {
  if (r == 1 && c == 1) { // base case
    return 1;
  } // if
  if (r < 1 || c < 1) { // out-of-bounds
    return 0;
  } // if
  if (memo[r][c] != 0) { // recursive call made before, fetch from memo
    return memo[r][c];
  } // if
  // solve and store in memo
  return memo[r][c] = count_paths_helper(r - 1, c, memo) + count_paths_helper(r, c - 1, memo);
} // count_paths_helper()
```

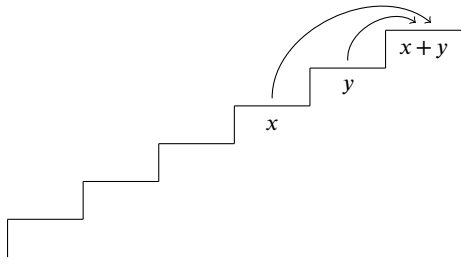
If we use a bottom-up approach, we would start with the base cases and build upward toward the solution using the relationships between subproblems. Here, we know that $F(1,1) = 1$, so we can use that to compute that $F(1,2)$ and $F(2,1)$, which can be used to compute $F(2,2)$, and so on, until we reach $F(m,n)$. The code for a bottom-up solution is shown below.

```
int32_t count_paths(int32_t m, int32_t n) {
  std::vector<std::vector<int32_t>> memo(m + 1, std::vector<int32_t>(n + 1, 1));
  for (int32_t r = 2; r <= m; ++r) {
    for (int32_t c = 2; c <= n; ++c) {
      memo[r][c] = memo[r - 1][c] + memo[r][c - 1];
    } // for c
  } // for r
  return memo[m][n];
} // count_paths()
```

Since there are a total of $\Theta(mn)$ subproblems that we may encounter, and each subproblem is solved only once in constant time with the help of the memo (i.e., add up two values), the time complexity of the above dynamic programming solution is $\Theta(mn)$. Similarly, since a memo of size $\Theta(mn)$ is used, the auxiliary space used by the implementation is also $\Theta(mn)$.

Example 23.4 You are climbing a staircase requiring n steps to reach the top. Write a function takes in the number of steps n and returns the number of distinct ways you climb the stairs, if you can only climb either 1 or 2 steps at any point in time.

Similar to the previous problem, the number of ways to reach the top of the staircase is dependent on the number of ways to reach the steps directly below it. In this case, we can reach the n^{th} step from either the step directly below it (by climbing one step) or two steps below it (by climbing two steps). Thus, if there are x ways to reach the step at position $n-2$ and y ways to reach the step at position $n-1$, then there must be $x+y$ ways to reach the step at position n. The base cases occur when $n = 0$ and $n = 1$, which both have a solution of 1 (there is only one way to reach the top of the stairs if the staircase only consists of one step, or if there are no stairs at all).

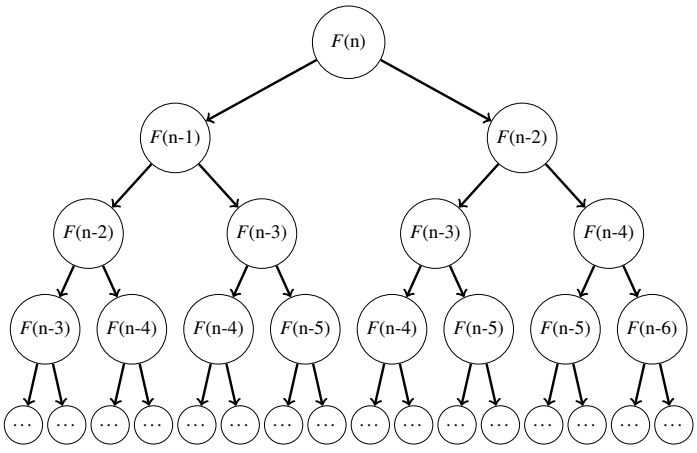


Since this problem can be defined in terms of solutions to smaller subproblems, we can express the problem using the following recurrence relation, where $F(i)$ represents the number of ways to reach the i^{th} step:

$$F(i) = \begin{cases} 1, & \text{if } i = 0 \text{ or } i = 1 \\ F(i-1) + F(i-2), & \text{if } i > 1 \end{cases}$$

[6] In this implementation, we are initializing a memo with dimensions $(m+1)\times(n+1)$ so that we can use 1-indexing, where the solution for $F(m,n)$ is stored in memo$[m][n]$ instead of memo$[m-1][n-1]$. However, this is not necessary, and it is perfectly fine to use 0-indexing and declare a memo with dimensions $m \times n$; you will just need to return memo$[m-1][n-1]$ at the end of the algorithm instead of memo$[m][n]$ and treat the base case as $(r = 0, c = 0)$ instead of $(r = 1, c = 1)$.

If we draw out the recursion tree for this recurrence relation, we would see that there are overlapping subproblems:



We can therefore implement a dynamic programming solution for this problem. Each subproblem is identified using a single variable (a "step position" that goes up to n), so we will initialize a one-dimensional memo of length n (where the solution to $F(i)$ for any position i is stored at position `memo[i]`). If we use a top-down approach to solve this problem, we would make the recursive calls as shown by the recurrence relation, but with a check before each call to see if its solution is already in the memo. The code for a top-down solution is shown below:

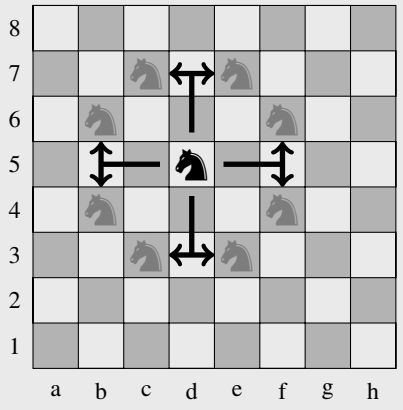
```
1   int32_t climb_stairs(int32_t n) {
2     std::vector<int32_t> memo(n + 1);
3     return climb_stairs_helper(n, memo);
4   } // climb_stairs()
5
6   int32_t climb_stairs_helper(int32_t i, std::vector<int32_t>& memo) {
7     if (i == 0 || i == 1) { // base cases
8       return 1;
9     } // if
10    if (memo[i] != 0) { // recursive call made before, fetch from memo
11      return memo[i];
12    } // if
13    // solve and store in memo
14    return memo[i] = climb_stairs_helper(i - 1, memo) + climb_stairs_helper(i - 2, memo);
15  } // climb_stairs_helper()
```

If we use a bottom-up approach, we start with the base cases and build upward to our desired solution. To calculate $F(n)$, we would use $F(1)$ and $F(2)$ to calculate $F(3)$, and then use the solution of $F(3)$ to calculate $F(4)$, and so on, until we reach $F(n)$. The code for a bottom-up solution is shown below:

```
1   int32_t climb_stairs(int32_t n) {
2     if (n == 0 || n == 1) {
3       return 1;
4     } // if
5     std::vector<int32_t> memo(n + 1);
6     memo[0] = 1;
7     memo[1] = 1;
8     for (int32_t i = 2; i <= n; ++i) {
9       memo[i] = memo[i - 1] + memo[i - 2];
10    } // for i
11    return memo[n];
12  } // climb_stairs()
```

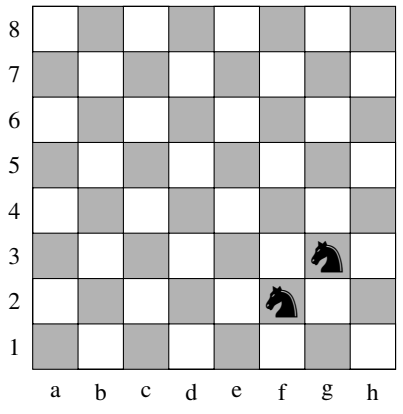
The worst-case time complexity and auxiliary space of this dynamic programming approach are both $\Theta(n)$, since there are n subproblems that are solved at most once, each taking $\Theta(1)$ time to compute with the help of a memo of size $\Theta(n)$.

Example 23.5 In the game of chess, the knight travels in an "L" shape, moving two spaces in any direction, then turning 90 degrees left or right and moving forward another space. For example, the following chessboard depicts the squares that the knight in the middle can access in one move.



Write a function that, when given positive integers m and n, returns the total number of distinct ways a knight at the top-left corner of a $n \times n$ chessboard can move to the bottom-right corner in exactly m moves.

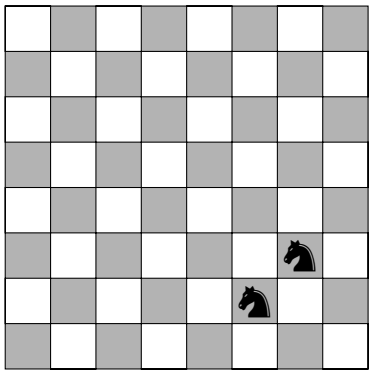
Notice that this problem is very similar to the grid problem we covered earlier. There are two positions that allow a knight to directly reach the bottom-right corner of the grid, as shown:



Therefore, if we solve for the number of ways to get to these two positions, we can use it to solve for the number of ways to get to the bottom-right corner. However, there is an additional variable that we have to worry about in this problem: the number of moves! Simply knowing the number of ways to get to a square is no longer enough, since you have to make sure the knight lands in the bottom-right corner after the correct number of moves. To address this, we must consider this additional dimension in our analysis of the recurrence relation.

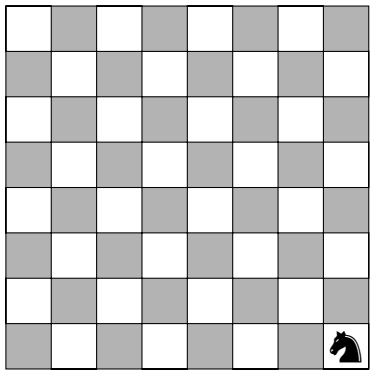
Using the dynamic programming pattern covered in this section, we would conclude that the number of ways to reach the bottom-right corner is equal to the total number of ways to reach squares that allow a knight direct access to the bottom-right corner. However, since we are considering move count as well, we will add this additional dimension to the subproblems we want to solve. Here, the number of ways to reach the bottom-right corner in exactly m moves is equal to the number of ways to reach the two preceding squares in $m-1$ moves:

After *m* - 1 Moves



The total number of ways to get to these positions in $m-1$ moves...

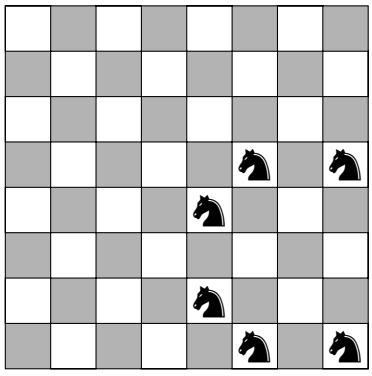
After *m* Moves



...is equal to the number of ways to get to the bottom-right corner in m moves!

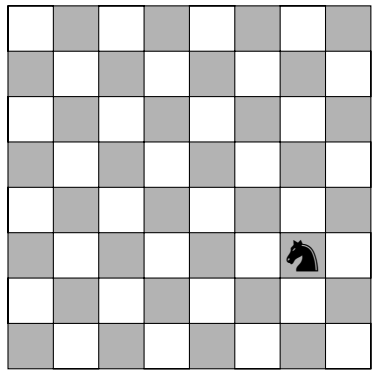
Similarly, the number of ways to get to these positions in $m-1$ moves can be expressed in terms of the number of ways to get to preceding positions in $m-2$ moves.

After *m* - 2 Moves



The total number of ways to get to any of these positions in $m-2$ moves...

After *m* - 1 Moves

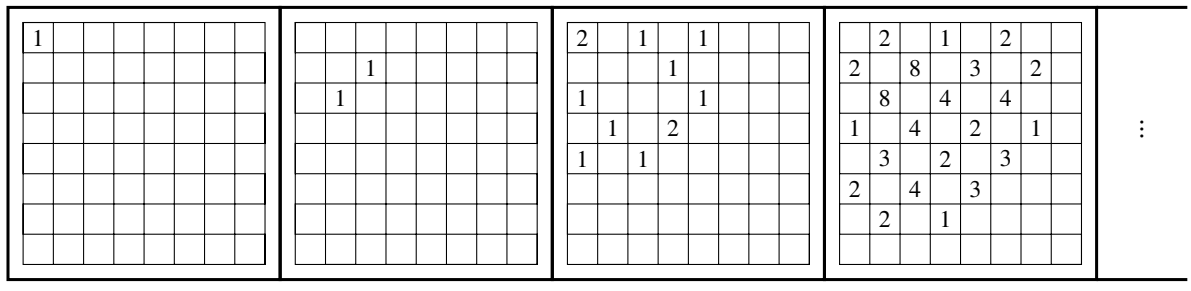


...is equal to the number of ways to get to this square in $m-1$ moves!

We can therefore express this problem using the following recurrence relation, where $F(i,r,c)$ represents the total number of ways for a knight to reach the square at (row r, column c) in exactly i moves:

$$F(i,r,c) = \begin{cases} 1, & \text{if } i=0, r=0, c=0 \\ 0, & \text{if } i=0, r\neq 0, c\neq 0 \\ 0, & \text{if } r<0 \text{ or } c<0 \text{ or } r\geq n \text{ or } c\geq n \text{ (out-of-bounds)} \\ F(i-1,r-2,c-1)+F(i-1,r-2,c+1) & \\ +F(i-1,r+2,c-1)+F(i-1,r+2,c+1) & \\ +F(i-1,r-1,c-2)+F(i-1,r-1,c+2) & \\ +F(i-1,r+1,c-2)+F(i-1,r+1,c+2), & \text{if } i>0, \text{ and } r \text{ and } c \text{ not out-of-bounds} \end{cases}$$

This recurrence relation exhibits the property of overlapping subproblems, so we can use dynamic programming to implement a solution. Each subproblem is identified using three changing variables — the move number, row, and column — so we will initialize a memo with dimensions $m \times n \times n$ (since row and column go up to n and the number of moves goes up to m), where the solution to $F(i,r,c)$ is stored in `memo[i][r][c]`. You can think of this as a memo of chessboards, where `memo[i]` stores the number of ways to reach each chessboard position in exactly `i` moves.[7]



[7]The blank cells in the illustration represent unreachable squares and have a value of 0 (since there are 0 ways to reach them).

A top-down solution for this problem is shown below:

```
int32_t knight_moves(int32_t m, int32_t n) {
  std::vector<std::vector<std::vector<int32_t>>> memo(m + 1,
    std::vector<std::vector<int32_t>>(n, std::vector<int32_t>(n, -1)));
  return knight_moves_helper(m, n - 1, n - 1, memo);
} // knight_moves()

int32_t knight_moves_helper(int32_t moves_left, int32_t row, int32_t col,
  std::vector<std::vector<std::vector<int32_t>>>& memo) {
  if (moves_left == 0 && row == 0 && col == 0) {
    return 1;
  } // if
  if (row < 0 || col < 0 || row >= memo[0].size() || col >= memo[0].size() || moves_left == 0) {
    return 0;
  } // if
  if (memo[moves_left][row][col] != -1) {  // recursive call made before
    return memo[moves_left][row][col];
  } // if
  // solve and store in memo
  return memo[moves_left][row][col] =
    knight_moves_helper(moves_left - 1, row - 2, col - 1, memo) +
    knight_moves_helper(moves_left - 1, row - 2, col + 1, memo) +
    knight_moves_helper(moves_left - 1, row + 2, col - 1, memo) +
    knight_moves_helper(moves_left - 1, row + 2, col + 1, memo) +
    knight_moves_helper(moves_left - 1, row - 1, col - 2, memo) +
    knight_moves_helper(moves_left - 1, row - 1, col + 2, memo) +
    knight_moves_helper(moves_left - 1, row + 1, col - 2, memo) +
    knight_moves_helper(moves_left - 1, row + 1, col + 2, memo);
} // knight_moves_helper()
```

A bottom-up solution is shown below:

```
int32_t knight_moves(int32_t m, int32_t n) {
  std::vector<std::vector<std::vector<int32_t>>> memo(m + 1,
    std::vector<std::vector<int32_t>>(n, std::vector<int32_t>(n)));
  memo[0][0][0] = 1;
  for (int32_t num_move = 1; num_move <= m; ++num_move) {
    for (int32_t row = 0; row < n; ++row) {
      for (int32_t col = 0; col < n; ++col) {
        memo[num_move][row][col] =
          memo_val_bounds_check(num_move - 1, row - 2, col - 1, memo) +
          memo_val_bounds_check(num_move - 1, row - 2, col + 1, memo) +
          memo_val_bounds_check(num_move - 1, row + 2, col - 1, memo) +
          memo_val_bounds_check(num_move - 1, row + 2, col + 1, memo) +
          memo_val_bounds_check(num_move - 1, row - 1, col - 2, memo) +
          memo_val_bounds_check(num_move - 1, row - 1, col + 2, memo) +
          memo_val_bounds_check(num_move - 1, row + 1, col - 2, memo) +
          memo_val_bounds_check(num_move - 1, row + 1, col + 2, memo);
      } // for col
    } // for row
  } // for num_move
  return memo[m][n - 1][n - 1];
} // knight_moves()

int32_t memo_val_bounds_check(int32_t move, int32_t row, int32_t col,
                              std::vector<std::vector<std::vector<int32_t>>>& memo) {
  if (row < 0 || col < 0 || row >= memo[0].size() || col >= memo[0].size()) {
    return 0;
  } // if
  return memo[move][row][col];
} // memo_val_bounds()
```

The worst-case time and space complexity of this implementation is $\Theta(mn^2)$, since each of the $\Theta(mn^2)$ subproblems can be solved in $\Theta(1)$ time, and each subproblem is only solved once with the help of the memo.

Example 23.6 You are given n dice, each with m faces numbered 1, 2, …, m. Given n, m, and a target value x, write a function that returns the total number of distinct ways to roll the dice so that the face-up values sum to x. For example, given $n = 2$, $m = 6$, and $x = 7$, you would return 6, since there are six distinct ways to roll a 7: (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), and (6, 1).

Since we are asked to count the number of distinct ways to attain a goal, we will start by identifying a recurrence relation that sums up the number of ways to reach all preceding states. To help with this process, consider the following example: suppose you are given three six-sided dice, with a target value of 15 ($n = 3$, $m = 6$, and $x = 15$). Here, there are six different preceding states that allow us to reach exactly 15:

- If the sum of the first two dice is 9.
- If the sum of the first two dice is 10.
- If the sum of the first two dice is 11.
- If the sum of the first two dice is 12.
- If the sum of the first two dice is 13.
- If the sum of the first two dice is 14.

If the sum of the first two dice is anything but a value between 9 and 14, then it would be impossible to roll the third die in a way such that the total sum of all three dice is exactly 15. Therefore, the total number of ways to roll a sum of 15 using three dice is equal to the total number of ways to roll a sum between 9 and 14 using two dice!

We can generalize this to any n, m, and x. The base case happens when $n = 1$, which trivially implies there is only one way to sum to any target between 1 and m (since there is only one die). For any other n, the number of ways to roll a sum of x using n m-sided dice is equal to the number of ways to roll a value between $\min(x-m, 1)$ and $x-1$ using $n-1$ dice. This can be converted into the following recurrence relation, where $F(n,x)$ represents the number of ways to roll a sum of x using n m-sided dice (note that m is not a changing variable in this case, so each subproblem can be uniquely identified using just n and x):

$$F(n,x) = \begin{cases} 1, & \text{if } n = 1, 1 \leq x \leq m \\ 0, & \text{if } n = 1, \text{ otherwise} \\ F(n-1,x-1) + F(n-1,x-2) + \ldots + \min(F(n-1,x-m), F(n-1,1)), & \text{if } n > 1 \end{cases}$$

If you were to draw out the recurrence tree, you would see that there are overlapping subproblems (e.g., $F(n-1,x-1)$ and $F(n-1,x-2)$ both rely on the value of $F(n-2,x-3)$). Therefore, dynamic programming can be used to solve this problem. To store our subproblems, we will use a two-dimensional memo with size based on n and x, where `memo[i][j]` stores the number of ways to roll a sum of j using i dice. A top-down solution is shown below:

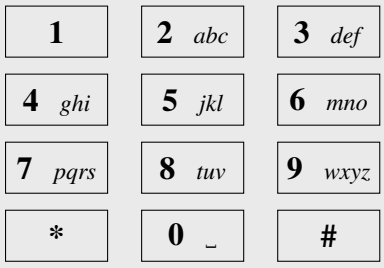
```
int32_t dice_roll(int32_t n, int32_t m, int32_t x) {
  std::vector<std::vector<int32_t>> memo(n + 1, std::vector<int>(x + 1, -1));
  memo[1] = std::vector<int32_t>(x + 1, 0);
  for (int32_t j = 1; j <= m && j <= x; ++j) {
    memo[1][j] = 1;  // only one way to reach [1, m] if only one dice
  } // for j
  return dice_roll_helper(n, m, x, memo);
} // dice_roll()

int32_t dice_roll_helper(int32_t num_dice, int32_t faces, int32_t target,
                         std::vector<std::vector<int32_t>>& memo) {
  if (memo[num_dice][target] != -1) {  // recursive call made before
    return memo[num_dice][target];
  } // if
  // solve and store in memo
  int32_t num_ways = 0;
  for (int32_t diff = 1; diff <= faces && diff < target; ++diff) {
    num_ways += dice_roll_helper(num_dice - 1, faces, target - diff, memo);
  } // for diff
  return memo[num_dice][target] = num_ways;
} // dice_roll_helper()
```

A bottom-up solution is shown below:

```
int32_t dice_roll(int32_t n, int32_t m, int32_t x) {
  std::vector<std::vector<int32_t>> memo(n + 1, std::vector<int32_t>(x + 1));
  for (int32_t j = 1; j <= m && j <= x; ++j) {
    memo[1][j] = 1;  // only one way to reach [1, m] if only one dice
  } // for j
  for (int32_t i = 2; i <= n; ++i) {
    for (int32_t j = i; j <= x; ++j) {
      for (int32_t k = 1; k <= m && k < j; ++k) {
        memo[i][j] += memo[i - 1][j - k];
      } // for k
    } // for j
  } // for i
  return memo[n][x];
} // dice_roll()
```

The auxiliary space used by this implementation is $\Theta(nx)$ for the size of the memo. The time complexity is $\Theta(mnx)$, since each of the nx subproblems takes $\Theta(m)$ time to compute (as you have to loop over the number of faces to identify all preceding states).

Example 23.7 Your friend is texting you a message — however, they only have a flip phone and therefore have to rely on the following mapping of digits to send messages:



To add a letter, your friend must press the key of the corresponding digit n times, where n is the position of the letter in the key. For example, to add the letter 's' to the message, the number 7 has to be pressed four times in succession. For this problem, you may assume that the 0, 1, *, and # keys are never used.

However, due to a data error, you ended up receiving a string of pressed keys instead of the message itself. For instance, if your friend sent the message "eecs", you would get the message "33332227777". Implement a function that, when given a string representing the sequence of keys pressed, returns the total number of possible messages that could have been sent.

Example: Given the string `"22228"`, you would return 7, since there are seven possible messages that could have been sent with the sequence of keys: "aaaat", "aabt", "act", "baaat", "babt", "bbat", and "cat".

This is another problem where we are asked to count ways, so we will consider a dynamic programming approach. Like before, our recurrence will rely on the idea of summing up the number of ways to reach the preceding states of each potential sequence. However, what are these preceding states? To conceptualize this, consider the sequence of presses "2222". What are the possible ways that could allow us to end up with this sequence? Notice there are three possibilities that allow us to reach this combination:

- Our previous message consisted of the sequence "222", and "2" was pressed to add the letter "a".
- Our previous message consisted of the sequence "22", and "22" was pressed to add the letter "b".
- Our previous message consisted of the sequence "2", and "222" was pressed to add the letter "c".

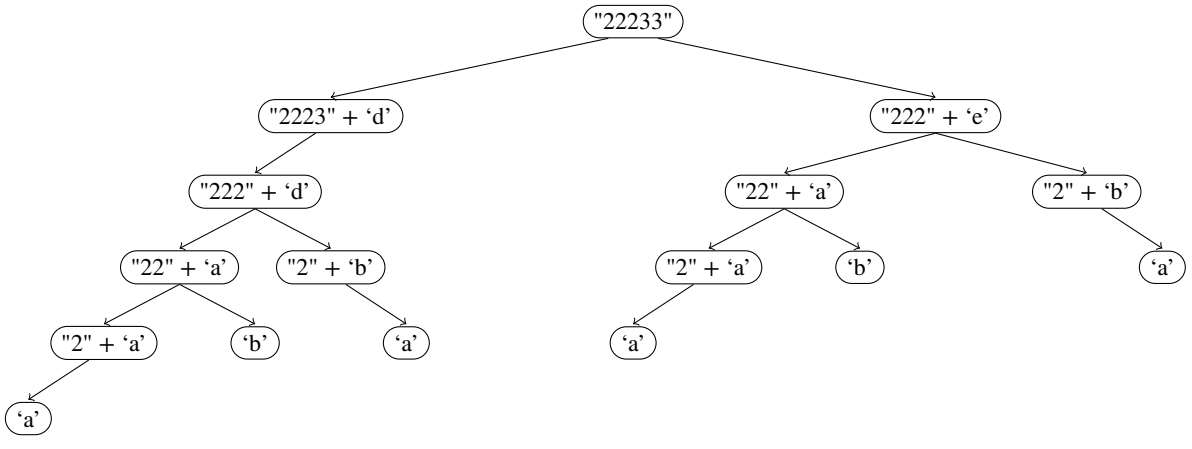
Therefore, the total number of possible messages from the sequence "2222" is the sum of the number of possible messages from the sequences "222", "22", and "2", which are the states that could have preceded "2222".

Note that you only have to consider multiple states if the last numbers in the sequence are the same. For a sequence like "2223", the only preceding state is "222", since the final "3" is different from the previous digit and cannot be merged with the digits before it to form a new letter (i.e., you cannot get to "2223" directly from "22" or "2"). Therefore, the number of ways to obtain the combination "2223" is the same as the number of ways to obtain the combination "222", since "222" is the only state that allows you to directly reach "2223".

We can use this information to devise a recurrence relation. For a sequence of presses up to the n^{th} digit (0-indexing), the total number of possible messages $F(n)$ can be represented as:

$$F(n) = \begin{cases} 1, & \text{if } n \leq 0 \\ F(n-1) & \\ +F(n-2) & \text{if the last two digits match} \\ +F(n-3) & \text{if the last three digits match} \\ +F(n-4) & \text{if the last four digits match (for '7' and '9')} \end{cases}$$

This recurrence involves repeated subproblems (as shown by the recurrence tree below), so dynamic programming would be a viable approach.



A top-down solution is shown below:

```
int32_t get_number_of_messages(const std::string& sequence) {
  std::vector<int32_t> memo(sequence.size(), -1);
  memo[0] = 1;
  return get_number_of_messages_helper(sequence, sequence.size() - 1, memo);
} // get_number_of_messages()

int32_t get_number_of_messages_helper(const std::string& sequence, int32_t position,
                                      std::vector<int32_t>& memo) {
  // position == 0 indicates the number of possible messages in the subsequence up to index 0
  // if position goes below 0, you should return the number of possible messages given an
  // empty sequence, which is 1 since there is only 1 possible message from an empty string
  if (position == -1) {
    return 1;
  } // if
  if (memo[position] != -1) {  // recursive call made before
    return memo[position];
  } // if
  // solve and store in memo
  int32_t max_key_press = (sequence[position] == '7' || sequence[position] == '9') ? 4 : 3;
  int32_t press_num = 0, curr_pos = position, total_messages = 0;
  // iterate over the previous characters (up to the maximum possible repeats) and
  // add subproblem if the characters match
  while (press_num++ < max_key_press && curr_pos >= 0 && sequence[curr_pos] == sequence[position]) {
    --curr_pos;
    total_messages += get_number_of_messages_helper(sequence, curr_pos, memo);
  } // while
  return memo[position] = total_messages;
} // get_number_of_messages_helper()
```

A bottom-up solution is shown below:

```
int32_t get_number_of_messages(const std::string& sequence) {
  std::vector<int32_t> memo(sequence.size() + 1, -1);
  for (size_t i = 1; i <= sequence.size(); ++i) {
    memo[i] = memo[i - 1];
    if (i >= 2 && sequence[i - 1] == sequence[i - 2]) {
      memo[i] += memo[i - 2];
      if (i >= 3 && sequence[i - 1] == sequence[i - 3]) {
        memo[i] += memo[i - 3];
        if ((sequence[i - 1] == '7' || sequence[i - 1] == '9') &&
            i >= 4 && sequence[i - 1] == sequence[i - 4]) {
          memo[i] += memo[i - 4];
        } // if
      } // if
    } // if
  } // for i
  return memo.back();
} // get_number_of_messages()
```

The time complexity of this solution is $\Theta(n)$ given a sequence of length n, since each of the n subproblems takes $\Theta(1)$ time to compute. The auxiliary space used by this implementation is $\Theta(n)$ for the size of the memo (however, as we will see in section 23.8, it is possible to reduce the auxiliary space of the bottom-up solution to $\Theta(1)$ if we only keep track of the subproblems we may actually need in the future).

※ 23.3.2 *Path and Decision Optimization*

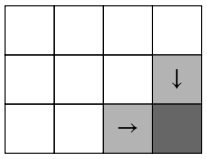
Another common pattern of dynamic programming problems involves finding the best path or sequence of decisions to reach a certain goal, given a cost or value associated with each state. To identify the best path to reach any given state for these types of problems, we can use a strategy similar to the one we used to count distinct ways: compute the best possible path to reach any state *directly preceding the current state*, and add it to the cost or value of the current state.

Example 23.8 You are given a $m \times n$ grid filled with non-negative integers. Write a function that finds a path from the top-left corner of the grid to the bottom-right with *the minimum sum* and returns this sum, assuming that you can only move down or right at any point in time. For example, given the following 3×4 grid, you would return 14, since this is the cost of the minimum path that starts from the top-left and ends at the bottom-right with only down and right movements:

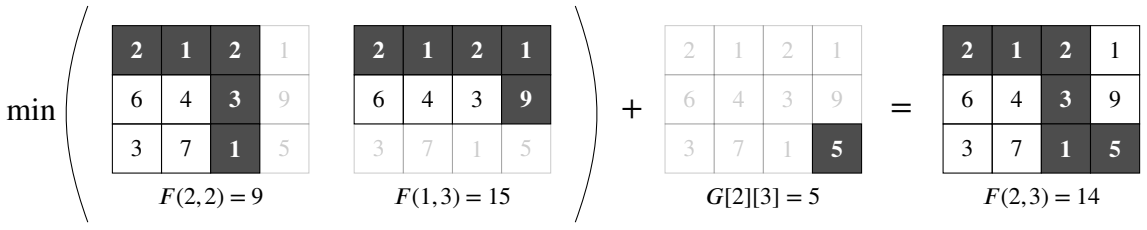
2	1	2	1
6	4	3	9
3	7	1	5

2	**1**	**2**	1
6	4	**3**	9
3	7	**1**	**5**

This problem is very similar to the one in which we had to count the number of ways to get to the bottom-right corner of a grid. Notice here that there are two ways to reach the bottom-right square: either from the cell above or the cell to the left.



Therefore, if we knew the best path to either of these two squares, we would be able to compute the overall best path by simply adding it to the value of the final square! This gives us our recursive relationship between subproblems, where $F(r,c)$ represents the minimum cost path to reach the cell at (row r, column c):

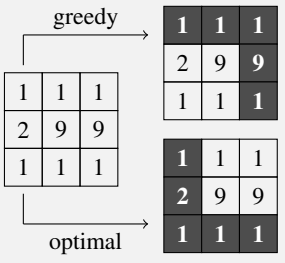


The recurrence relation for this problem is as follows. Given a grid of values G, we can compute the minimum path to reach (row r, column c) by solving the subproblems $F(r-1,c)$ and $F(r,c-1)$, and then combining the smaller of these two values with the value of $G[r][c]$. The base case occurs when there is only a single square in the grid ($r = c = 0$), which trivially implies that the minimum path cost is simply the value of that square, $G[0][0]$. Since this recurrence relation exhibits overlapping subproblems, a dynamic programming approach can be used.

$$F(r,c) = \begin{cases} G[0][0], & \text{if } r = 0, c = 0 \\ \infty \text{ (i.e., should be ignored)}, & \text{if } r < 0 \text{ or } c < 0 \text{ (out-of-bounds)} \\ \min(F(r-1,c), F(r,c-1)) + G[r][c], & \text{if } r > 0, c > 0 \end{cases}$$

Remark: Since this is an optimization problem with an optimal substructure, it is good practice to check if a greedy solution exists before we start planning a dynamic programming solution. If we can find an algorithm that guarantees an optimal solution by simply making a locally optimal choice at each step, we would not need to potentially solve and store every possible subproblem as required by a dynamic programming approach.

As covered in chapter 21, a greedy approach works if the problem exhibits two properties: an *optimal substructure* (where the optimal solution of a problem can be computed using the optimal solutions of its subproblems) and the *greedy-choice property* (at least one optimal solution contains the first greedy choice). Although the problem exhibits an optimal substructure, it does *not* exhibit the greedy-choice property. In the following grid, the first greedy choice would be to move to the right, since the cost of going right (1) is better than the cost of going down (2). However, the optimal solution goes down first, with a total path cost of $1 + 2 + 1 + 1 + 1 = 6$. Since the greedy-choice property does not hold, we can conclude that the problem cannot be solved using a greedy approach.



A top-down solution for this problem is shown below:

```
1   int32_t min_path(const std::vector<std::vector<int32_t>>& grid) {
2     size_t m = grid.size(), n = grid[0].size();
3     std::vector<std::vector<int32_t>> memo(m, std::vector<int32_t>(n, -1));
4     return min_path_helper(m - 1, n - 1, grid, memo);
5   } // min_path()
6
7   int32_t min_path_helper(int32_t row, int32_t col, const std::vector<std::vector<int32_t>>& grid,
8                           std::vector<std::vector<int32_t>>& memo) {
9     if (row == 0 && c == 0) {
10      return grid[0][0];
11    } // if
12    if (row < 0 || col < 0) {
13      // prevents algorithm from choosing a path out-of-bounds
14      return std::numeric_limits<int>::max();
15    } // if
16    if (memo[row][col] != -1) {  // recursive call made before
17      return memo[row][col];
18    } // if
19    else {  // solve and store in memo
20      return memo[row][col] = std::min(min_path_helper(row - 1, col, grid, memo),
21                                       min_path_helper(row, col - 1, grid, memo)) + grid[row][col];
22    } // else
23  } // min_path_helper()
```

A bottom-up solution for this problem is shown below:

```
int32_t min_path(const std::vector<std::vector<int32_t>>& grid) {
  size_t m = grid.size(), n = grid[0].size();
  std::vector<std::vector<int32_t>> memo(m, std::vector<int32_t>(n));
  memo[0][0] = grid[0][0];
  for (size_t i = 1; i < m; ++i) {
    memo[i][0] = memo[i - 1][0] + grid[i][0];
  } // for i
  for (size_t j = 1; j < n; ++j) {
    memo[0][j] = memo[0][j - 1] + grid[0][j];
  } // for j
  for (size_t i = 1; i < m; ++i) {
    for (size_t j = 1; j < n; ++j) {
      memo[i][j] = std::min(memo[i - 1][j], memo[i][j - 1]) + grid[i][j];
    } // for j
  } // for i
  return memo[m - 1][n - 1];
} // min_path()
```

The time and space complexity of this problem is $\Theta(mn)$, since up to $\Theta(mn)$ subproblems need to be solved if a memo is used to store overlapping solutions, and each subproblem takes $\Theta(1)$ time to compute.

Example 23.9 You are given an integer array of coin denominations and a target value. Write a function that returns the fewest number of coins that you need to make change for the given target, or `0` if no such way to make change exists.

This is a problem we have seen before when discussing greedy algorithms. Although the greedy approach works for some coin denominations, it does not find an optimal solution in all cases (see example 21.3). On the contrary, a dynamic programming approach *does* guarantee an optimal solution, regardless of which coin denominations are used! This is because dynamic programming considers all relevant subproblems when building up a solution, and not just the locally optimal ones.

Since this is a decision optimization problem, where we have to optimize the coin we select at each step, we can begin a dynamic programming solution by considering the optimal solutions of all preceding states that allow us to reach x cents with the addition of an available coin. Given any set of n coin denomiations $d_1, d_2, \ldots, d_n$ and a target amount x, there are up to n preceding states that allow us to directly reach our goal:

- If we have already made optimal change for $x - d_1$ cents, we can simply add a d_1 cent coin.
- If we have already made optimal change for $x - d_2$ cents, we can simply add a d_2 cent coin.
- If we have already made optimal change for $x - d_n$ cents, we can simply add a d_n cent coin.

Therefore, to determine the minimum number of coins needed to make change for x cents, we can first compute the minimum number of coins to make change for $x - d_1$ cents, $x - d_2$ cents, …, and $x - d_n$ cents, find the solution out of these subproblems that requires the fewest number of coins, and then add one to that solution (for the additional coin we need to reach x cents). The recurrence relation is shown below, where $F(x)$ represents the minimum number of coins needed to make change for x cents.

$$F(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1 + \min(F(x - d_1), F(x - d_2), \ldots, F(\min(x - d_n, 0)), & \text{if } x > 0 \end{cases}$$

This recurrence exhibits overlapping subproblems, so dynamic programming can be used. A top-down solution is shown below:

```
int32_t coin_change(const std::vector<int32_t>& coins, int32_t target) {
  std::vector<int32_t> memo(target + 1, -1);
  return coin_change_helper(coins, target, memo);
} // coin_change()

int32_t coin_change_helper(std::vector<int32_t>& coins, int32_t rem_change,
                           std::vector<int32_t>& memo) {
  if (rem_change == 0) {
    return 0;
  } // if
  if (rem_change < 0) {
    return -1;  // invalid
  } // if
  // if recursive call made before, fetch solution from memo
  if (memo[rem_change] != -1) { // recursive call made before
    return memo[rem_change];
  } // if
  else { // solve and store in memo
    int32_t curr_min = std::numeric_limits<int32_t>::max();
    for (int32_t coin_value : coins) {
      int32_t result = coin_change_helper(coins, rem_change - coin_value, memo);
      if (result >= 0 && result < curr_min) {
        curr_min = result + 1;
      } // if
    } // for coin_value
    return memo[rem_change] =
      (curr_min == std::numeric_limits<int32_t>::max()) ? -1 : curr_min;
  } // else
} // coin_change_helper()
```

A bottom-up solution is shown below:

```
int32_t coin_change(const std::vector<int32_t>& coins, int32_t target) {
  std::vector<int32_t> memo(target + 1);
  for (int32_t i = 1; i <= target; ++i) {
    memo[i] = std::numeric_limits<int32_t>::max();
    for (int32_t coin_value : coins) {
      if (coin_value <= i &&
          memo[i - coin_value] != std::numeric_limits<int32_t>::max()) {
        memo[i] = std::min(memo[i], memo[i - coin_value] + 1);
      } // if
    } // for coin_value
  } // for i
  return memo[target] == std::numeric_limits<int32_t>::max() ? -1 : memo[target];
} // coin_change()
```

Given a target of x cents and n coins, the time complexity of this solution is $\Theta(nx)$, and the auxiliary space is $\Theta(x)$.

Example 23.10 You currently have a job that requires you to travel around to different cities. Your travel days are planned in advance, and they are stored in a sorted integer array `days`. Luckily, you are able to buy train passes that allow you travel freely within a certain range of time after the date of purchase. There are three types of passes you can buy:

- a **1-day** pass
- a **7-day** pass
- a **30-day** pass

For instance, if you buy a 7-day pass on day 25, you can travel for 7 days starting on day 25: days 25, 26, 27, 28, 29, 30, and 31. The price of each pass is stored in an array `costs`, where `costs[0]` stores the price of a 1-day pass, `costs[1]` stores the price of a 7-day pass, and `costs[2]` stores the price of a 30-day pass. Given `days` and `costs`, implement a function that returns the minimum cost required to travel on all the specified days in the `days` array.

Example: Given `days = [1, 5, 11, 13, 17, 20, 37]` and `costs = [3, 7, 15]`, you would return `18`, since this is the minimum cost required to travel on all the given days (buy a 30-day pass on day 1 for $15, and a 1-day pass on day 37 for $3).

We want to have a valid pass on all of our specified travel days up to some final travel day x. Notice that there exist three options that allow us to travel on day x while also potentially minimizing total cost:

- We buy a 1-day pass on day x
- We buy a 7-day pass on day $x-6$
- We buy a 30-day pass on day $x-29$

If we buy a 1-day pass on day x, the minimum cost across all of our travel days would be the price of the 1-day pass on day x plus the minimum cost of satisfying all of our travel days up to day $x-1$. If we let $F(x)$ denote the minimum cost of traveling up to day x, then the minimum cost of traveling up to day x, **assuming we buy a 1-day pass on day x**, is equal to:

$$F(x-1) \ + \ (\text{cost of 1-day pass})$$

If we buy a 7-day pass on day $x-6$, the minimum cost across all of our travel days would be the price of the 7-day pass on day $x-6$ plus the minimum cost of satisfying all of our travel days up to day $x-7$. In other words, the minimum cost of traveling up to day x, **assuming we buy a 7-day pass on day $x-6$**, is equal to:

$$F(x-7) \ + \ (\text{cost of 7-day pass})$$

Similarly, if we buy a 30-day pass on day $x-29$, the minimum cost across all of our travel days would be the price of the 30-day pass on day $x-29$ plus the minimum cost of satisfying all of our travel days up to day $x-30$. The minimum cost of traveling up to day x, **assuming we buy a 30-day pass on day $x-29$**, is equal to:

$$F(x-30) \ + \ (\text{cost of 30-day pass})$$

The actual minimum cost to travel on day x would therefore be the best of these three options:

$$F(x) = \min(F(x-1)+\text{cost}[0],\ F(x-7)+\text{cost}[1],\ F(x-30)+\text{cost}[2])$$

However, there is an additional detail we must consider when solving this problem: if the current day x we are considering is not a travel day, then we do not have to consider buying a pass for that day. Thus, we can forward propogate the best cost from the previous travel day $x-1$ and use that value as the best cost up to day x. The recurrence relation for this problem is shown below. The base case occurs when $x < 1$ (outside the travel window), which has a cost of zero since you do not have to buy any tickets before the earliest possible travel date. There exist overlapping subproblems, so dynamic programming can be used.

$$F(x) = \begin{cases} 0, & \text{if } x < 1 \\ F(x-1), & \text{if } x \text{ not in travel days} \\ \min(F(x-1)+\text{cost}[0], F(x-7)+\text{cost}[1], F(x-30)+\text{cost}[2]), & \text{if } x \leq \text{final travel day, otherwise} \end{cases}$$

A top-down solution is shown below. Here, we use an unordered set to identify which days are travel days in constant time.[8]

```
1  int32_t min_cost_travel(const std::vector<int32_t>& days, const std::vector<int32_t>& costs) {
2    std::vector<int32_t> memo(days.back() + 1, -1);
3    std::unordered_set<int32_t> days_set(days.begin(), days.end());
4    return min_cost_helper(days_set, costs, days.back(), memo);
5  } // min_cost_travel()
6
7  int32_t min_cost_helper(const std::unordered_set<int32_t>& days_set,
8                          const std::vector<int32_t>& costs,
9                          int32_t day, std::vector<int32_t>& memo) {
10   if (day < 1) { // base case
11     return 0;
12   } // if
13   if (memo[day] != -1) { // recursive call made before
14     return memo[day];
15   } // if
16   else { // solve and store in memo
17     // if not a travel day, forward propogate cost from previous day
18     if (!days_set.count(day)) {
19       return memo[day] = min_cost_helper(days_set, costs, day - 1, memo);
20     } // if
21     else {
22       return memo[day] = std::min({
23         min_cost_helper(days_set, costs, day - 1, memo) + costs[0],
24         min_cost_helper(days_set, costs, day - 7, memo) + costs[1],
25         min_cost_helper(days_set, costs, day - 30, memo) + costs[2]
26       });
27     } // else
28   } // else
29 } // min_cost_helper()
```

A bottom-up solution is shown below:

```
1  int32_t min_cost_travel(const std::vector<int32_t>& days, const std::vector<int32_t>& costs) {
2    std::vector<int32_t> memo(days.back() + 1);
3    memo[0] = 0; // not needed since default init, but included for clarity
4    std::unordered_set<int> days_set(days.begin(), days.end());
5    for (int32_t i = 1; i < memo.size(); ++i) {
6      if (days_set.count(i)) {
7        memo[i] = std::min({
8          memo[i - 1] + costs[0],
9          memo[std::max(0, i - 7)] + costs[1],
10         memo[std::max(0, i - 30)] + costs[2]
11       });
12     } // if
13     else {
14       memo[i] = memo[i - 1];
15     } // else
16   } // for i
17   return memo.back();
18 } // min_cost_travel()
```

The time and auxiliary space complexity of this dynamic programming solution is $\Theta(n)$, where n is the final travel day.

Example 23.11 You are given an integer k and an integer array `prices`, where `prices[j]` represents the price of a given stock on the j^{th} day. Implement a function that returns the maximum profit you can attain, if you are only allowed to complete at most k transactions. A buy + sell operation is considered together as part of a single transaction. You are not allowed to engage in multiple transactions simultaneously; that is, you must sell the stock before you can buy again. For this problem, you are also not allowed to sell a stock before buying it (sorry, no short selling allowed!).

Similar to the previous problem, we can solve this question by first identifying the choices that can be made on each day, and then identifying a recursive relationship that allows us to build up an optimal solution using these choices. Since each "buy" must always be paired with a "sell" later down the line, we can simplify our recurrence by considering buying and selling in unison (rather than as separate actions); this allows us to define the problem in terms of individual transactions instead of separate buy and sell operations. In this case, there are two choices we can make on each day j:

- We can do nothing on the j^{th} day.
- We can complete a transaction on the j^{th} day.

Let us define $F(i, j)$ as the maximum profit we can earn from having performed a total of i transactions on the j^{th} day. If we do *nothing* on the j^{th} day, then the maximum profit we can earn given i transactions would be

$$F(i, j-1)$$

This is because the maximum profit we can earn from performing i transactions up to day j is the same as the maximum profit we can earn from performing i transactions up to day $j-1$ if we do not complete any transaction on day j.

[8] Note that `std::min()` (lines 22-26) must take in an initializer list if you want to find the minimum of more than two values.

On the other hand, if we want to complete a transaction on the j^{th} day, we must *sell* the stock on day j (this is because we are considering buy/sell together as a single transaction). Since the problem prohibits us from engaging in multiple transactions at the same time, this means that we must have purchased the stock on some day d, where $d < j$, for us to be able to sell it on day j. Assuming i transactions are allowed, the maximum profit we can earn if we buy on day d and sell on day j is

$$(\text{prices}[j] - \text{prices}[d]) + F(i-1, d)$$

Note that this equation includes the profit from performing the i^{th} transaction (prices[j] - prices[d]), plus the optimal profit from performing $i-1$ transactions up to day d (this is because all $i-1$ previous transactions must be finished before day d since simultaneous transactions are not allowed). Therefore, to identify the best profit we can earn from completing a transaction on the j^{th} day, we just have to find the optimal day d from 0 to $j-1$ that maximizes our profit:

$$\max_{0 \le d \le j-1} ((\text{prices}[j] - \text{prices}[d]) + F(i-1, d))$$

This forms the basis of our recurrence relation. To determine the maximum profit attainable from i transactions by the j^{th} day, we simply take the better outcome between doing nothing on the j^{th} day and completing a transaction on the j^{th} day. The base cases occur when the number of transactions i is 0 (which results in a profit of 0) and when j is 0 (which also results in a profit of 0 since you don't have any days to trade).

$$F(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max(F(i,j-1), \max_{0 \le d \le j-1}(\text{prices}[j] - \text{prices}[d]) + F(i-1,d)), & \text{if } i > 0 \text{ or } j > 0 \end{cases}$$

This recurrence relation exhibits the property of overlapping subproblems, so dynamic programming can be used. Using this recurrence relation, we can write a top-down solution as shown below. Since each subproblem depends on two changing variables i and j — where i can take on values from 0 to the number of transactions k, and j can take on values from 0 to the number of trading days n — there are a total of $(i+1) \times (j+1)$ subproblems we many encounter. Thus, we can initialize a memo of size $(k+1) \times (n+1)$ to store our subproblems, where memo[i][j] stores the solution to the subproblem $F(i,j)$.

```
1  int32_t max_profit(int32_t k, const std::vector<int32_t>& prices) {
2    std::vector<std::vector<int32_t>> memo(k + 1, std::vector<int32_t>(prices.size() + 1, -1));
3    return max_profit_helper(k, prices, k, prices.size(), memo);
4  } // max_profit()
5
6  int32_t max_profit_helper(int32_t k, const std::vector<int32_t>& prices, int32_t i, int32_t j,
7                            std::vector<std::vector<int32_t>>& memo) {
8    if (i == 0 || j == 0) {
9      return 0;
10   } // if
11   if (memo[i][j] != -1) {
12     return memo[i][j];
13   } // if
14   int32_t best = max_profit_helper(k, prices, i, j - 1, memo);
15   for (int32_t d = 1; d <= j; ++d) {
16     best = std::max(best, prices[j - 1] - prices[d - 1] +
17                     max_profit_helper(k, prices, i - 1, d, memo));
18   } // for d
19   return memo[i][j] = best;
20 } // max_profit_helper()
```

If we use a bottom-up approach, we would start with our base cases and build our solution upwards using the recurrence. We start by initializing all cells in row and column 0 of our memo with our base case value of 0. Then, we would iterate over our memo row by row, applying the rules of our recurrence relation to solve each subproblem and fill out its corresponding cell in the memo. For example, consider this partially filled memo given $k = 3$ and `prices = [3, 5, 2, 7, 4, 1, 8, 6, 9]`.

		3	5	2	7	4	1	8	6	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	5	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	7				
$i = 3$	0										

Suppose we want to calculate what goes in memo[2][7], or the best profit we can attain from 2 transactions up the 7th day. If we do nothing on the 7th day, then the profit we earn would be equal to the best profit we can attain from 2 transactions up to the 6th day, or the value in the cell directly to the left:

	3	5	2	7	4	1	8	6	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	5	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	**7**				
$i = 3$	0										

If we decide to complete a transaction on the 7th day, then the profit we earn would be equal to the best profit we can attain from 1 transaction completed by some optimal day $d < 7$, plus the profit from completing the transaction on day 7. This requires us to iterate over all columns in the $i = 1$ row up to day 6 and identify the optimal day to complete the first transaction:

	3	5	2	7	4	1	**8**	6	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	5	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	**7**				
$i = 3$	0										

In this case, the best profit from completing the final transaction on day 7 can be obtained by completing the first transaction on day 4 for a profit of 5 (this result has already been computed in memo[1][4]), and then completing the second transaction on day 7 for a profit of 7, for a combined profit of $5 + 7 = 12$. Therefore, we would earn 7 (i.e., memo[2][6]) if we did nothing on the 7th day, and 12 (i.e., prices[7] - prices[6] + memo[1][6]) if we completed our final transaction on the 7th day. The latter outcome is better, so memo[2][7] would store a value of 12.

	3	5	2	7	4	1	**8**	6	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	**5**	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	7	**12**			
$i = 3$	0										

The process repeats for the remaining cells. For instance, when determining the value of memo[2][8] (the best profit attainable from completing the final transaction on day 8), we can either do nothing on the 8th day to earn a profit of memo[2][7] = 12, or we can complete the second transaction on the 8th day. However, if we loop over all the columns in the $i = 1$ row up to day 7 and compute the optimal profit from completing the second transaction on day 8, there is no solution that ends up better than 12. Therefore, the best decision would be to do nothing on day 8, so memo[2][8] also stores a value of 12.

	3	5	2	7	4	1	8	**6**	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	5	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	7	12	**12**		
$i = 3$	0										

After filling out the entire memo, the solution to the original problem can be found in the bottom right cell. As shown, the optimal profit in our example is 15 (buy on day 3, sell on day 4, buy on day 6, sell on day 7, buy on day 8, and sell on day 9).

	3	5	2	7	4	1	8	6	9	**prices**
	0	1	2	3	4	5	6	7	8	9	*index*
$i = 0$	0	0	0	0	0	0	0	0	0	0	
$i = 1$	0	0	2	2	5	5	5	7	8	8	
$i = 2$	0	0	2	2	7	7	7	12	12	13	
$i = 3$	0	0	2	2	7	7	7	14	14	**15**	

The code for a bottom-up solution is shown below.

```
int32_t max_profit(int32_t k, const std::vector<int32_t>& prices) {
  std::vector<std::vector<int32_t>> memo(k + 1, std::vector<int32_t>(prices.size() + 1));
  for (int32_t i = 1; i <= k; ++i) {
    for (int32_t j = 1; j <= prices.size(); ++j) {
      int32_t best = memo[i][j - 1];
      for (int32_t d = 1; d <= j; ++d) {
        best = std::max(best, prices[j - 1] - prices[d - 1] + memo[i - 1][d]);
      } // for d
      memo[i][j] = best;
    } // for j
  } // for i
  return memo[k][prices.size()];
} // max_profit()
```

Since there are total of $\Theta(kn)$ subproblems you may encounter (where k is the number of transactions and n is the number of days in the `prices` vector), and the time to solve each subproblem takes up to $\Theta(n)$ (as you have to loop over the previous row with each subproblem), the overall time complexity of this implementation is $\Theta(kn^2)$. Since we use a $(k+1) \times (n+1)$ memo, the auxiliary space of this implementation is $\Theta(kn)$.

Remark: The above solution is actually not the most efficient way to solve the problem, and a better $\Theta(kn)$ solution exists. This improved solution relies on an observation that allows you to compute the best profit from completing a transaction *in constant time* without needing to loop over the previous row!

To illustrate this, suppose we want to find the best profit of completing 2 transactions by day 3. We would need to loop over values of d from 0 to 2 and compute the maximum of the following:

$$\begin{aligned}&\text{prices[3] - prices[0]} + F(1,0)\\&\text{prices[3] - prices[1]} + F(1,1)\\&\text{prices[3] - prices[2]} + F(1,2)\end{aligned}$$

Since prices[3] is constant across all three, the value of d with the largest value of (-prices[d] + $F(1,d)$) would yield the maximum profit. Now, suppose we want to solve for the best profit of completing 2 transactions by day 4. Using our current implementation, we would loop over values of d from 0 to 3 and compute the maximum of the following:

$$\begin{aligned}&\text{prices[4] - prices[0]} + F(1,0)\\&\text{prices[4] - prices[1]} + F(1,1)\\&\text{prices[4] - prices[2]} + F(1,2)\\&\text{prices[4] - prices[3]} + F(1,3)\end{aligned}$$

However, we already computed $\max_{0 \le d \le 2}$ (-prices[d] + $F(1,d)$) when solving the previous subproblem! Thus, we do not need to loop over the previous row again — we can instead compare the value of (-prices[3] + $F(1,3)$) with the value of $\max_{0 \le d \le 2}$ (-prices[d] + $F(1,d)$) we previously computed and add the larger of the two to prices[4] to get the maximum profit from completing a transaction on day 4. In general, if we keep track of the largest value of (-prices[d] + $F(i-1,d)$) we've seen for each value of i, we can use it to compute the best profit of later subproblems without needing an additional loop! The updated solution is shown below:

```
int32_t max_profit(int32_t k, const std::vector<int32_t>& prices) {
  std::vector<std::vector<int32_t>> memo(k + 1, std::vector<int32_t>(prices.size() + 1));
  for (int32_t i = 1; i <= k; ++i) {
    int32_t temp_max = !prices.empty() ? -prices[0] : 0;
    for (int32_t j = 1; j <= prices.size(); ++j) {
      memo[i][j] = std::max(memo[i][j - 1], prices[j - 1] + temp_max);
      temp_max = std::max(temp_max, memo[i - 1][j - 1] - prices[j - 1]);
    } // for j
  } // for i
  return memo[k][prices.size()];
} // max_profit()
```

With this change, each subproblem only takes $\Theta(1)$ time to solve, which brings our overall time complexity down to $\Theta(kn)$.

Remark: Although we won't discuss it in this chapter, the traveling salesperson problem can also be solved using dynamic programming. The *Held-Karp algorithm* is an algorithm that uses dynamic programming to solve TSP, and it runs in $\Theta(2^n n^2)$ time and uses $\Theta(n2^n)$ auxiliary space. In theory, this is asymptotically faster than the $\Theta(n!)$ worse-case time complexity of branch-and-bound. However, this "improved" performance isn't really discernible in practice, and the massive space complexity typically makes Held-Karp impractical for larger input sizes, compared to a branch-and-bound solution.

※ 23.3.3 *Decision Making: Take It or Leave It*

Some dynamic programming problems require you to choose a subset of items to include in a solution given a set of constraints, where you have to decide if each input value should be included or excluded to attain a desired result. To solve these problems, iterate over each item and compare previous states where the item is and isn't used, and use the solutions to these subproblems to determine if you should include the item or exclude it. A few examples are covered below.

Example 23.12 You are a robber planning to rob houses along a street. Each house has a certain amount of money stashed, stored in an integer array `houses`. However, you are not allowed to rob *adjacent* houses, since adjacent houses have security systems that will automatically alert the police if both houses are broken into on the same night. Write a function that takes in the `houses` array and returns the maximum amount of money you can rob in one night without alerting the police.

At each house, you have two choices you can make: you can either rob the house, or you can choose not to rob it. As a result, this is a decision problem, and the decision you make is dependent on the previous houses you decide to rob. If the house you are considering to rob is located at index i, then the choice to rob that house means that you cannot rob the house located at index $i-1$ (but you can rob the house at index $i-2$ and the optimal houses up to that house). On the other hand, if you choose not to rob the house at index i, you will be able to rob the house at index $i-1$ and all optimal houses up to that point. Therefore, the decision to rob the house at index i boils depends on which is more profitable:

- Rob the house at index i and combine the money with the optimal amount from robbing up to house $i-2$
- Do not rob the house at index i and keep the optimal amount from robbing up to house $i-1$

This gives us the following recurrence relation, where $F(i)$ represents the optimal profit attainable from robbing houses up to index i. The base case occurs when $i < 0$, since you earn 0 profit if you cannot rob any house at all.

$$F(i) = \begin{cases} 0, & \text{if } i < 0 \\ \max(F(i-2) + \text{houses}[i], F(i-1)), & \text{if } i \geq 0 \end{cases}$$

A top-down solution is shown below:

```
int32_t rob_houses(const std::vector<int32_t>& houses) {
  std::vector<int32_t> memo(houses.size(), -1);
  return rob_helper(houses, houses.size() - 1, memo);
} // rob_houses()

int32_t rob_helper(const std::vector<int32_t>& houses, int32_t idx, std::vector<int32_t>& memo) {
  if (idx < 0) {
    return 0;
  } // if
  if (memo[idx] != -1) {
    return memo[idx];
  } // if
  else {
    return memo[idx] = std::max(rob_helper(houses, idx - 2, memo) + houses[idx],
                                rob_helper(houses, idx - 1, memo));
  } // else
} // rob_helper()
```

A bottom-up solution is shown below:

```
int32_t rob_houses(const std::vector<int32_t>& houses) {
  if (houses.size() == 0) {
    return 0;
  } // if
  if (houses.size() == 1) {
    return houses[0];
  } // if
  std::vector<int32_t> memo(houses.size());
  memo[0] = houses[0];
  memo[1] = std::max(houses[0], houses[1]);
  for (int32_t i = 2; i < houses.size(); ++i) {
    memo[i] = std::max(memo[i - 2] + houses[i], memo[i - 1]);
  } // for i
  return memo.back();
} // rob_houses()
```

Given n houses, there are n potential subproblems that you will need to solve, each of which takes $\Theta(1)$ time. Since each subproblem is solved at most once using dynamic programming, the time complexity of the above solution is $\Theta(n)$. Similarly, since a memo of size $\Theta(n)$ is declared, the auxiliary space usage is $\Theta(n)$.

Example 23.13 Given an array of non-negative integers and a target sum x, write a function that determines if there exists a subset of the given set with a sum equal to x.

Example: Given the array `[6, 9, 10, 15, 36, 44, 68]` and a target of 104, you would return `true`, since $9 + 15 + 36 + 44 = 104$.

Since we choosing which numbers to include in our subset, this is a decision making problem. Therefore, we can consider each value in the array one-by-one and use the solutions of smaller subproblems to determine whether each value should be included or excluded from our solution. How do we know whether a value should be included or excluded? To answer this, consider the value 68 in the example above. Notice that the inclusion of 68 in the subset sum depends on the following two subproblems:

- Can the previous values `[6, 9, 10, 15, 36, 44]` sum up to 104? (If so, exclude 68 in solution.)
- Can the previous values `[6, 9, 10, 15, 36, 44]` sum up to $104 - 68 = 36$? (If so, include 68 in solution.)

If the previous numbers we can choose from can already sum up to our target value of 104, then we know that the solution to the entire problem must be `true`. However, if the previous numbers cannot sum to 104, then including 68 can help sum to 104 *only if the previous numbers can sum up to the difference between 104 and 68, or 36*. If the previous numbers cannot sum to 36, then there is no way to include 68 to attain our target sum of 104.

This idea can be applied to every value in our input array. If we define v_{n-1} as the value at index $n-1$ of the array (using 0-indexing) and $F(n,x)$ as whether the first n values in our input array can sum up to a target value of x, we can express the problem using the following recurrence relation.

$$F(n,x) = \begin{cases} \text{true}, & \text{if } n = 0, x = 0 \\ \text{false}, & \text{if } n = 0, x \neq 0 \\ F(n-1,x) \text{ or } F(n-1,x-v_{n-1}), & \text{otherwise} \end{cases}$$

Here, $F(n-1,x)$ represents whether the previous $n-1$ values can already sum to x, and $F(n-1,x-v_{n-1})$ represents whether the previous $n-1$ values can sum to $x-v_{n-1}$. If any of these two are true, then $F(n,x)$ is also true; otherwise it is false.

A top-down solution is shown below. This solution uses a `std::unordered_map<>` to keep track of subproblems because Booleans only take on a value of `true` or `false`, and we want to differentiate between subproblems that return `false` and subproblems that have not been encountered before.

```
1   bool subset_sum(const std::vector<int32_t>& nums, int32_t target) {
2     std::vector<std::unordered_map<int32_t, bool>> memo(nums.size() + 1);
3     return subset_sum_helper(nums, nums.size(), target, memo);
4   } // subset_sum()
5
6   bool subset_sum_helper(const std::vector<int32_t>& nums, int32_t idx, int32_t target,
7                          std::vector<std::unordered_map<int32_t, bool>>& memo) {
8     if (idx == 0) {
9       return target == 0;
10    } // if
11    if (target < 0) {
12      return false;
13    } // if
14    if (memo[idx].count(target)) {
15      return memo[idx][target];
16    } // if
17    else {
18      return memo[idx][target] = subset_sum_helper(nums, idx - 1, target, memo) ||
19                                 subset_sum_helper(nums, idx - 1, target - nums[idx - 1], memo);
20    } // else
21  } // subset_sum_helper()
```

A bottom-up solution is shown below:

```
1   bool subset_sum(std::vector<int32_t>& nums, int32_t target) {
2     std::vector<std::vector<bool>> memo(nums.size() + 1, std::vector<bool>(target + 1));
3     for (int32_t i = 0; i <= n; ++i){
4       memo[i][0] = true;
5     } // for i
6     for (int32_t i = 1; i <= target; ++i) {
7       memo[0][i] = false;
8     } // for i
9     for (int32_t i = 1; i <= n; ++i) {
10      for (int32_t j = 1; j <= target; ++j) {
11        memo[i][j] = memo[i - 1][j];
12        if (j >= nums[i - 1]) {
13          memo[i][j] = memo[i - 1][j] || memo[i - 1][j - nums[i - 1]];
14        } // if
15      } // for j
16    } // for i
17    return memo[nums.size()][target];
18  } // subset_sum()
```

There are a total of $\Theta(nx)$ subproblems we may need to solve, each taking $\Theta(1)$ time. Since the memo allows us to solve each subproblem at most once, the overall time complexity of our dynamic programming solution is therefore $\Theta(nx)$. Similarly, since we are using a $(n+1)\times(x+1)$ memo, the auxiliary space used by this implementation is also $\Theta(nx)$.

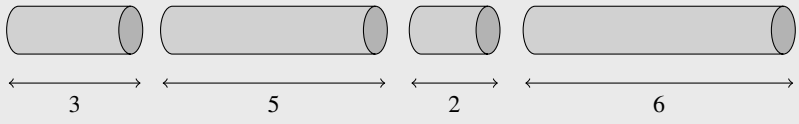
Remark: One of the best examples of a decision making problem that can be solved using dynamic programming is the *0-1 knapsack problem*, which involves finding the best subset of items to fit in a limited capacity knapsack to maximize overall value. However, this problem will not be discussed here since it will get its own chapter… so if you are interested in seeing another example of this dynamic programming pattern, you can read about it in the next chapter!

※ 23.3.4 *Interval Merging*

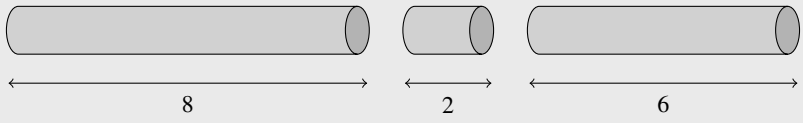
Interval merging is another pattern you may see when working through dynamic programming problems. In these problems, you are given a collection of items that you have to merge together with optimal cost. To solve interval merging questions, you should identify the optimal merging strategy for all possible subintervals and combine these solutions to determine the best way to merge the entire input. As an example, one common approach for solving these problems is to iterate over each position in the given interval, recursively solve the subproblems corresponding to the intervals to the left and right of this position, and then combine these solutions in a manner that will allow you to obtain the optimal cost. We will look at a few examples below.

Example 23.14 You are trying to weld together a series of n pipes with lengths $w_0, w_1, \ldots, w_{n-1}$. The cost to weld together two pipes of lengths w_i and w_j is $\max(w_i, w_j)$, and welding these pipes creates a new pipe of length $w_i + w_j$. Implement a function that returns the minimum cost required to weld together all the pipes, given the constraint that only adjacent pipes can be welded together. That is, pipes 1 and 2 can be welded together, but pipes 1 and 3 cannot.

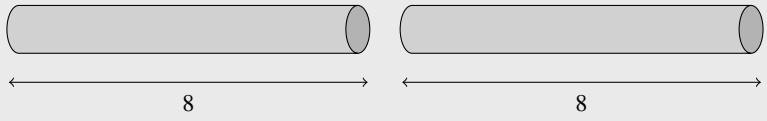
Example: Given pipes of lengths `[3, 5, 2, 6]`:



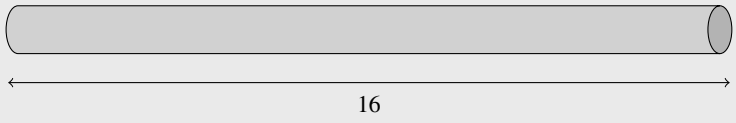
You would first merge together the pipes with lengths 3 and 5, for a running cost of $\max(3, 5) = 5$.



Then, you would merge the pipes with lengths 2 and 6 together, for a running cost of $5 + \max(2, 6) = 11$.

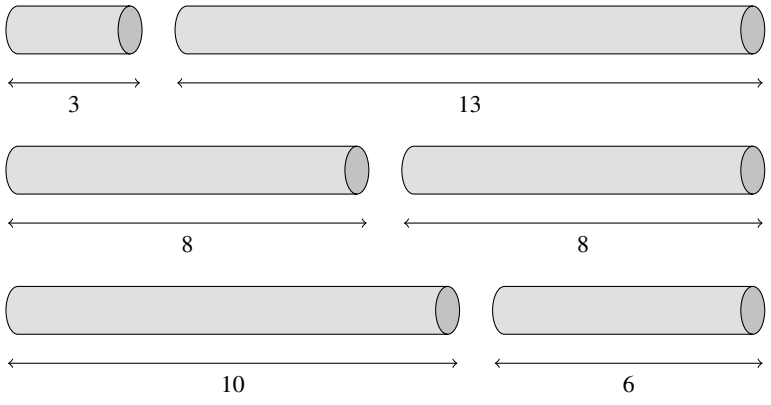


Lastly, you would merge the remaining two pipes together, for a running cost of $11 + \max(8, 8) = 19$. This is the minimum cost required to merge all the pipes together, so your function would return 19.



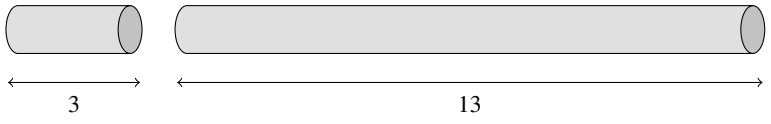
A dynamic programming solution to this problem, if there even is one, is not inherently obvious. Since this is an optimization problem, you could check if a greedy approach works first. However, you would quickly see that there is no viable greedy algorithm that always produces an optimal solution, as only adjacent pipes can be welded together (had this restriction been removed, then the greedy approach of welding the smallest pipes together would work).

Since greedy does not work, let's try to come up with a dynamic programming solution instead. A good starting point is to think about ways we can break the problem up into smaller instances of the same problem, and use these solutions to build up the solution for a larger problem. To come up with our subproblems, we can make the following observation: *regardless of how the pipes are welded together, there must always exist a position that is welded last*. By choosing where we want to complete our final weld, we can split the remaining pipe into two independent subproblems. For instance, there are three positions we can choose for the last weld using the example above:



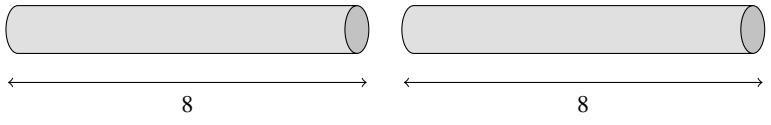
If we define $F(x, y)$ as the optimal cost of welding all pipes from pipe x up to pipe y (inclusive, using 0-indexing), we can recursively assign a cost for each of these final welds. The optimal cost to merge all the pipes if our final weld occurs directly after pipe 0 is

$$\max(3,\ 13) + F(0,0) + F(1,3)$$



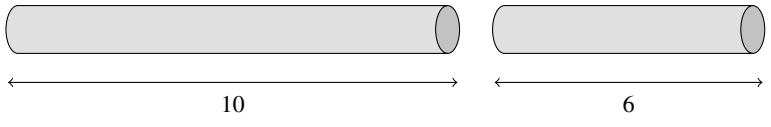
The optimal cost to merge all the pipes if our final weld occurs directly after pipe 1 is

$$\max(8,\ 8) + F(0,1) + F(2,3)$$



The optimal cost to merge all the pipes if our final weld occurs directly after pipe 2 is

$$\max(10,\ 6) + F(0,2) + F(3,3)$$



In general, if you are given n pipes, the optimal merge cost if the final weld occurs directly after pipe k can be expressed as

$$\underbrace{\max(\sum_{i=0}^{k} w_i,\ \sum_{j=k+1}^{n-1} w_j)}_{\substack{\text{cost to weld}\\ \text{pipes } k \text{ and } k+1}} + \underbrace{F(0,k)}_{\substack{\text{cost to weld all}\\ \text{pipes from 0 to } k}} + \underbrace{F(k+1,n-1)}_{\substack{\text{cost to weld all}\\ \text{pipes from } k+1 \text{ to } n\text{-}1}}$$

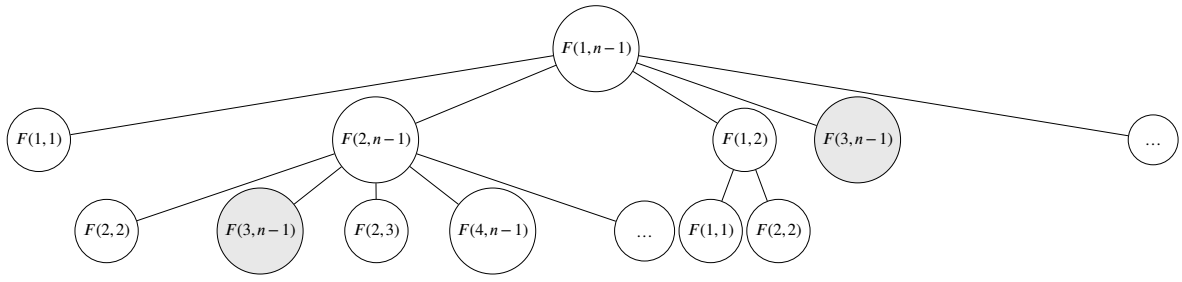
Therefore, to determine the optimal cost to merge all pipes from pipe x up to pipe y, we would solve the above equation for all possible values of k and take the minimum cost. This results in the recurrence relation shown below. The base case for $F(x, y)$ occurs when $x = y$, since it takes 0 cost to merge a pipe with itself.

$$F(x,y) = \begin{cases} 0, & \text{if } x = y \\ \min_{x \le k < y}(\max(\sum_{i=x}^{k} w_i, \sum_{j=k+1}^{y} w_j) + F(x,k) + F(k+1,y)), & \text{otherwise} \end{cases}$$

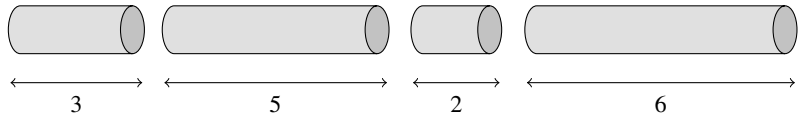
Since the problem requires us to find the optimal cost of merging all pipes from pipe 0 to pipe $n-1$, we want to solve for $F(0, n-1)$. If we were to write a naïve recursive solution for this problem, we would get something like this:

```
1  int32_t weld_pipes(const std::vector<int32_t>& pipes) {
2    return weld_helper(pipes, 0, pipes.size() - 1);
3  } // weld_pipes()
4
5  int32_t weld_helper(const std::vector<int>& pipes, int32_t first, int32_t last) {
6    if (first == last) {
7      return 0; // no work to merge pipe with itself
8    } // if
9    int32_t best = std::numeric_limits<int32_t>::max();
10   for (int32_t k = first; k < last; ++k) {
11     int32_t cost_left = std::accumulate(pipes.begin() + first, pipes.begin() + k + 1, 0);
12     int32_t cost_right = std::accumulate(pipes.begin() + k + 1, pipes.begin() + last + 1, 0);
13     best = std::min(best, std::max(cost_left, cost_right) +
14               weld_helper(pipes, first, k) + weld_helper(pipes, k + 1, last));
15   } // for k
16   return best;
17 } // weld_helper()
```

However, this implementation is inefficient, since there are overlapping subproblems, and we are solving several subproblems more than once (shown by the partial recurrence tree below):



To address this problem, we can use dynamic programming to store the solutions of these repeating subproblems. Since computing each subproblem requires knowledge of both the weight and optimal cost of the pipes to the left and right of the final weld, we want to store both *weight* and *cost* in our memo (or use two separate memos, one for weight and one for cost). This allows us to fetch the weight of any pipe in constant time, without having to complete a separate linear-time summation every time. An example of this memo structure is shown below:



weights

	0	1	2	3
0				
1	■			
2	■	■		
3	■	■	■	

costs

	0	1	2	3
0				
1	■			
2	■	■		
3	■	■	■	

Here, `weights[i][j]` stores the weight of the fully welded pipe from pipe `i` to pipe `j` (i.e., $w_i + w_{i+1} + \ldots + w_j$), and `costs[i][j]` stores the optimal cost of welding together all pipes from pipe `i` up to pipe `j` (i.e., $F(i, j)$). The shaded cells do not need to be filled out since their values are mirrored across the diagonal: `weights[i][j] == weights[j][i]`.

Using the example provided, the memos would be filled out as follows. The solution of the problem would eventually be stored in the top-right cell of the `costs` table, since our goal is to solve for the optimal cost of welding all pipes from pipe 0 to pipe $n-1$, or `costs[0][n-1]`.

weights

	0	1	2	3
0	3	8	10	16
1	■	5	7	13
2	■	■	2	8
3	■	■	■	6

costs

	0	1	2	3
0	0	5	12	**19**
1	■	0	5	12
2	■	■	0	6
3	■	■	■	0

A top-down solution is shown below:

```
1  int32_t weld_pipes(const std::vector<int32_t>& pipes){
2    std::vector<std::vector<int32_t>> weights(pipes.size(), std::vector<int32_t>(pipes.size()));
3    std::vector<std::vector<int32_t>> costs(pipes.size(), std::vector<int32_t>(pipes.size(), -1));
4    return weld_helper(pipes, costs, weights, 0, pipes.size() - 1);
5  } // weld_pipes()
6
7  int32_t weld_helper(const std::vector<int32_t>& pipes, std::vector<std::vector<int32_t>>& costs,
8                      std::vector<std::vector<int32_t>>& weights, int32_t begin, int32_t end){
9    if (costs[begin][end] != -1) {
10     return costs[begin][end];  // no need to redo work
11   } // if
12   if (begin == end){  // base cases
13     costs[begin][end] = 0;
14     weights[begin][end] = pipes[begin];
15     return 0;
16   } // if.
17   int32_t min_cost = std::numeric_limits<int32_t>::max();
18   for (int32_t k = begin; k < end; ++k) {
19     int32_t cost_k = weld_helper(pipes, costs, weights, begin, k) +
20                      weld_helper(pipes, costs, weights, k + 1, end);
21     cost_k += std::max(weights[begin][k], weights[k + 1][end]);
22     min_cost = std::min(min_cost, cost_k);
23   } // for k
24   costs[begin][end] = min_cost;
25   weights[begin][end] = weights[begin][begin] + weights[begin + 1][end];
26   return costs[begin][end];
27 } // weld_helper()
```

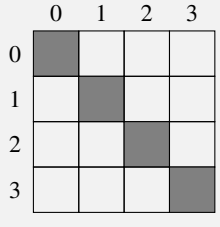
A bottom-up solution is shown below:

```
1  int32_t weld_pipes(const std::vector<int32_t>& pipes) {
2    std::vector<std::vector<int32_t>> weights(pipes.size(), std::vector<int32_t>(pipes.size()));
3    std::vector<std::vector<int32_t>> costs(pipes.size(), std::vector<int32_t>(pipes.size()));
4    for (int32_t col = 0; col < pipes.size(); ++col) {
5      for (int32_t row = col; row >= 0; --row) {
6        if (row == col) { // base cases
7          costs[row][col] = 0;
8          weights[row][col] = pipes[row];
9          continue;
10       } // if
11       int32_t min_cost = std::numeric_limits<int32_t>::max();
12       for (int32_t k = row; k < col; ++i){
13         int32_t cost_k = costs[row][k] + costs[k + 1][col] +
14                          std::max(weights[row][k], weights[k + 1][col]);
15         min_cost = std::min(min_cost, cost_k);
16       } // for k
17       weights[row][col] = weights[row][row] + weights[row + 1][col];
18       costs[row][col] = min_cost;
19     } // for row
20   } // for col
21   return costs[0][pipes.size() - 1];
22 } // weld_pipes()
```

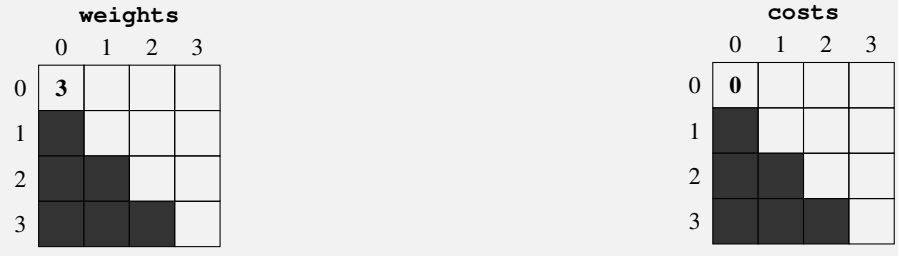
The time complexity of this dynamic programming solution is $\Theta(n^3)$ with $\Theta(n^2)$ auxiliary space, where n is the number of pipes. This is because there are $\Theta(n^2)$ subproblems we may need to solve, each taking $\Theta(n)$ time.

Remark: Unlike the top-down solution, which directly converts the recurrence relation into code, the bottom-up solution may be a bit more difficult to follow. If you are having trouble understanding what is going on in the bottom-up solution, the following provides a step-by-step illustration of what is happening (feel free to skip this if you are comfortable with the code).

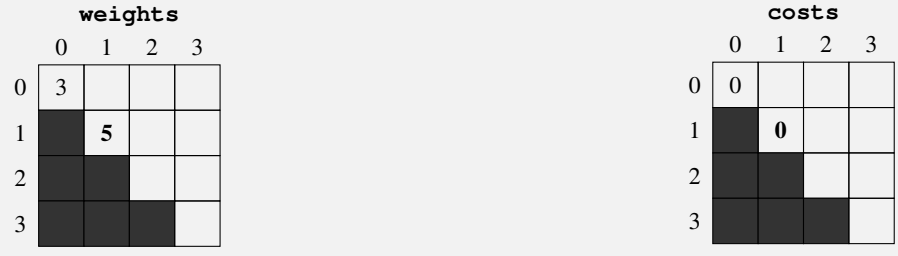
First, notice that the cell at (row i, column j) of the memo corresponds to the optimal cost of merging together all pipes between i and j, inclusive. Since the base case occurs when you try to merge a pipe with itself, each base case is stored along the diagonal of our memo, shaded below. When using bottom-up dynamic programming, we want to solve the base cases before we move on to larger subproblems: this is why the `for` loop on line 5 starts from the center diagonal of the table and iterates upward by row (i.e., `row` = `col` → 0).



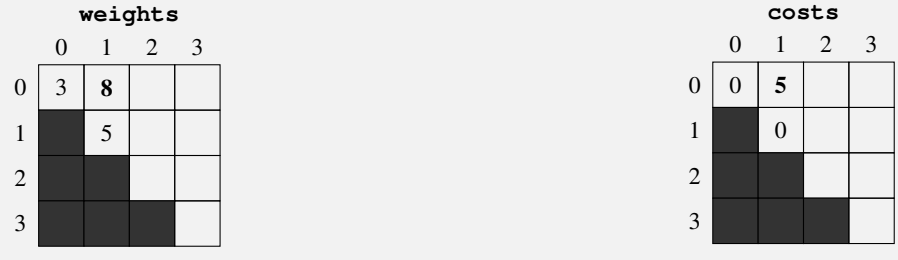
After declaring our weight and cost memos on lines 2 and 3, our bottom-up solution considers each pipe one by one using the `for` loop on line 4. The very first subproblem we consider is (`row` = 0, `col` = 0), which involves welding pipe 0 with itself. This is a base case, as welding a pipe with itself does not change its weight nor has any cost. Thus, we enter our base case condition on line 6 and fill out our memos using the logic on lines 7 and 8:



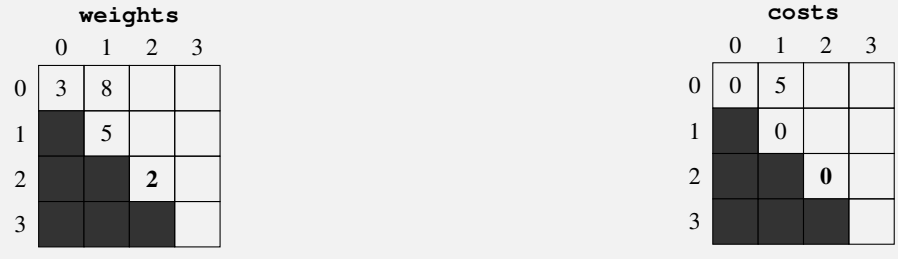
We then increment `col` and consider all subproblems that weld up to pipe 1. First, we solve for (`row` = 1, `col` = 1), which is the cost of welding pipe 1 with itself. Similar to (`row` = 0, `col` = 0), we fill out the corresponding cells as base cases.



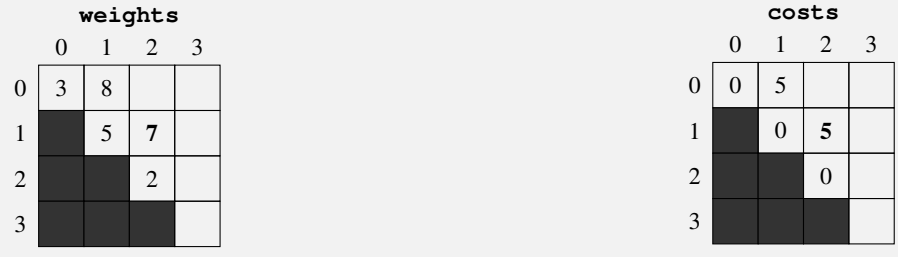
Then, we decrement `row` and consider the subproblem (`row` = 0, `col` = 1), which represents the optimal weight and cost of welding pipes 0 and 1. This optimal cost is equal to the optimal cost of welding pipes 0 and 0 + the optimal cost of welding pipes 1 and 1 + the cost of welding pipes 0 and 1. This result is computed on lines 13-15 and saved in the memo.



Now that we have filled out the entirety of column 1, we increment `col` and consider all subproblems that weld up to pipe 2. First, we solve for (`row` = 2, `col` = 2), which is our base case:



Then, we decrement `row` and consider the subproblem (`row` = 1, `col` = 2), which is the optimal cost of welding pipes 1 and 2. This optimal cost is equal to the optimal cost of welding pipes 1 and 1 + the optimal cost of welding pipes 2 and 2 + the cost of welding pipes 1 and 2.



We decrement `row` again and consider the subproblem (`row` = 0, `col` = 2), which is the optimal cost of welding pipes 0 *through* 2. This is a bit more complex to solve, since there are two ways to weld all the pipes from 0 to 2: we can either

- weld pipe 0 with pipe 1, then weld this combined pipe with pipe 2
- weld pipe 1 with pipe 2, then weld this combined pipe with pipe 0

This is what the `for` loop on line 12 handles — it loops through all the possible locations of the final weld (for this subproblem, 0 and 1), calculates the cost to weld, and stores the minimum cost in the memo. In this example, the optimal cost of welding 0 and 1 first is equal to 5 + 8 = 13 (5 from merging pipes 0 and 1, and 8 from merging the combined pipe with pipe 2 — both values can be retrieved from (row 0, column 1) of the cost and weight memos). On the other hand, the optimal cost of welding 1 and 2 first is equal to 5 + 7 = 12 (5 from merging pipes 1 and 2, and 7 from merging the combined pipe with pipe 0). Since 12 is the better cost, it is used as the solution for this subproblem.

weights

	0	1	2	3
0	3	8	**10**	
1		5	7	
2			2	
3				

costs

	0	1	2	3
0	0	5	**12**	
1		0	5	
2			0	
3				

We repeat this procedure for the final pipe, filling out column 3 from the base case (`row` = 3, `col` = 3) up to the final solution (`row` = 0, `col` = 3). The value at (`row` = 0, `col` = 3) of the cost memo is then returned.

Example 23.15 You are given a rod of length n inches and the prices of all rod prices with length less than or equal to n. Write a function that calculates the maximum value obtainable from cutting up the rod and selling the pieces. (Assume you can only cut the rod to form integer lengths.)

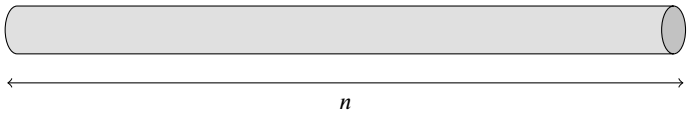
Example: Given a rod of length $n = 8$ and the following prices:

length	0	1	2	3	4	5	6	7	8
price	0	1	4	7	9	13	14	15	19

you would return 20, since this is the maximum value you can earn from selling the rod (by cutting the rod into subrods with lengths 3 and 5, and selling them for 7 and 13, respectively).

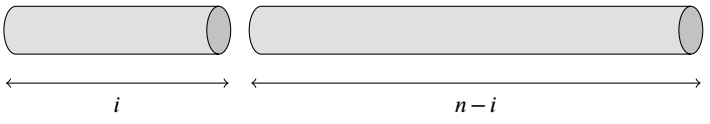
One method for solving this problem would be to consider all possible ways to cut the rod and choose the option that is best. However, given a rod of length n, there are a total of 2^{n-1} ways to cut the rod: this is because there are $n-1$ places we can make a cut, and we have two decisions we can make at each of these $n-1$ positions (cut or no cut). Therefore, this brute force solution would take exponential time, which isn't viable if we want to solve the problem efficiently.

To improve our solution, we can use a dynamic programming approach instead. First, we will need to find a way to break our problem up into smaller, overlapping subproblems. To illustrate how this is done, let's consider the following rod of length n.



Let's assume we make the first cut at some length i on the rod. If we let p_i represent the price of a rod with length i, and let $F(n)$ represent the maximum profit we can earn from a rod of length n, the total earnings from this specific cut can be defined as

$$\text{maximum profit if first cut at length } i = p_i + F(n-i)$$



From this, we can see that the maximum profit attainable from a rod of length n, or $F(n)$, can be computed by finding the maximum $p_i + F(n-i)$ across all possible values of i from 1 to n. Therefore, the recurrence relation we end up with is

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ \max_{1 \le i \le n}(p_i + F(n-i)), & \text{if } n > 0 \end{cases}$$

Converting this naïve implementation to code, we would get the following:

```
int32_t cut_rod(const std::vector<int32_t>& prices, int32_t n) {
  if (n == 0) {
    return 0;
  } // if
  int32_t best = std::numeric_limits<int32_t>::min();
  for (int32_t i = 1; i <= n; ++i) {
    best = std::max(best, prices[i] + cut_rod(prices, n - i));
  } // for i
  return best;
} // cut_rod()
```

However, we end up with a bunch of overlapping subproblems, since every call to $F(i)$ makes additional recursive calls to $F(i-1), F(i-2), \ldots,$ all the way down to $F(0)$. To prevent our algorithm from performing duplicate work, we can store the results of our subproblems in a memo. A top-down approach is shown below:

```
int32_t cut_rod(const std::vector<int32_t>& prices, int32_t n) {
  std::vector<int32_t> memo(n + 1, -1);
  return cut_rod_helper(prices, n, memo);
} // cut_rod()

int32_t cut_rod_helper(const std::vector<int32_t>& prices, int32_t n, std::vector<int32_t>& memo) {
  if (n == 0) {
    return 0;
  } // if
  if (memo[n] != -1) {
    return memo[n];
  } // if
  else {
    int32_t best = std::numeric_limits<int32_t>::min();
    for (int32_t i = 1; i <= n; ++i) {
      best = std::max(best, prices[i] + cut_rod_helper(prices, n - i, memo));
    } // for i
    return memo[n] = best;
  } // else
} // cut_rod_helper()
```

A bottom-up solution is shown below:

```
int32_t cut_rod(const std::vector<int32_t>& prices, int32_t n) {
  std::vector<int32_t> memo(n + 1, -1);
  memo[0] = 0;
  for (int32_t j = 1; j <= n; ++j) {
    int32_t best = std::numeric_limits<int32_t>::min();
    for (int32_t i = 1; i <= j; ++i) {
      best = std::max(best, prices[i] + memo[j - i]);
    } // for i
    memo[j] = best;
  } // for j
  return memo[n];
} // cut_rod()
```

The time complexity of this dynamic programming implementation is $\Theta(n^2)$, since each subproblem $F(i)$ runs through i iterations of a `for` loop (from i to 0), producing n^2 iterations in total for an initial input size of n. Additionally, the auxiliary space used by this solution is $\Theta(n)$, since an array of size n is declared to store our subproblems.

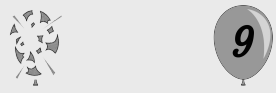
Example 23.16 You are given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number stored in an array `value`. If you burst the `i`$^{\text{th}}$ balloon, you will earn points based on the product of the popped ballon and its adjacent balloons. Write a function that returns the maximum number of points you can earn from bursting all the balloons.

For instance, suppose there are four balloons, with values 4, 2, 7, and 9.

It is optimal to first pop the balloon with a value of 2, to earn $4 \times 2 \times 7 = 56$ points.

Then, it would be optimal to pop the balloon with a value of 7, to earn $4 \times 7 \times 9 = 252$ points.

Then, it would be optimal to pop the balloon with a value of 4, to earn $4 \times 9 = 36$ points.



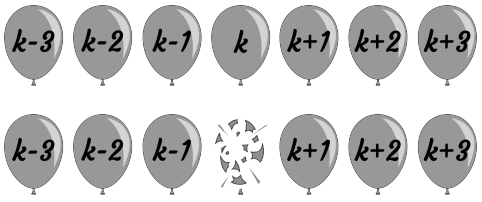
Then, the final balloon is popped to earn 9 points.

All of the balloons have been popped, so your final score is $56 + 252 + 36 + 9 = 353$. This is the highest score you can attain from popping the balloons, so you should return 353.

If you wanted to solve this problem using brute force, you would need to iterate over all the popping orders and calculate the number of points you can earn for each order, taking the best at the very end. Given n balloons, however, there are $n!$ unique popping orders. In addition, it takes $\Theta(n)$ time to sum up the number of points earned for each order, resulting in an overall time complexity of $\Theta(n \times n!)$. Is there a way to do better?

Since this is an optimization problem, we could try to see if a greedy solution exists first. However, if you were to try some approaches out, you would see that there is no greedy strategy that satisfies all possible examples. (One common incorrect approach is to pop from smallest to largest among the balloons not at the ends, and then pop the final two balloons in ascending order, but this does not work in the case of 2, 8, 2).

Instead, we will have to see if there exists a way to recursively break the problem into smaller subproblems, so that an approach such as divide-and-conquer or dynamic programming can be used. A reasonable starting point would be to consider what would happen if we pop *any* balloon in the sequence:

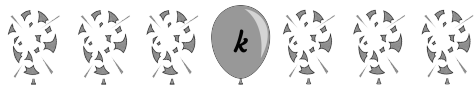


Notice here that we end up with two remaining intervals after the balloon at index k is popped: one subsequence consisting of all balloons to the left of k and one consisting of all balloons to the right. We can therefore think of the problem in terms of intervals, where the optimal score to pop all the balloons in a smaller interval can be used to construct the optimal score of a larger interval. In this case, we want to find a recurrence relation that expresses our optimal solution in terms of the two subsequences that remain after the balloon at index k is popped.

However, there is a flaw with this approach. If we think about the recurrence in terms of the *first* balloon we pop, we run into the issue where the solution of one interval cannot be solved without knowledge of another. For instance, if we popped balloon k and then wanted to find the optimal cost of popping balloon $k-1$, our solution would depend on whether balloon $k+1$ has been popped already, as $k+1$ would be adjacent to $k-1$. This prevents us from being able to solve the subproblems independently!

To fix this issue, we need to reverse our thinking. The reason why the subproblems cannot be solved independently is that balloon k has already been popped, which creates a dependency between balloons $k-1$ and $k+1$. Thus, to be able to solve the subproblems properly, we must keep balloon k unpopped to separate balloons $k-1$ and $k+1$ so that their solutions do not depend on each other. This is the key insight for solving this problem: *because we cannot pop balloon k before solving the left and right subproblems without introducing a dependency between subproblems, we will instead think about our recurrence in terms of the* ***last*** *balloon we pop*. Notice that this is the same idea we used to solve the pipe welding problem covered earlier, where we defined our subproblems using the position of the last weld rather than the first weld.

To illustrate this, suppose there is one balloon remaining to pop, located at index k:



What is the maximum possible score attainable after popping this last balloon? If we define $F(x, y)$ as the maximum score attainable from popping all balloons from index x up to index y, we can express the maximum score from popping all balloons from x to y *if the balloon at index k is popped last* (assuming $x \leq k \leq y$) using the following equation:[9]

$$\underbrace{F(0,k-1)}_{\substack{\text{maximum score from}\\ \text{popping left balloons}}} + \underbrace{F(k+1,n-1)}_{\substack{\text{maximum score from}\\ \text{popping right balloons}}} + \underbrace{\text{value}[x-1]\times\text{value}[k]\times\text{value}[y+1]}_{\substack{\text{points from popping}\\ \text{final balloon}}}$$

To compute the maximum score attainable after popping *all* balloons, we can simply solve the previous equation for all possible final positions k and take the largest solution. This gives us our recurrence relation, as shown below. If x and y are out-of-bounds, we can simply substitute in a value of 1, since anything multiplied with 1 does not change in value.

$$F(x,y) = \begin{cases} 1, & \text{if } x, y < 0 \text{ or } x, y \geq n \text{ (out-of-bounds)} \\ \max_{x \leq k \leq y}(F(x,k-1) + F(k+1,y) + \text{value}[x-1]\times\text{value}[k]\times\text{value}[y+1]), & \text{otherwise} \end{cases}$$

Since this recurrence relation involves overlapping subproblems that may be needed more than once, a dynamic programming approach can be used. We begin our implementation by declaring a memo where the solution to $F(x, y)$ is stored at position memo$[x][y]$. The completed memo for the provided example is shown below, where the final solution $F(0, n-1)$ for n balloons is stored at memo$[0][n-1]$.

	0	1	2	3
0	8	84	344	**353**
1		56	308	344
2			126	144
3				63

A top-down solution is shown below:

```
int32_t get_balloon_value(const std::vector<int32_t>& nums, int32_t index) {
  if (index < 0 || index >= nums.size()) {
    return 1;
  } // if
  return nums[index];
} // get_balloon_value()

int32_t max_points(const std::vector<int32_t>& nums) {
  std::vector<std::vector<int32_t>> memo(nums.size(), std::vector<int>(nums.size(), -1));
  return max_points_helper(nums, 0, nums.size() - 1, memo);
} // max_points()

int32_t max_points_helper(const std::vector<int32_t>& nums, int32_t left, int32_t right,
                          std::vector<std::vector<int32_t>>& memo) {
  if (right < left) {
    return 0;
  } // if
  if (memo[left][right] != -1) {
    return memo[left][right];
  } // if
  int32_t best = 0;
  for (int32_t i = left; i <= right; ++i) {
    int32_t points =
      max_points_helper(nums, left, i - 1, memo) + max_points_helper(nums, i + 1, right, memo) +
      get_balloon_value(nums, left - 1) * get_balloon_value(nums, i) *
      get_balloon_value(nums, right + 1);
    best = std::max(best, points);
  } // for i
  return memo[left][right] = best;
} // max_points_helper()
```

[9] Note that we still need to consider any adjacent balloons outside the range $[x, y]$ that we popped, which is why we multiply the value of the last balloon with value$[x-1]$ and value$[y+1]$ in the above equation. In addition, we will also need to consider the edge case where value$[x-1]$ or value$[y+1]$ goes out-of-bounds; when this happens, we should substitute in a value of 1 to ensure we don't index off the end of the array.

A bottom-up solution is shown below. The loop on line 7 considers all subproblems of each interval length, the loop on line 8 considers all starting positions of the interval to pop, and the loop on line 10 considers all positions of the final pop within each interval.

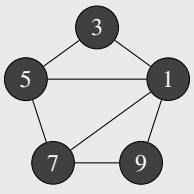
```
1  int32_t get_balloon_value(const std::vector<int32_t>& nums, int32_t index) {
2    return (index < 0 || index >= nums.size()) ? 1 : nums[index];
3  } // get_balloon_value()
4
5  int32_t max_points(const std::vector<int32_t>& nums) {
6    std::vector<std::vector<int32_t>> memo(nums.size(), std::vector<int32_t>(nums.size()));
7    for (int32_t len = 0; len < nums.size(); ++len) {
8      for (int32_t left = 0; left < nums.size() - len; ++left) {
9        int32_t right = left + len;
10       for (int32_t k = left; k <= right; ++k) {
11         int32_t best_left = (k == left) ? 0 : memo[left][k - 1];
12         int32_t best_right = (k == right) ? 0 : memo[k + 1][right];
13         memo[left][right] = std::max(memo[left][right],
14           best_left + best_right + get_balloon_value(nums, left - 1) *
15           get_balloon_value(nums, k) * get_balloon_value(nums, right + 1));
16       } // for k
17     } // for left
18   } // for len
19   return memo[0][nums.size() - 1];
20 } // max_points()
```

Remark: Instead of using a separate `get_balloon_value()` function to keep track of whether an index is out of bounds, an alternative would be to insert a dummy value of 1 at both the beginning and end of the original input vector — this dummy value gets indexed if you go out of bounds (and multiplying any value by 1 does not change the value).

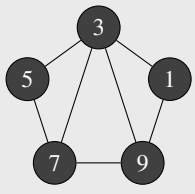
In this dynamic programming approach, there are a total of $\Theta(n^2)$ subproblems to solve, each taking up to $\Theta(n)$ time (since you have to loop across all indices between left and right). Using the memo, each subproblem is solved at most once, so the overall time complexity is worst-case $\Theta(n^3)$. The dimensions of the memo is $n \times n$, so the auxiliary space used by this solution is $\Theta(n^2)$.

Example 23.17 You are given a convex n-sided polygon where each vertex has an integer value. You are given a vector `val` of these values, where `val[i]` is the value associated with the i^{th} vertex (in counterclockwise order). Your goal is to triangulate the polygon into $n-2$ triangles in a way that *minimizes* the sum of the products of each of the triangles' vertices.

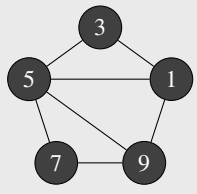
Example: Given the vertex values `[1, 3, 5, 7, 9]`, you would return the sum 113. This is because there are five ways to triangulate a polygon with these values, and this is the minimum sum attainable:



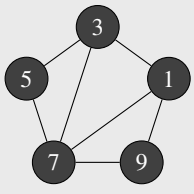
Value of triangulation:
$(1\times3\times5)+(1\times5\times7)+(1\times7\times9)$
$= 15+35+63 = 113$



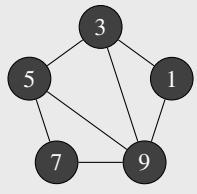
Value of triangulation:
$(3\times5\times7)+(3\times7\times9)+(1\times3\times9)$
$= 105+189+27 = 321$



Value of triangulation:
$(1\times3\times5)+(1\times5\times9)+(5\times7\times9)$
$= 15+45+315 = 375$



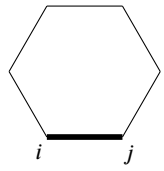
Value of triangulation:
$(3\times5\times7)+(1\times3\times7)+(1\times7\times9)$
$= 105+21+63 = 189$



Value of triangulation:
$(5\times7\times9)+(3\times5\times9)+(1\times3\times9)$
$= 315+135+27 = 477$

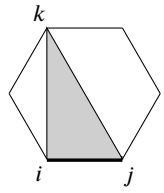
While this may not seem like an interval merging dynamic programming problem at first, it in fact is quite similar to the balloon problem discussed previously. To visualize how the intervals can be defined, we must first make the observation that every edge of the polygon must be part of *exactly one triangle* after any possible triangulation. Why is this the case? If any of a polygon's edges were not part of a triangle, then that polygon would not be fully triangulated. On the other hand, if any edge were part of more than one triangle, then the triangulation would include triangles that overlap. Both of these cases are not valid, so each edge of the polygon can only be part of one triangle after any potential triangulation. You can prove that this is the case by drawing some triangulations out.

Because each edge must be part of exactly one triangle in the final triangulation, we can reduce our problem into smaller subproblems by fixing a single edge and then considering all the possible triangles we can get that includes that edge. To visualize how this works, consider the following polygon, where we fix the edge from vertex i to vertex j.

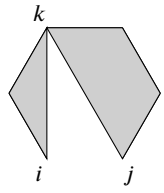


For any polygon with n vertices, there are $n-2$ possible triangles that can be formed using each edge. This is shown for the polygon above:

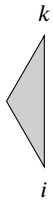
We can use this idea to break our problem down into smaller subproblems. Consider the following triangle that is formed using the edge $\overline{ij}$, where the other vertex is denoted as k. Using the rules of the problem, we know that the score of $\triangle ijk$ is equal to the value of $i \times j \times k$.



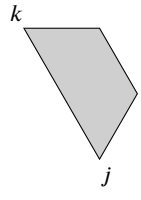
However, this leaves us with two additional polygons to the left and right of $\triangle ijk$. To identify the best possible triangulation that involves $\triangle ijk$, we will also have to know the best way to triangulate these additional polygons.



This where the recursive subproblems come into play! Let us define $F(i,k)$ as the minimum triangulation possible for the polygon spanning the range from vertex i to vertex k, and $F(k,j)$ as the minimum triangulation possible for the polygon spanning the range from vertex k to vertex j.

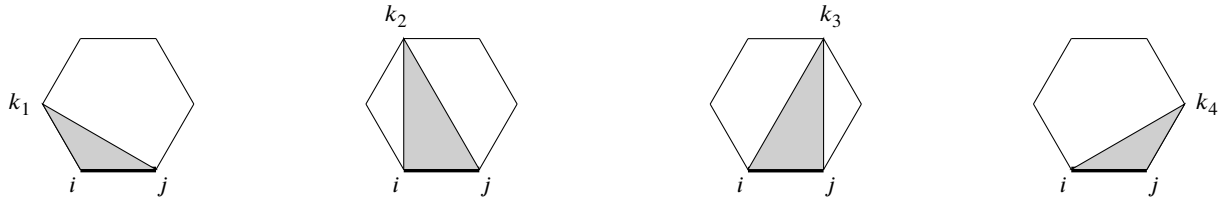


$F(i,k)$



$F(k,j)$

By doing so, we can denote the minimum triangulation that includes $\triangle ijk$ as equal to $F(i,k)+F(k,j)+\text{value}[i]\times\text{value}[j]\times\text{value}[k]$. However, this is just one triangle that can be formed using edge $\overline{ij}$. To determine the best possible triangulation for the *entire* polygon, we would need to identify the minimum triangulation that includes $\triangle ijk$ for *all* $n-2$ possible values of k between i and j.



$$F(i,j) = \min(\underbrace{F(i,k_1)+F(k_1,j)+\text{value}[i]\times\text{value}[j]\times\text{value}[k_1]}_{\text{minimum triangulation involving}\triangle ijk_1} + \underbrace{F(i,k_2)+F(k_2,j)+\text{value}[i]\times\text{value}[j]\times\text{value}[k_2]}_{\text{minimum triangulation involving}\triangle ijk_2} + \underbrace{F(i,k_3)+F(k_3,j)+\text{value}[i]\times\text{value}[j]\times\text{value}[k_3]}_{\text{minimum triangulation involving}\triangle ijk_3} + \underbrace{F(i,k_4)+F(k_4,j)+\text{value}[i]\times\text{value}[j]\times\text{value}[k_4]}_{\text{minimum triangulation involving}\triangle ijk_4})$$

More generally, we can write this recurrence relation for any n-sided polygon as such, where $F(i, j)$ represents the minimum value attainable from triangulating the polygon formed by the vertices from i to j (where $i < j$):

$$F(i,j) = \begin{cases} 0, & \text{if } j - i \leq 1 \text{ (adjacent vertices, so no triangle can be formed)} \\ \min_{i<k<j}(F(i,k) + F(k,j) + \text{value}[i] \times \text{value}[k] \times \text{value}[j]), & \text{otherwise} \end{cases}$$

As this recurrence relation involves overlapping subproblems, we can use a dynamic programming approach to solve it. Since each subproblem is defined in terms of i and j, we will declare a two-dimensional memo such that the solution to $F(i, j)$ is stored at position memo[i][j]. Then, to solve each subproblem, we iterate over all vertices k between i and j and find the minimum value of $F(i,k) + F(k,j) + \text{value}[i] \times \text{value}[k] \times \text{value}[j]$, storing each result in our memo. A top-down solution is shown below:

```
int32_t min_triangulation(const std::vector<int32_t>& values) {
  std::vector<std::vector<int32_t>> memo(values.size(), std::vector<int32_t>(values.size(), -1));
  return min_triangulation_helper(values, 0, values.size() - 1, memo);
} // min_triangulation()

int32_t min_triangulation_helper(const std::vector<int32_t>& values, int32_t i, int32_t j,
                                 std::vector<std::vector<int32_t>>& memo) {
  if (j - i <= 1) {
    return 0;
  } // if
  if (memo[i][j] != -1) {
    return memo[i][j];
  } // if
  int32_t best = std::numeric_limits<int32_t>::max();
  for (int32_t k = i + 1; k < j; ++k) {
    int32_t curr = min_triangulation_helper(values, i, k, memo) +
                   min_triangulation_helper(values, k, j, memo) +
                   values[i] * values[k] * values[j];
    best = std::min(best, curr);
  } // for k
  return memo[i][j] = best;
} // min_triangulation_helper()
```

A bottom-up solution is shown below. Notice that `i` is *decremented* in the outer `for` loop, while `j` is *incremented* in the inner `for` loop. This was done intentionally, as it ensures that all dependent subproblems (in this case, $F(i,k)$ and $F(k,j)$) are already solved *before* we try to solve for the larger subproblem of $F(i, j)$. Remember that the subproblems must be solved in the correct order if you use a bottom-up approach!

```
int32_t min_triangulation(const std::vector<int32_t>& values) {
  std::vector<std::vector<int32_t>> memo(values.size(), std::vector<int32_t>(values.size(), 0));
  for (int32_t i = values.size() - 1; i >= 0; --i) {
    for (int32_t j = i + 1; j < values.size(); ++j) {
      int32_t& best = memo[i][j];
      for (int32_t k = i + 1; k < j; ++k) {
        int32_t curr = best == 0 ? std::numeric_limits<int32_t>::max() : best;
        best = std::min(curr, memo[i][k] + memo[k][j] + values[i] * values[k] * values[j]);
      } // for k
    } // for j
  } // for i
  return memo[0][values.size() - 1];
} // min_triangulation()
```

What is the time complexity of this solution? Given a polygon with n vertices, there are a total of n^2 subproblems that we may need to compute, each of which may take up to $\Theta(n)$ time to solve (since we have to do a linear pass to find the minimum sum). The overall time complexity of this problem is therefore worst-case $\Theta(n \times n^2) = \Theta(n^3)$. The auxiliary space used by this problem is $\Theta(n^2)$ for the size of the memo.

※ 23.3.5 *Dynamic Programming on Strings and Sequences*

Dynamic programming questions on strings and sequences tend to have more variation, so there is not always a fast-and-hard rule that can be applied to all types of problems. However, for most of these problems, you will want to iterate over the characters of the given sequence(s) and solve a subproblem based on a condition (e.g., such as when characters match). We will look at a couple examples of string and sequence dynamic programming problems in this section.

Example 23.18 Given two strings `s1` and `s2`, return the length of the longest common subsequence (LCS), or 0 if no common subsequence exists. A subsequence is a sequence derived from an original sequence with certain elements deleted, without changing the relative order of the remaining elements (for example, {A, C, D, F} is a subsequence of {A, B, C, D, E, F, G}).

Example: Given `s1 = "parrot"` and `s2 = "almost"`, you would return 3, since the LCS of `"aot"` has a length of 3.

If we were to brute force this problem, we would have to check all of the subsequences in one string and see if they are subsequences of the other string. Assuming the shorter string has a length of m and the longer string has a length of n, the time complexity of such an approach would be $\Theta(n2^m)$, since there are 2^m subsequences that can be built using the smaller string, and comparing each subsequence with the other string each requires at most $\Theta(n)$ work. As a result, this is not a viable approach if you want to solve this problem efficiently.

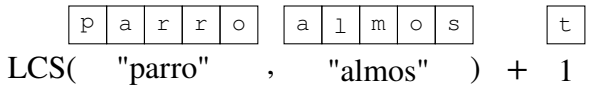
Instead, we can try to improve the solution by finding a recurrence relation that relates the length of a string to smaller subproblems — this allows us to explore better algorithm families such as dynamic programming. It turns out that such a recurrence relation does exist, and it exhibits the property of overlapping subproblems that makes dynamic programming useful. Notice that, when given two strings s_1 and s_2 with respective lengths m and n, there are two possibilities that arise:

1. **The last characters of s_1 and s_2 match.** If this happens, then this last character *must* be the last character of the longest common subsequence (LCS). This is shown using a proof by contradiction: if this last character is not the final character of the LCS, we can add it to make the LCS even longer, as it is shared by both strings after all the other characters.
2. **The last characters of s_1 and s_2 do not match.** If this happens, we know that at least one of these two characters is not in the LCS. We do not know which character is not, but we can recursively solve for it by comparing two values:
   - the LCS between s_1 and the first $n-1$ characters of s_2 (i.e., all but the last character of s_2) — this would tell us the longest LCS we could obtain if the last character of s_2 were not in our final LCS.
   - the LCS between s_2 and the first $m-1$ characters of s_1 (i.e., all but the last character of s_1) — this would tell us the longest LCS we could obtain if the last character of s_1 were not in our final LCS.

   We would then take the larger of these two values to identify which character should belong in our LCS.

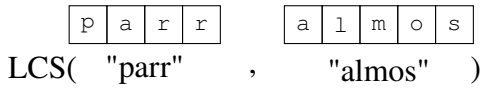
To illustrate this, let's look at the example provided.

Since these two strings share the same final letter, we know for certain that the longest common subsequence ends with the letter 't' (otherwise, we can simply add this letter at the end to extend the length of any common subsequence). Therefore, the longest common subsequence of the entire string must be the longest common subsequence of "parro" or "almos", plus the letter 't'.

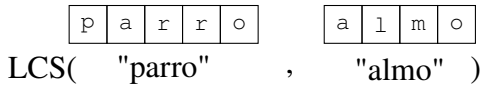


However, what if the last two characters were not the same? For example, suppose we wanted to find the longest common subsequence between "parro" and "almos". Since the last characters do not match, one of these letters must not be in the final longest common subsequence.

Thus, we are left with two possiblities. If the "o" in "parro" is omitted, then the longest common subsequence is simply the longest common subsequence between "parr" and "almos":



On the other hand, if the "s" in "almos" is omitted, then the longest common subsequence is the longest common subsequence between "parro" and "almo":



To identify which is correct, we will have to solve both recurrences and take the better result.

Lastly, note that the longest common subsequence of any empty string is trivially 0, which provides a base case for our problem. Putting everything together, we end up with the following recurrence relation, where M and N represent the lengths of the two strings S_1 and S_2, and $S_1[1 \dots M]$ represents all characters in S_1 from position 1 to position M (using one-indexing):

$$\text{LCS}(S_1[1 \dots M], S_2[1 \dots N]) = \begin{cases} 0, & \text{if } M = 0 \text{ or } N = 0 \\ \text{LCS}(S_1[1 \dots M-1], S_2[1 \dots N-1]) + 1, & \text{if } S_1[M] = S_2[N] \\ \max(\text{LCS}(S_1[1 \dots M-1], S_2[1 \dots N]), \text{LCS}(S_1[1 \dots M], S_2[1 \dots N-1])), & \text{if } S_1[M] \neq S_2[N] \end{cases}$$

Since overlapping subproblems are involved, a dynamic programming approach is useful for solving this problem. There are a total of $M+1$ possibilities for the length of the first string (0, 1, …, M) and $N+1$ possibilities for the length of the second string (0, 1, …, N), so we will create a memo with dimensions $(M+1)\times(N+1)$. If we use a top-down approach to solve this problem, we will make recursive calls using the conditions of the recurrence relation above, retrieving any solutions from the memo if they have been encountered before. The code for a top-down solution is shown below:

```
1  int32_t lcs(const std::string& s1, const std::string& s2) {
2    size_t m = s1.length(), n = s2.length();
3    std::vector<std::vector<int32_t>> memo(m + 1, std::vector<int32_t>(n + 1, -1));
4    return lcs_helper(s1, s2, s1.length(), s2.length(), memo);
5  } // lcs_helper()
6
7  int32_t lcs_helper(const std::string& s1, const std::string& s2, int32_t m, int32_t n,
8                     std::vector<std::vector<int32_t>>& memo) {
9    if (m == 0 || n == 0) {
10     return 0;
11   } // if
12   if (memo[m][n] != -1) {
13     return memo[m][n];
14   } // if
15   // final characters match (minus 1 since strings use 0-indexing)
16   if (s1[m - 1] == s2[n - 1]) {
17     return memo[m][n] = lcs_helper(s1, s2, m - 1, n - 1, memo) + 1;
18   } // if
19   else {
20     return memo[m][n] = std::max(
21       lcs_helper(s1, s2, m - 1, n, memo),
22       lcs_helper(s1, s2, m, n - 1, memo)
23     );
24   } // else
25 } // lcs_helper()
```

If we use a bottom-up approach to solve the problem, we would build up the subproblems starting from the base case. Using the rules of the recurrence relation, if the two characters at $S_1[i]$ and $S_2[j]$ match, then we would set the value of memo$[i][j]$ to memo$[i-1][j-1]+1$ (examples of this case are shown below).

			a	l	m	o	s	t
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0
a	2	0	**1**	1	1	1	1	1
r	3	0	1	1	1	1	1	1
r	4	0	1	1	1	1	1	1
o	5	0	1	1	1	**2**	2	2
t	6	0	1	1	1	2	2	**3**

If $S_1[i] == S_1[j]$, then we would set the value of memo$[i][j]$ to max(memo$[i-1][j]$, memo$[i][j-1]$).

			a	l	m	o	s	t
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0
a	2	0	1	1	1	1	**1**	1
r	3	0	1	1	1	1	1	1
r	4	0	1	1	1	1	1	1
o	5	0	1	1	1	2	2	2
t	6	0	1	1	1	**2**	2	3

The code for a bottom-up approach is shown below:

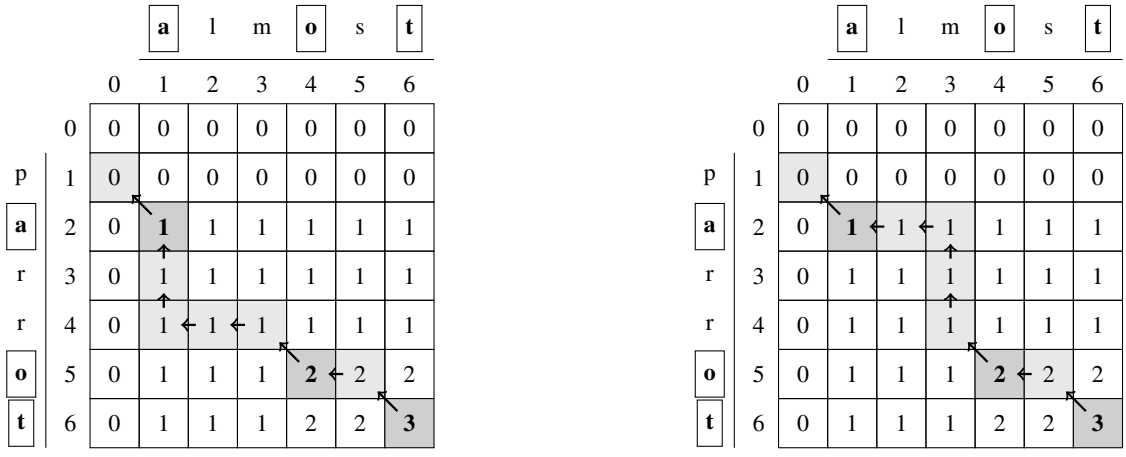
```
int32_t lcs(const std::string& s1, const std::string& s2) {
  size_t m = s1.length(), n = s2.length();
  std::vector<std::vector<int32_t>> memo(m + 1, std::vector<int32_t>(n + 1));
  for (int32_t i = 1; i <= m; ++i) {
    for (int32_t j = 1; j <= n; ++j) {
      if (s1[i - 1] == s2[j - 1]) {
        memo[i][j] = memo[i - 1][j - 1] + 1;
      } // if
      else {
        memo[i][j] = std::max(memo[i - 1][j], memo[i][j - 1]);
      } // else
    } // for j
  } // for i
  return memo[m][n];
} // lcs()
```

Since there are $\Theta(MN)$ subproblems that we may encounter, each of which can be solved in constant time, the overall time complexity of this solution is $\Theta(MN)$. Similarly, a memo of size $\Theta(MN)$ is used, so the auxiliary space used by this solution is also $\Theta(MN)$.

Example 23.19 Solve the longest common subsequence problem, but return the actual LCS instead of its length. If there are multiple LCS, you may return any of them.

It may seem this problem requires us to store more information, but our memo from the previous example is enough to find the actual LCS! We simply have to backtrack from the solution cell (memo[M][N]) back to a base case, using the final character at each position to determine where we came from. If the characters $S_1[i]$ and $S_2[j]$ match, then that character must be in the LCS, and we must have come from memo[$i-1$][$j-1$] when solving the subproblem. On the other hand, if the characters $S_1[i]$ and $S_2[j]$ do not match, then we must have come from the larger of memo[$i-1$][j] and memo[i][$j-1$] (or either, if both memo values are equal). If we use these rules to walk through the memo, we can identify the characters in our LCS by keeping track of the characters that match along the way.

Two backtracking paths are shown below, both of which lead to the answer of `"aot"`:



The code for the backtracking process is shown below:

```
std::string lcs(const std::string& s1, const std::string& s2) {
  std::vector<std::vector<int32_t>> memo(m + 1, std::vector<int32_t>(n + 1));
  // ...solve subproblems and fill out memo...
  int32_t i = m, j = n;
  while (i > 0 && j > 0) {
    if (s1[i - 1] == s2[j - 1]) {  // minus 1 since strings use 0-indexing
      lcs = s1[i - 1] + lcs;
      --i;
      --j;
    } // if
    else if (memo[i][j - 1] > memo[i - 1][j]) {
      --j;
    } // else if
    else {
      --i;
    } // else
  } // while
} // lcs()
```

Since the length of the discovered path has a length of at most $M+N+1$, the time complexity of finding the actual LCS string using the memo is $\Theta(M+N)$. This is asymptotically smaller than the $\Theta(MN)$ time complexity required to solve the original problem, so the overall time complexity of the problem remains $\Theta(MN)$, regardless of whether we want to find the length of the LCS or the actual LCS itself.

Example 23.20 You are given two strings, `s1` and `s2`, with respective lengths of m and n. You are allowed to perform the following three operations on a word:

1. Insert a character.
2. Delete a character.
3. Replace a character.

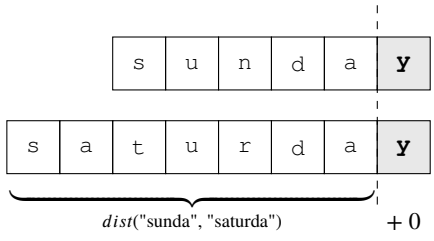
Write a function that returns the minimum number of operations needed to turn `s1` into `s2` (known as the *edit distance*).

Example: Given `s1 = "sunday"` and `s2 = "saturday"`, you would return 3, since a minimum of 3 operations are needed to turn `"sunday"` into `"saturday"`:

- **Operation 1:** replace `'n'` with `'r'` (su**n**day → su**r**day)
- **Operation 2:** insert `'t'` before `'u'` (surday → s**t**urday)
- **Operation 3:** insert `'a'` before `'t'` (sturday → s**a**turday)

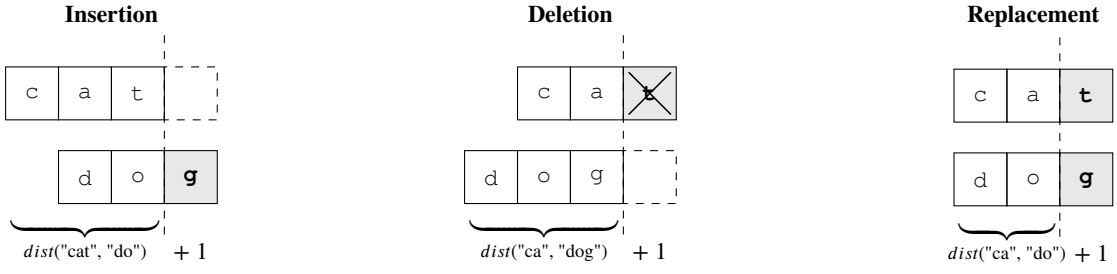
Although this problem may seem new, it actually follows a dynamic programming pattern that we have already discussed. Similar to the longest common subsequence problem, we can compare the last characters of the two strings to devise a recurrence relation that relates the edit distance of larger strings to the edit distances of smaller substrings. Notice that there are only two outcomes that can happen if we compare the last characters of the source and target string: either they match, or they don't.

1. If the last characters match, then no operation is needed on the last character to convert from the source string to the target string. Thus, the edit distance between the two strings is equal to the *edit distance of the two strings up to their second-to-last characters*. For example, suppose we want to convert the string "sunday" to the string "saturday". Since the last two characters of these strings are the same, the edit distance between "sunday" and "saturday" must be the same as the edit distance between "sunda" and "saturda".



2. If the last characters do *not* match, then we have to perform one operation on the last character of the source string to match it with the last character of the target string. To determine which operation to perform (i.e., insert, delete, or replace), we will identify the one that involves the fewest number of edit operations. For example, suppose we are trying to convert the starting string "cat" to the ending string "dog". The last characters between these two strings do not match, so there are three choices we can make:
   - We can perform an *insertion* to match these last two characters. If we perform an insertion, the total edit distance is equal to the edit distance between "cat" and "do" + 1 (to insert the "g" in "dog").
   - We can perform a *deletion* to match these last two characters. If we perform a deletion, the total edit distance is equal to the edit distance between "ca" and "dog" + 1 (to delete the "t" in "cat").
   - We can perform a *replacement* to match these last two characters. If we perform a replacement, the total edit distance is equal to the edit distance between "ca" and "do" + 1 (to replace the "t" in "cat" with the "g" in "dog").

   The edit distance of the original two strings would therefore be the minimum of these three options.



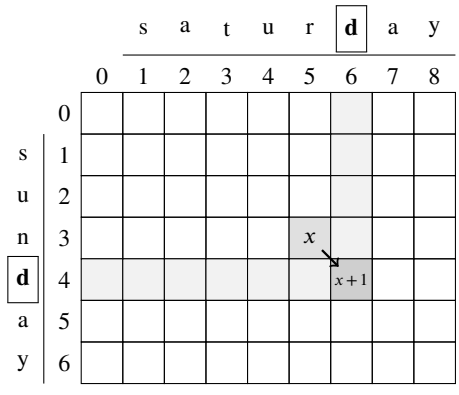
The base cases occur if the length of the starting or ending string is 0. If the starting string has length 0, the minimum edit distance would just be the length of the ending string (since all of its characters will have to be inserted). Similarly, if the ending string has length 0, the minimum edit distance would be the length of the starting string (since all of its characters will have to be deleted). Putting this all together, if we define $D(i, j)$ as the edit distance between $S_1[0...i]$ and $S_2[0...j]$ (where $S_1[0...i]$ represents all characters in S_1 up to index i), we can devise the following recurrence relation for this problem:

$$\mathrm{D}(i,j) = \begin{cases} i, & \text{if } i \neq 0, j = 0 \\ j, & \text{if } i = 0, j \neq 0 \\ D(i-1, j-1), & \text{if } S_1[i] == S_2[j] \\ \min(D(i-1,j), D(i,j-1), D(i-1,j-1)) + 1, & \text{if } S_1[i] \neq S_2[j] \end{cases}$$

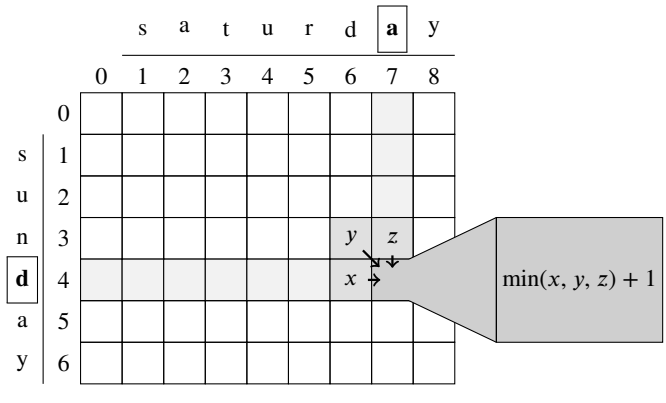
There are overlapping subproblems involved in this recurrence relation, so dynamic programming can be applied. Given strings of lengths m and n, there are a total of $\Theta(mn)$ subproblems that may be encountered (one subproblem for every possible substring pair), so we will declare a memo of size $\Theta(mn)$. If we use a top-down approach, we would make the corresponding recursive calls and store the solutions we encounter in our memo along the way. A top-down solution is shown below:

```
1  int32_t edit_distance(const std::string& s1, const std::string& s2) {
2    std::vector<std::vector<int32_t>> memo(s1.length() + 1, std::vector<int32_t>(s2.length() + 1, -1));
3    return edit_distance_helper(s1, s2, s1.length(), s2.length(), memo);
4  } // edit_distance()
5
6  int32_t edit_distance_helper(const std::string& s1, const std::string& s2, int32_t i, int32_t j,
7                               std::vector<std::vector<int32_t>>& memo) {
8    if (i == 0) {
9      return j;
10   } // if
11   if (j == 0) {
12     return i;
13   } // if
14   if (memo[i][j] != -1) {
15     return memo[i][j];
16   } // if
17   if (s1[i - 1] == s2[j - 1]) {
18     return memo[i][j] = edit_distance_helper(s1, s2, i - 1, j - 1, memo);
19   } // if
20   else {
21     int32_t insertion = edit_distance_helper(s1, s2, i, j - 1, memo);
22     int32_t deletion = edit_distance_helper(s1, s2, i - 1, j, memo);
23     int32_t replacement = edit_distance_helper(s1, s2, i - 1, j - 1, memo);
24     return memo[i][j] = std::min({insertion, deletion, replacement}) + 1;
25   } // else
26 } // edit_distance_helper()
```

If we use a bottom-up approach, we would build up the edit distances of all pairs of substrings, starting from the base cases. This is done by comparing the characters corresponding to each cell of the memo and identifying the relevant subproblems we need the solutions to. If the characters match, we simply copy over the value in the cell on the top left.



If the characters do not match, we add one to the minimum of the cell directly above, the cell directly on the left, and the cell directly on the top left (this picks the best outcome out of insertion, deletion, and replacement).



After filling out the entire memo, the solution would be stored at the cell in the bottom right corner.

The code for a bottom-up solution is shown below:

```
int32_t edit_distance(const std::string& s1, const std::string& s2) {
  std::vector<std::vector<int32_t>> memo(s1.length() + 1, std::vector<int32_t>(s2.length() + 1));
  // base cases
  for (int32_t i = 0; i <= s1.length(); ++i) {
    memo[i][0] = i;
  } // for i
  for (int32_t j = 0; j <= s2.length(); ++j) {
    memo[0][j] = j;
  } // for j
  for (int32_t i = 1; i <= s1.length(); ++i) {
    for (int32_t j = 1; j <= s2.length(); ++j) {
      if (s1[i - 1] == s2[j - 1]) {
        memo[i][j] = memo[i - 1][j - 1];
      } // if
      else {
        memo[i][j] = 1 + std::min({memo[i][j - 1], memo[i - 1][j], memo[i - 1][j - 1]});
      } // else
    } // for j
  } // for i
  return memo[s1.length()][s2.length()];
} // edit_distance()
```

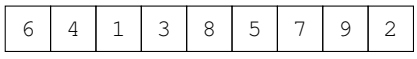
There are a total of $\Theta(mn)$ subproblems, where m and n are the lengths of the two strings, and each subproblem can be solved in constant time. With the help of the memo, the time complexity of the edit distance problem is therefore $\Theta(mn)$. Similarly, this solution initializes a $(m+1)\times(n+1)$ memo, so the auxiliary space used is also $\Theta(mn)$.

Example 23.21 You are given a sequence of numbers S. Write a function that returns the length of the longest increasing subsequence (LIS) of S (i.e., a subsequence of S with all its elements in strictly increasing order).

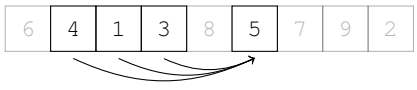
Example: Given the sequence `[6, 4, 1, 3, 8, 5, 7, 9, 2]`, you would return 5, since the LIS is `[1, 3, 5, 7, 9]`.

To solve this problem, we could brute force a solution and identify every subsequence, check if it is increasing, and keep track of the longest subsequence we've encountered. However, this is not optimal. How can we improve the performance of our solution?

The key insight to notice is that we can define the problem recursively by keeping track of the LIS of previous states and using these values to compute the LIS of states we encounter later on. For instance, consider the sequence in the example:



Let's select any arbitrary value from this sequence, such as 5. For 5 to be in the longest increasing subsequence, it must directly follow either 4, 1, or 3 (otherwise, the sequence wouldn't be strictly increasing):



Therefore, if we know the longest increasing subsequences ending at either 4, 1, or 3, we can simply add one to the longest of these three values to get the longest increasing subsequence ending at 5. That is, if the longest increasing subsequence ending at 4 has length x, the longest increasing subsequence ending at 1 has length y, and the longest increasing subsequence ending at 3 has length z, the longest increasing subsequence ending at 5 has length $\max(x, y, z) + 1$.

In general, given any index i of the sequence, the longest increasing subsequence ending at index i can be computed by finding the longest increasing subsequence ending at any index j such that $j < i$ and $S[j] < S[i]$, and adding 1 to that value:

$$\text{LIS}(i) = 1 + \max(\text{LIS}(j) \text{ such that } j < i \text{ and } S[j] < S[i])$$

One base case occurs when index i is 0, since the length of the LIS ending at index i must be 1 (the first element itself). Another base case occurs when the value at index i is smaller than all the elements before it; this also implies that the LIS ending with the value at index i is 1, as no value before it can be added to form an increasing subsequence. Putting this all together, we get the following recurrence relation:

$$\text{LIS}(i) = \begin{cases} 1, & \text{if } i = 0 \text{ or } S[j] \geq S[i] \text{ for all } 0 \leq j < i \\ 1 + \max(\text{LIS}(j) \text{ such that } j < i \text{ and } S[j] < S[i]), & \text{otherwise} \end{cases}$$

Since each subproblem requires us to compute the LIS of earlier values in the array, the same subproblem may be needed more than once. As a result, there exist overlapping subproblems, and dynamic programming can be used. Here, each subproblem can be distinguished using an index value from 0 to $n-1$, where n is the number of elements in the original array, so our memo will also be one-dimensional with size n.

If we use a top-down approach, we would make the recursive calls indicated by the recurrence relation, storing any solutions we encounter in our memo. A top-down solution to this problem is shown below:

```
int32_t lis(const std::vector<int32_t>& nums) {
  std::vector<int32_t> memo(nums.size(), -1);
  for (int32_t i = nums.size() - 1; i >= 0; --i) {
    lis_helper(nums, i, memo);
  } // for i
  return *std::max_element(memo.begin(), memo.end());
} // lis()

int32_t lis_helper(const std::vector<int32_t>& nums, int32_t idx, std::vector<int32_t>& memo) {
  if (idx == 0) {
    return 1;
  } // if
  if (memo[idx] != -1) {
    return memo[idx];
  } // if
  int32_t best = 1;
  for (int32_t i = idx - 1; i >= 0; --i) {
    if (nums[i] < nums[idx]) {
      best = std::max(best, 1 + lis_helper(nums, i, memo));
    } // if
  } // for i
  return memo[idx] = best;
} // lis_helper()
```

If we use a bottom-up approach, we would build up the subproblems starting from the base case. To determine what goes at memo[i], we would iterate over the sequence up to index i and find the largest value of memo[j] such that $S[j] < S[i]$, and then add one to this value. A bottom-up solution is shown below:

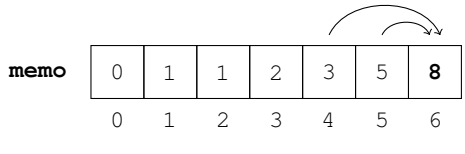
```
int32_t lis(const std::vector<int32_t>& nums) {
  std::vector<int32_t> memo(nums.size(), 1);
  for (int32_t i = 1; i < nums.size(); ++i) {
    for (int32_t j = 0; j < i; ++j) {
      if (nums[j] < nums[i]) {
        if (memo[j] + 1 > memo[i]) {
          memo[i] = memo[j] + 1;
        } // if
      }  // if
    }  // for j
  }  // for i
  return *std::max_element(memo.begin(), memo.end());
}  // lis()
```

For any index i, we have to iterate over all i previous values in the `nums` array to determine if it contributes to the subsequence (and potentially needing to solve a subproblem at each index). With the help of the memo, this can completed in $\Theta(n^2)$ time, where n is the size of the `nums` array. The auxiliary space used by this algorithm is $\Theta(n)$, for the size of the memo.

23.4 Memo Space Optimization for Bottom-Up Dynamic Programming

When using bottom-up dynamic programming, there are certain situations where you can optimize memory usage by reusing previous memo cells for multiple subproblems. This optimization technique can be used if you know that a subproblem will never be needed again when building up your solution.

For instance, consider the Fibonacci problem introduced at the beginning of this chapter. To find the i^{th} Fibonacci number, you only need to query the $(i-1)^{\text{th}}$ and $(i-2)^{\text{th}}$ Fibonacci numbers.

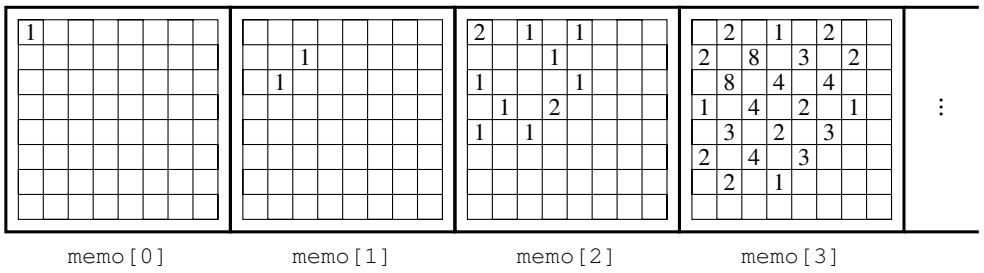


In the example above, you only need to know `memo[4]` and `memo[5]` to compute `memo[6]`. The remaining elements in the memo are not needed at all — in fact, you will never need them again when you solve for larger subproblems! Thus, there is no point in storing `memo[0]` up to `memo[3]`, since they will never be referenced later.

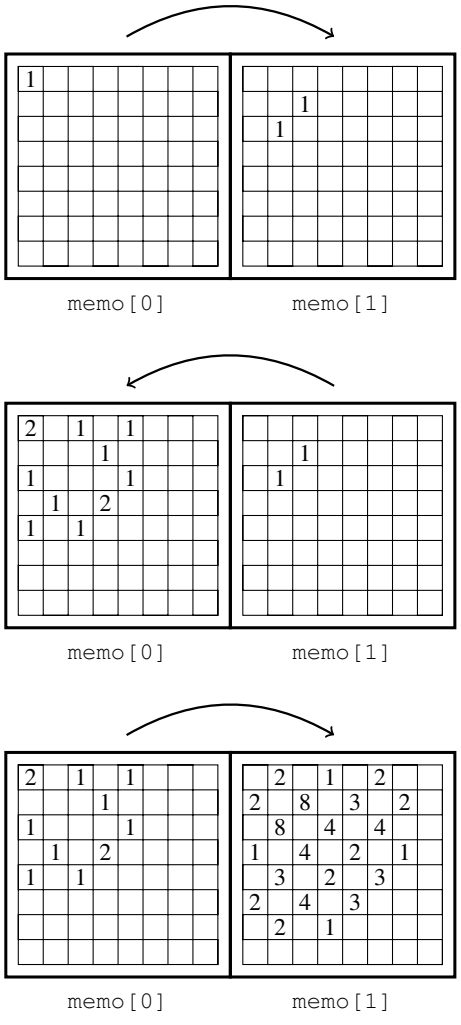
For problems like these, you can save on memory by recycling the memo space used for subproblems that will never be needed later. This often involves collapsing a dimension of the memo (for example, $\Theta(n) \to \Theta(1)$ or $\Theta(mn) \to \Theta(n)$) based on how far back you need to query to solve a subproblem. In the example above, we only need to keep track of two values at any point in time, so we can reduce the size of our memo from $\Theta(n)$ to $\Theta(1)$. A solution using this strategy is shown below:

```
1  uint64_t fib(int32_t n) {
2    if (n < 2) {
3      return n;
4    } // if n
5    uint64_t prev2 = 0, prev1 = 1, curr = 0;
6    for (int32_t i = 2; i <= n; ++i) {
7      curr = prev2 + prev1;
8      prev2 = prev1;
9      prev1 = curr;
10   } // for i
11   return curr;
12 } // fib()
```

This strategy can be used to optimize multidimensional dynamic programming problems as well. Consider the knight moves problem covered earlier. Initially, we used a memo of size $\Theta(mn^2)$ to store all the ways to reach each of the n^2 cells on the chessboard using up to m moves.



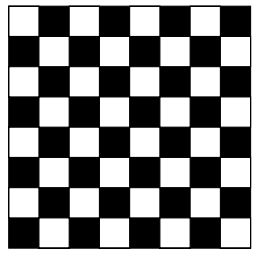
However, some of this information will never be used again once the subproblems become large enough! For any i, the solutions for i moves only depend on the solutions for $i-1$ moves, so there is no reason to store the subproblems for $i-2$ and fewer moves if you've already built up the solution to the i^{th} move. This insight allows us to collapse a dimension of our memo, as we only need to store two boards in our memo at any single point in time. We can implement this by using one board for odd-numbered moves and another board for even-numbered moves, alternating between the two as we build up the solution.



The code for this optimized solution is shown below: note that the solution is nearly identical to the bottom-up solution introduced earlier, but with a memo of outer dimension 2 instead of $m+1$, and with modulo 2 applied when indexing a move (on lines 8, 20, and 29) to determine whether a subproblem should be stored in the even or odd board of our memo.

```
1  int32_t knight_moves(int32_t m, int32_t n) {
2    std::vector<std::vector<std::vector<int32_t>>> memo(2,
3      std::vector<std::vector<int32_t>>(n, std::vector<int32_t>(n)));
4    memo[0][0][0] = 1;
5    for (int32_t num_move = 1; num_move <= m; ++num_move) {
6      for (int32_t row = 0; row < n; ++row) {
7        for (int32_t col = 0; col < n; ++col) {
8          memo[num_move % 2][row][col] =
9            memo_val_bounds_check(num_move - 1, row - 2, col - 1, memo) +
10           memo_val_bounds_check(num_move - 1, row - 2, col + 1, memo) +
11           memo_val_bounds_check(num_move - 1, row + 2, col - 1, memo) +
12           memo_val_bounds_check(num_move - 1, row + 2, col + 1, memo) +
13           memo_val_bounds_check(num_move - 1, row - 1, col - 2, memo) +
14           memo_val_bounds_check(num_move - 1, row - 1, col + 2, memo) +
15           memo_val_bounds_check(num_move - 1, row + 1, col - 2, memo) +
16           memo_val_bounds_check(num_move - 1, row + 1, col + 2, memo);
17       } // for col
18     } // for row
19   } // for num_move
20   return memo[m % 2][n - 1][n - 1];
21 } // knight_moves()
22
23 int32_t memo_val_bounds_check(int32_t move, int32_t row, int32_t col,
24                               std::vector<std::vector<std::vector<int32_t>>>& memo) {
25   if (row < 0 || col < 0 || row >= memo[0].size() || col >= memo[0].size()) {
26     return 0;
27   } // if
28   else {
29     return memo[move % 2][row][col];
30   } // else
31 } // memo_val_bounds()
```

This optimization brings the auxiliary space usage from $\Theta(mn^2)$ down to $\Theta(n^2)$. From a complexity perspective, this is the best we can do — however, it is actually possible to further reduce the number of boards we need to store from two down to one! A single board is sufficient because odd and even-numbered moves use different squares of the table, and the subproblem positions of the i^{th} move will never conflict with the subproblem positions of the $(i-1)^{\text{th}}$ move. In the figure below, the white squares are only used on even-numbered moves (0, 2, 4, …), while the black squares are only used on odd-numbered moves (1, 3, 5, …).



Since odd and even subproblems do not overlap, there is a way to judiciously build up the subproblems using a single board, by computing the black squares in terms of white and the white squares in terms of black (zeroing out the old value at each cell before solving each subproblem). The auxiliary space remains $\Theta(n^2)$ since our memory is only reduced by a constant factor.[10]

It is important to note that this strategy of recycling memo positions only works for bottom-up dynamic programming, and *not* top-down. This is because the bottom-up approach gives you control over the order in which the subproblems are encountered as you build up your solution, making it easy to identify which subproblems will never be needed again (and whose memo positions can therefore be recycled). On the other hand, top-down dynamic programming relies on recursion to solve the subproblems, and there is no practical way to dictate the order in which the subproblems are encountered throughout the recursive process. Since there is no way of predetermining which subproblems will never be needed again during future recursive calls, we cannot reuse memo positions in a similar fashion if using a top-down approach.

[10] Although the single-board solution technically uses less memory, it might not be worth it to go this far, since the amount of memory you are saving over the two-board solution is often negligible compared to the time it would take to write such a solution, especially since the space complexity would still be the same. This example is provided to show that it is *possible* to solve the problem using a single board, but actually micro-optimizing up to this point is not necessary.

23.5 Summary of Dynamic Programming Patterns

A summary of the previous dynamic programming patterns is provided in the table below. Note that these are merely common strategies that can be used to solve dynamic programming problems, and that not all problems fall perfectly into any one of these categories. To fully master dynamic programming, the best method is to practice and expose yourself to as many types of dynamic programming problems as possible.

Type	Definition	Approach
Count Ways	Count the number of ways to achieve a goal	Sum all possible ways to reach the current state
Path Optimization	Find the best path to a goal, given a cost for each step	Choose the best (min or max) cost of all possible states from which the current state is reachable, and add the cost of the current state
Decision Making	Choose a subset of items to include given constraints	Decide whether to include the current item by looking at previous states where that item was and was not used, and choosing the better option
Interval Merging	Combine all elements with the optimal cost	Identify the cost of merging each possible subinterval and use the optimal merging strategy at each step to combine the entire input
String/List	Typically involves modifying, adding, or removing characters from strings or lists	Depends on the type of problem, but try looking at each position of the string/list and identify a recurrence relationship from the subproblems that involves previous items in the string/list, and then use this relationship to make a choice that leads you toward the solution