Recall the recursive algorithm

\[ \text{fib}(n) \]

If \( n = 1 \) or \( 2 \)
    Return 1

Return \( \text{fib}(n-1) + \text{fib}(n-2) \)

The runtime of this algorithm is exponential and thus terrible. There is, fortunately, a closed form expression for the Fibonacci numbers which is easy to compute, but what if that were not the case? We can still improve the algorithm by caching previous results. This top-down approach is called memoization (creating memos). This is distinct from memorization in that whoever came up with the term screwed up.

Anyway, here's the memoized algorithm.

Let \( M: \mathbb{N} \rightarrow \mathbb{N} \) be initialized to \( \bot \) everywhere. \( M(1) = M(2) = 1 \)

\[ \text{fib}(n) \]

If \( M(n) \neq \bot \)
    Return \( M(n) \)

Let \( a = \text{fib}(n-1) \)
Let \( b = \text{fib}(n-2) \) ← the assignment of \( a \) will make this trivial, but for clarity we include the recursive call

Let \( M(n) = a+b \)

Return \( a+b \)

Here we fill in our map \( M \) via our recursive calls. Once \( M(n) \) is set, we never need to calculate it again. The runtime of our memoized algorithm is thus \( O(n) \) now instead of \( O(2^n) \). Also, do note that our input size is \( \log n = m \), so our proper runtime is \( O(2^m) \) as opposed to \( O(2^{2m}) \).
Now what if we don't want to keep M around after execution? It is kind of a memory hog. Even if we only fill in the entries we need, it's potentially an awful lot of them.

We can instead build solutions bottom-up using dynamic programming. Here we build larger solutions from sub-solutions.

Consider the memoized example for computing \( \binom{n}{k} \).

Let \( M: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) be initialized to 1 everywhere.

\[
\text{Combination}(n, k) \quad \text{// } k \leq n
\]

If \( k = 0 \) or \( k = n \)
Return 1

If \( M(n, k) \neq 1 \)
Return \( M(n, k) \)

Let \( M(n, k) = \text{Combination}(n-1, k) + \text{Combination}(n-1, k-1) \)
Return \( M(n, k) \)

A call to \( \text{Combination}(5, 3) \) fills in the table as per the right. The other entries remain undefined. We generate entries starting with large inputs in a top-down approach (drawn upside-down). Now consider a bottom-up approach where we start with small solutions.

\[
\text{Combination}(n, k)
\]

Let \( M: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) be a map
Define \( M(n, k) = 0 \) if \( k < 0 \) or \( k > n \)
Let \( M(0, 0) = 1 \).

For \( i = 0 \) to \( n \)
   For \( j = 0 \) to \( i \)
      \[
      M(i, j) = M(i-1, j) + M(i-1, j-1)
      \]
Return \( M(n, k) \)
This approach uses more memory (temporarily) and time. However, it is entirely self-contained. If you don’t expect to require many calls to it (or if the nature of the problem changes from call to call as we’ll see below), then avoiding the recursive overhead is typically worthwhile.

An example algorithm where we took this approach is Dijkstra’s, Bellman-Ford, and Floyd-Warshall.

With Dijkstra’s, we extended the next shortest path to find longer shortest paths to more distant vertices.

With Bellman-Ford, we increased the number of permitted intermediate edges. The explicit formula for this is

\[ D_{i+1}(v) = \min \left( D_i(v), \min_{(u,v) \in E} (D_i(u) + w(u,v)) \right) \]

We simply iterated over \( i = 0 \) to \( |V|-1 \) and \( v \in V \) to build up our optimal solutions.

We did something similar with Floyd-Warshall. Here, we looped over \( i = 1 \) to \( |V| \), \( j = 1 \) to \( |V| \), and \( k = 1 \) to \( |V| \) to fill in a 3D table according to the following formula.

\[ D_{i+1}(i,k) = \min \left( D_i(i,k), D_{i-1}(i,j) + D_{i-1}(j,k) \right) \]

Anytime we can write down a recursive formula like this, we can take a similar approach to solving the problem presented to us. Note that with these problems, memoization makes little sense as the graph \( G \) and weighting \( w \) will change frequently. When they don’t, however, we would typically precompute all solutions and then just reference a table of values.
This precomputation approach is technically not memoization since we just compute all possible results ahead of time.

Now this bottom-up approach has a name: dynamic programming (DP). Dijkstra’s is a special case. When we know precisely what choice to make to get more solutions, we call this a greedy algorithm. Here’s an explicit example of the difference.

Make Change (n) // We have coins 25c, 10c, 5c, 1c

Let C be an empty collection of coins

While n ≥ 25
    Add a quarter to C
    n -= 25
While n ≥ 10
    Add a dime to C
    n -= 10
While n ≥ 5
    Add a nickel to C
    n -= 5
While n ≥ 1
    Add a penny to C
    n -= 1
Return C

Here we know explicitly how to construct the solution and simply do so. That makes this algorithm greedy. In other words, there is an optimal substructure we take advantage of. With Dijkstra’s, the optimal substructure is that if w lies on the shortest path p from u to v, then the path q from u → w and path r from w → v such that p = q r results in q being the shortest path from u → w and likewise for r being the shortest path from w → v.
the change-making problem when we have an arbitrary collection
of coins requires dynamic programming.

Make change \((n, K)\) // make change for \(n \in \mathbb{N}\) using coins \(K \subseteq \mathbb{N}\).

Let \(R : \mathbb{N} \to \mathbb{N}\) be a map of solutions \((R(n) = \infty \text{ if } n < 0)\)

\(R(0) = 0\)

(Note that we are only tracking the

number of coins required for

the sake of clarity)

For \(i = 1\) to \(n\)

\(R(i) = 1 + \min_{k \in K} (R(i-k))\) // \(1\) is guaranteed to be in \(K\).

Return \(R(n)\)

Ex) Consider if we have \(K = \{5, 10, 20, 25\}\).

The greedy algorithm would return 3 coins, 25, 10, and 5

for \(n = 40\). The optimal solution, however, returns 20 and 20.

We can explicitly identify the difference between the greedy and

dp algorithms here via their recursive definition.

Greedy: \[ R(n) = \begin{cases} 
1 + R(n-25) & 25 \leq n \\
1 + R(n-10) & 10 \leq n < 25 \\
1 + R(n-5) & 5 \leq n < 10 \\
n & 0 \leq n < 5 
\end{cases} \]

DP: \[ R(n) = 1 + \min_{k \in K} (R(n-k)) \]

The greedy definition knows precisely how to handle \(n\).

The dp definition requires us to deduce how to handle \(n\).
In general, all DP algorithms (greedy algorithms included) look the same. We give this algorithm below for a DP problem below.

\[ \text{DP}(N, R) \]

Let \( M \) be a map from input space to solution space.

Iterate over input space until we reach \( N \)

\[ M(\text{next input}) = R(\text{next input}) \]

Return \( R(N) \)

The manner in which we iterate over input space can be simple as in Floyd-Warshall or it can be complex as is the case with Dijkstra’s.

Let’s now look at a slew of useful greedy and DP problems.