Dynamic Programming

Instructor: Bo Waggoner *Lecture 4*

Dynamic programming is a general technique where we define the solution to a problem in terms of smaller subproblems. Then we go through the subproblems in dependency order and solve them, using saved results to solve the next ones.

Dynamic programming algorithms follow a common formula. We will outline it after seeing an example.

Objectives:

- Know the elements of a dynamic programming (DP) algorithm.
- Be able to solve dynamic programming problems by defining each of the elements.
- Become familiar with a variety of DP problems and ways that the solutions can vary.

1 Longest increasing subsequence

This is the first problem we will solve with dynamic programming. Before defining it, some terminology: Given a list of numbers, also called a sequence, a **subsequence** is any result of deleting some elements of the list. For example, given the list $(1, 2, 3, 4, 5)$, the list $(1, 3, 5)$ is a subsequence but $(1, 5, 2)$ is not. A list of numbers is **(weakly) increasing** if each number is at least as large as the previous one.

The **longest increasing subsequence (LIS)** problem is:

- Input: a nonempty list of integers.
- Output: the length of its longest subsequence that is weakly increasing.

For the previous example, the longest increasing subsequence is the entire thing, $(1, 2, 3, 4, 5)$. If the input is $(1, 6, 5, 3, 4, 8)$, then the longest increasing subsequence is $(1, 3, 4, 8)$ with a length of 4 elements.

The idea behind the algorithm is to consider all the prefixes of the input list. First, we solve a variant of the LIS problem for the prefix of length one; this is easy. Then we use this to solve it for the prefix of length two. And so on up to the end.

Consider the above example, input $(1, 5, 6, 3, 4, 8)$. Suppose we're trying to solve the LIS problem for the prefix $(1, 5, 6, 3, 4)$, with the requirement that we must include the final element, 4. Well, an increasing subsequence that includes 4 can only be one of the following options:

- No prior element – the subsequence starts with 4, so its length is just one.
- 1 as the prior element. We would take the LIS ending with 1, and append 4, making it one longer.
- 3 as the prior element. We would take the LIS ending with 3, and append 4, again making it one longer.

The solution for the prefix ending at 4 is whichever of these options is the longest, which is of course the last one, giving a LIS of $(1, 3, 4)$.

The inner `for` loop of Algorithm 1 implements the logic discussed above: it sets $L[j]$ to be 1 if there are no prior elements weakly smaller, and otherwise 1 plus the best of the previous eligible subsequences.

Algorithm 1 Longest Increasing Subsequence

```
1: Input: List A of integers, length n.
2: Create array L and set L[1] = 1.
3: for j = 2, ..., n do
4:    Set L[j] = 1.
5:    for i = 1, ..., j − 1 do
6:       if A[i] ≤ A[j] and L[j] < L[i] + 1 then
7:          Set L[j] = L[i] + 1.
8:       end if
9:    end for
10: end for
11: Return max_j L[j]
```

Correctness. We prove by induction the following statement: $L[j]$ is the length of the LIS of the input up to index j that includes the element at index j. This will prove correctness because we return $\max_j L[j]$, i.e. the longest increasing subsequence that ends at any location.

Base case: $L[1] = 1$ is correct, because we can take the first element to be a subsequence of itself, with length one.

Inductive case: Suppose $L[1], \ldots, L[j-1]$ are all correct. Now at index j, one possible subsequence is just the element $A[j]$ itself, which has length 1. The other kind of possibility is a subsequence that starts earlier and ends at j, with the prior index included being i. This can only occur if $A[i] \leq A[j]$, and if so, its maximum length is $L[i] + 1$ because we appended the element at index j. The algorithm takes the maximum over these possibilities, so $L[j]$ is correct.

Now if each $L[j]$ is correct, then the algorithm is correct because it returns the maximum of $L[j]$ for all j, one of which must be the LIS of the input.

Efficiency. We claim the running time is $O(n^2)$ and space use is $O(n)$.

Initialization runs in constant time and returning the answer runs in $O(n)$, finding the maximum element of L.

The outer `for` loop runs $n-1$ times, and each iteration, the inner `for` loop runs at most $n-1$ times, with constant-time operations. So the running time is $O(n^2)$.

(This analysis might seem loose, but it's not: considering only the iterations $j = \frac{n}{2}, \ldots, n$ and $i = \frac{j}{2}, \ldots, j-1$, we see that there are at least $\frac{n}{2}$ outer loops for which there are at least $\frac{n}{4}$ inner loops, give or take 1, which leads to at least $\frac{n^2}{8}$ iterations of line 6.)

The space usage is dominated by the array L, which is $O(n)$.

1.1 Reconstructing the subsequence itself.

Algorithm 1 returns the length of the LIS, but not the subsequence itself. Luckily, we can modify it quite easily to do this as well. The approach is similar to the modification of breadth-first-search and Dijkstra's algorithm to return the shortest path itself (not just its length). The result is Algorithm 2.

Correctness. We have already argued above that the length returned by the algorithm is correct (and our new modifications do not affect this). So j^* is correct, i.e. there is a LIS that ends at index j^*.

To be careful and rigorous, we prove by induction the following: `prev`$[j]$ is equal to the index of the element prior to $L[j]$ in some LIS of the prefix of length j that includes the final element.

If the length of the LIS is 1, i.e. contains only the final element, then $L[j^*] = 1$ and lines 7 and 8 did not execute for $j = j^*$, so `prev`$[j^*]$ = NULL, which is correct.

If the length is greater than 1, then $L[j] = L[i] + 1$ where i is the index of a previous element in the LIS. This must have been set in line 7, where line 8 set `prev`$[j] = i$.

All of this implies that Reconstruct-LIS is correct, as it starts with j^* which is the final element in the (global) LIS, and builds the list back-to-front using the `prev` pointers.

Algorithm 2 LIS-2

```
1: Input: List A of integers, length n
2: Create array L and set L[1] = 1.
3: Create array prev and set prev[j] = NULL for all j.
4: for j = 2, ..., n do
5:    Set L[j] = 1.
6:    for i = 1, ..., j − 1 do
7:       if A[i] ≤ A[j] and L[j] < L[i] + 1 then
8:          Set L[j] = L[i] + 1.
9:          Set prev[j] = i.
10:      end if
11:   end for
12: end for
13: Let j* = arg max_j L[j].
14: Return L[j*] along with Reconstruct-LIS(j*).
```

Algorithm 3 Reconstruct-LIS(j^*)

```
1: Create empty list S.
2: Add A[j*] to S.
3: Let j = j*.
4: while prev[j] is not NULL do
5:    Let j =prev[j].
6:    Add A[j] to the beginning of S.
7: end while
8: Return S.
```

Running time. Initializing `prev` is a linear-time $O(n)$ operation, while updating it in line 9 is constant-time. Finally, Reconstruct-LIS runs in time $O(n)$ as it consists of constant-time operations plus a loop containing constant-time operations, and the loop moves backward at least one index each iteration, so has length $O(n)$. So the running time of the overall algorithm has not increased in big-O complexity and is still $O(n^2)$. Similarly, `prev` dominates the added space and is linear, so space usage remains $O(n)$.

Exercise 1. In Algorithm 1, why would it be wrong to return $L[n]$, the last entry of our answer array? Give an example input where returning $L[n]$ would be incorrect.

Exercise 2. Simulate the algorithm on input $(5, 1, 3, 2, 4, 0)$. (To simulate a DP algorithm, it usually suffices to write down spaces for the main array or table, in this case L, and gradually fill it in.)

2 Elements of dynamic programming

We will soon see several more examples of dynamic programming algorithms, but first, let's discuss their common structure.

Every dynamic programming algorithm *must* have the following components:

- **Subproblem definition.** For example with LIS, subproblem j was "compute the length of the LIS of the prefix of the input up to j, requiring it to include the final element." We stored the solutions in an array L.
- **Computing the final answer** from the subproblem answers. For LIS, we took the maximum solution to any subproblem, i.e. $\max_j L[j]$.
- **Recurrence.** The *recurrence* states how to solve any given subproblem. It always has two parts:

- **Base case / initialization.** For LIS, we initialized $L[1] = 1$, because this is the only possible solution for a prefix of length 1.
- **Inductive case:** how to solve a generic subproblem given the solution to "earlier" subproblems. For LIS, we said $L[j]$ was the maximum of 1 and $1 + L[i]$ over any $i < j$ where $A[i] \leq A[j]$.

- **(Optional) reconstructing** the object that witnesses the solution. DP algorithms usually return the *size* or *value* of some object, for example, the length of the LIS. Then, they can usually be modified in a straightforward, formulaic way to construct that actual object itself, for example, the actual subsequence as we did above. This modification usually proceeds by remembering which choices we made when solving a subproblem, e.g. when setting $L[j] = L[i] + 1$, remembering which index i was used.

Every dynamic programming solution (at least in this class) has the above components. The key question you usually need to answer is: What are the subproblems, and what is the recurrence? Usually, the subproblems can be arranged in an array, since they must be solved in order. Often this array is multidimensional, as we will see. Sometimes the subproblem is essentially the same as the original problem, just on a prefix or subset of the input. But often the subproblem is slightly different, as in the LIS example where the subproblem required the subsequence to include the final element.

Once you define the above elements, the DP algorithm has essentially been defined:

1. Create a data structure (usually an array) to store the subproblem solutions.
2. Initialize the base cases of the recurrence.
3. Iterate through the subproblems in dependency order and solve using the inductive case of the recurrence.
4. Compute the final answer using the subproblem answers; return it.

To reconstruct the witnessing object as well, it can generally be modified by creating a data structure that remembers the choices made, at each subproblem, when solving the recurrence; then backtracking through these choices.

Exercise 3. Recall: what are the components of a dynamic programming algorithm?

Proofs of correctness. Every DP algorithm is proven correct with the following inductive proof:

1. (Base case) We prove the algorithm initializes the base cases or initial subproblems correctly.
2. (Inductive case) At each step, assuming the previous subproblems were solved correctly, we prove that the next subproblem is solved correctly.
3. By induction, steps (1) and (2) prove that all subproblems are solved correctly.
4. (Returning the final answer) We prove that, if all the subproblems were solved correctly, we return the final answer correctly.

For example, with the Longest Increasing Subsequence problem, we proved that $L[1] = 1$ always (base case). Then, we proved that for each subproblem $j = 2, \ldots, n$, assuming that $L[1], \ldots, L[j-1]$ were all correct, the algorithm computes $L[j]$ correctly (inductive step). Finallly, we argued that if $L[j]$ is correct for each j, then it is correct to return $\max_j L[j]$.

Because the proof always follows this template, we will not always write out the full details of the proof of the algorithm's correctness. Instead, we only need to write out the proof of recurrence – base case and inductive case – and returning the final answer. We also need to make sure that we solve the subproblems in dependency order.

3 Edit Distance

The *edit distance* between two strings (i.e. lists of characters) is the smallest number of transformations needed to convert one string into the other. The allowable transformations are:

- Substitute one character for any character.
- Insert a new character anywhere.
- Delete any existing character.

Note this is symmetric: If we can get from string X to Y in a series of transformations, we can get back in the same number of steps by reversing them.

Example. Transforming X = SOLVE to Y = ALIVE in three steps. We use 1-indexing, so $X[1]$ = S and so on.

S	O	L	V	E		
	O	L	V	E		*deletion*
	O	L	I	V	E	*insertion*
	A	L	I	V	E	*substitution*

To solve this as a dynamic programming problem, we need to define the components of any DP solution as outlined above.

Subproblem. Let $E[i, j]$ = edit distance between the length-i prefix of X and the length-j prefix of Y.

Computing final solution. Here the subproblem is exactly the original problem, just on prefixes of the input. So if we solve all the subproblems, then in particular the final subproblem, $E[|X|, |Y|]$, is our answer.

Recurrence. The **base cases** are pretty easy: if X is an empty string, then the only solution is to insert all the characters of Y. Similarly, if Y is empty, then the only solution is to delete all characters of X. So this gives:

- $E[0, j] = j$ for all j.
- $E[i, 0] = i$ for all i.

The trickiest part is the **inductive case**: how do we compute a generic entry $E[i, j]$ assuming that we've solved the "previous" subproblems?

Claim 1. *For $1 \le i \le |X|$ and $1 \le j \le |Y|$, we have*

$$E[i,j] = \min \begin{cases} E[i-1,j]+1 \\ E[i,j-1]+1 \\ E[i-1,j-1] + \begin{cases} 1 & \text{if } X[i] \neq Y[j] \\ 0 & \text{if } X[i] = Y[j] \end{cases} \end{cases}.$$

To prove Claim 1, we will need a key fact about optimal edit sequences.

Fact 1. *In an optimal sequence of edits from X to Y, the edits can only include the following kinds:*

- *Deletion of a character in X.*
- *Insertion of a character in Y (which is never subsequently deleted or modified).*

- *Modification of a character in X to one in Y (which is never deleted or modified).*

Proof. If there is an edit not of the above form, we can obtain a strictly shorter edit sequence that also produces Y. First, suppose we insert a character c that does not form part of Y at the end of the edits. If it is eventually deleted, then we can remove both the insertion and deletion step and get a strictly shorter sequence. If it is eventually modified to some c', then remove the insertion and modification steps, and instead just insert c'; this reduces 2 steps to just 1.

Next, suppose we modify a character c in X to some c' not in Y. If it is deleted, then we can just remove the modification step and delete the original character, reducing 2 steps to just 1. If it is later modified again to some c'', then we can remove both modification steps and replace them with a step that directly modifies c to c'', again reducing 2 steps to 1.

Finally, observe that the above cases already cover all possibilities where we delete some character not in X, showing that such an edit cannot be optimal. □

Proof of Claim 1. Consider an optimal edit sequence transforming $X[1:i]$ into $Y[1:j]$. Thanks to Fact 1, there are only three possibilities: $X[i]$ is deleted, $X[i]$ becomes $Y[j]$ (either by staying unchanged if they are equal, or else by a modification), or $Y[j]$ is inserted after $X[i]$. This follows because if $X[i]$ is not deleted, then it always comes after the other characters in X, so none of those can become $Y[j]$, nor can $Y[j]$ be inserted before $X[i]$.

If $X[i]$ is deleted, then the rest of the edits transform $X[1:i-1]$ into $Y[1:j]$, so the total length is $1 + E[i-1, j]$.

If $Y[j]$ is inserted after $X[i]$, then the rest of the edits transform $X[1:i]$ into $Y[1:j-1]$, so the total length is $1 + E[i, j-1]$.

If $X[i]$ becomes $Y[j]$, then the rest of the edits transform $X[1:i-1]$ into $Y[1:j-1]$. If $X[i] = Y[j]$, then the total cost is $E[i-1, j-1]$. If $X[i] \neq Y[j]$, then we need an edit to modify $X[i]$ to $Y[j]$, so the total cost is $1 + E[i-1, j-1]$. □

Here we see that the ordering of subproblems is that to compute $E[i, j]$, we need to have computed $E[i-1, j], E[i, j-1]$, and $E[i-1, j-1]$. We have now specified all components, so we can put them together into a dynamic programming algorithm.

Algorithm 4 EditDist

Input: Strings X, Y
Output: edit distance
Set $E[0, j] = j$ for all j and $E[i, 0] = i$ for all i.
for $i = 1, \ldots, |X|$ **do**
  for $j = 1, \ldots, |Y|$ **do**
    Let $a = E[i-1, j] + 1$
    Let $b = E[i, j-1] + 1$
    Let $c = E[i-1, j-1] + \begin{cases} 1 & \text{if } X[i] \neq Y[j] \\ 0 & \text{if } X[i] = Y[j] \end{cases}$
    Set $E[i, j] = \min\{a, b, c\}$.
  end for
end for
Return $E[|X|, |Y|]$

Correctness. The key point in proving correctness of a dynamic programming problem is the recurrence. We have proven the recurrence correct in Claim 1. Algorithm 4 implements the recurrence and solves all subproblems in order of dependence. For edit distance, the final subproblem $E[|X|, |Y|]$ is exactly the original problem, so returning it is correct.

Efficiency. To measure the efficiency, we need to measure the input size. Let $n = |X|$ and let $m = |Y|$; note the input size is $n + m$.

Initialization requires time $O(n + m)$. The innermost part of the loops requires constant time, and there are $n \cdot m$ loops total. Returning the answer is constant time. So total time is $O(n \cdot m)$.

The space usage is dominated by E, which requires space $O(n \cdot m)$.

Exercise 4. Simulate the edit distance algorithm on a nontrivial example, for instance, $X =$ PEACE and $Y =$ GEESE.

Exercise 5. Explain how to modify the algorithm so that space usage is only $O(\min\{|X|, |Y|\})$. *Hint:* To compute $E[i, j]$, what do we need to store?

3.1 Reconstructing the sequence of edits

Knowing the edit distance is nice, but we might also want to reconstruct the actual sequence of edits that transforms e.g. ALGORITHM to ANARCHISM in as few steps as possible. Recall that this optional reconstruction is the last common feature of dynamic programming. The general approach is to keep track of the decisions made at each step, then backtrack through the decisions that led to the optimal answer. Given X, Y, and the matrix E constructed by the algorithm, we can run the following subroutine to reconstruct the edit sequence. In this case, we don't need to modify the original algorithm to keep track of which decision was made: there are only a constant number of possibilities, and we can just use E to check which occurred.

Subroutine 5 EditDist-Reconstruct(X, Y, E)

```
Create empty list L
Set i = |X|, j = |Y|
while i > 0 or j > 0 do
  if i = 0 then
    Add "insert Y[j] at beginning" to start of L
    Set j = j − 1
  else if j = 0 or E[i, j] = 1 + E[i − 1, j] then
    Add "delete X[i]" to start of L
    Set i = i − 1
  else if E[i, j] = 1 + E[i, j − 1] then
    Add "insert Y[j] after X[i]" to start of L
    Set j = j − 1
  else if X[i] = Y[j] then
    Set i = i − 1 and j = j − 1                          // no operation needed
  else
    Add "change X[i] to Y[j]" to start of L
    Set i = i − 1 and j = j − 1
  end if
end while
Return L
```

(An actual implementation would want to be a bit more precise about what the entries of L mean. But this gives the idea.)

Correctness follows almost immediately from the correctness of the recurrence. For example, if $E[i, j] = E[i - 1, j] + 1$, then an optimal edit sequence is to transform $X[1 : i - 1]$ into $Y[1 : j]$, then delete $X[i]$, which is exactly what is returned by the algorithm.

For efficiency, we observe that the reconstruction routine runs in $O(|X| + |Y|)$ time, because at least one of i or j is decreased each loop. And only $O(1)$ additional space is required, although we do require all of E to be retained.

Exercise 6. Recall: What are the elements of a dynamic programming solution? (Hint: there are four elements, one has two parts, one is optional.)

4 Memoization

A counterpart to dynamic programming is the technique of *memo-ization*. This utilizes roughly the same *elements* of DP discussed above, but in a different order.

A **recursive memoization** algorithm works as follows (let's ignore the reconstruction step, which would work similarly):

1. Create a data structure, like an array or hash table, to hold answers to the subproblems.
2. Make a call to a recursive procedure ComputeSubproblem() for the subproblem(s) needed for the final solution.
3. Compute the final solution.

where the recursive procedure **ComputeSubproblem** works like:

1. Look up the answer to the given subproblem in the data structure.
2. If it's already there, then return it.
3. Otherwise:
   (a) Compute the answer using the recurrence.
   (b) To do so, if necessary, make recursive calls to ComputeSubproblem() for the inductive steps of the recurrence.
   (c) Save the answer in the data structure.
   (d) Return the answer.

Algorithm 7 gives an example along these lines for Edit Distance, with recursive procedure Subroutine 6.

Subroutine 6 EditDistRecurse

```
Input: i, j
if we have already saved the answer to E[i, j] then
    return E[i, j]
else if i = 0 then
    Save the answer E[i, j] = j
else if j = 0 then
    Save the answer E[i, j] = i
else
    Let a =EditDistRecurse(i − 1, j) + 1
    Let b =EditDistRecurse(i, j − 1) + 1
    Let c =EditDistRecurse(i − 1, j − 1) + 1_{X[i]≠Y[j]}
    Save the answer E[i, j] = min{a, b, c}.
end if
Return E[i, j]
```

Algorithm 7 EditDist2

```
Input: strings X, Y
Return EditDistRecurse(|x|, |y|)
```

Exercise 7. In memoization, imagine we didn't use a data structure to store answers and instead recomputed the answer each time the recursive subroutine is called. I claim that this generally results in exponential time complexity. Why?

Exercise 8. More concretely, can you come up with instances of edit distance with strings of size n where the time complexity is $3^{\Omega(n)}$? (Recall this means there exists $C > 0$ such that for all large enough n, the time complexity exceeds 3^{Cn}.)

5 All-pairs shortest paths

Input: Graph $G = (V, E)$ with n vertices, weighted, directed, no negative cycles. As with Bellman-Ford, this implies that there is a well-defined shortest path length between any two vertices.

Output: two-dimensional array $D[u, v]$ = length of shortest path from u to v.

Subproblem. As in knapsack, we'll introduce an extra variable to break down our subproblems. The key idea is to consider paths that only use a subset of the vertices. So let $d[u, v, k]$ = length of the shortest path from u to v using as intermediate nodes only vertices $1, \ldots, k$.

Final solution. In particular, with n vertices, $d[u, v, n]$ = the length of the shortest path from u to v using all vertices. So if we set $D[u, v] = d[u, v, n]$ for all u, v, this will be correct.

Recurrence. For the base case, consider $k = 0$. Then only paths with no intermediate nodes can be used. So we have $d[u, u, 0] = 0$ for all u, and for all edges $(u, v) \in E$ with length w_{uv}, we have $d[u, v, 0] = w_{uv}$. For all other pairs, we have $d[u, v, 0] = \infty$.

For the inductive case, imagine we've solved $d[u, v, k-1]$ for all u, v and now we want to compute $d[u, v, k]$.

Claim 2. *For $k \geq 1$,*

$$d[u, v, k] = \min \begin{cases} d[u, v, k-1] \\ d[u, k, k-1] + d[k, v, k-1] \end{cases}.$$

Informally, this says we can either use the old route that didn't include k at all, or we can include k. If we do, then we must route from u to k somehow, using distance $d[u, k, k-1]$, and then route to v somehow, using distance $d[k, v, k-1]$.

Proof. The shortest path from u to v, using only intermediate vertices $1, \ldots, k$, either uses vertex k or it doesn't. Suppose it doesn't. Then $d[u, v, k] = d[u, v, k-1]$, by definition.

Suppose it does. Then the shortest path using $1, \ldots, k$ has the form $u, \ldots, k, \ldots, v$. Then the portion $u, \ldots, k$ must be a shortest path from u to k using intermediate vertices $1, \ldots, k-1$. (Otherwise, we could take the shortest path and shorten the distance from u to v, a contradiction.) Similarly, the portion $k, \ldots, v$ must be a shortest path from k to v. So in this case, $d[u, v, k] = d[u, k, k-1] + d[k, v, k-1]$.

Since the shortest path must be one of these two cases, it is the smaller of the two. □

Correctness. As usual with dynamic programming, correctness follows from above arguments that the subproblem, final solution, and recurrence are correct.

Efficiency. Initialization requires up to $O(n^2)$ time, since we set $d[u, v, 0]$ for all pairs of nodes. Similarly, returning the solution requires constructing an $O(n^2)$ array, which has the same running time. There are three nested loops, each with n iterations, and constant-time operations within each. So the running time is dominated by $O(n^3)$.

The space includes D and local variables, but is dominated by d which uses $O(n^3)$ space.

Algorithm 8 Floyd-Warshall

Set $d[u, v, 0] = w_{uv}$ if there is an edge (u, v), or 0 if $u = v$, or else ∞
for $k = 1, \ldots, n$ **do**
  for $u = 1, \ldots, n$ **do**
    for $v = 1, \ldots, n$ **do**
      Set $d[u, v, k] = \min\{d[u, v, k-1]\ ,\ d[u, k, k-1] + d[k, v, k-1]\}$.
    end for
  end for
end for
Set $D[u, v] = d[u, v, n]$ for all u, v
Return D

5.1 Reconstructing the solution.

In this case, the solution is the actual path between u and v that is shortest. As usual, reconstructing it will involve remembering the choices made when solving the subproblems, but here the full procedure is a bit unusual.

A merge approach. The most direct approach, applying our usual DP approach, is as follows. Let us create a variable $\texttt{inter}[u, v]$ standing for "intermediate" vertices between u and v. Whenever we make a modification

$$d[u, v, k] = d[u, k, k-1] + d[k, v, k-1],$$

we set

$$\texttt{inter}[u, v] = k.$$

Initially, we set $\texttt{inter}[u, v] = \perp$, where the "bottom" symbol $\perp$ means "none".

Now, we can reconstruct the path as follows:

- If $\texttt{inter}[u, v] = \perp$, then we must have followed an edge directly from u to v, so the path is just u, v.
- Otherwise, if $\texttt{inter}[u, v] = k$, then we make a recursive call to reconstruct the path from u to k, another to get the path from k to v, and we concatenate these.

A "next" approach. Notice that if a shortest path is of the form $u, x, \ldots, v$, then it is also true that $x, \ldots, v$ is a shortest path from x to v. This implies that we only need to know, for each pair u, v, what the "next" vertex is on a shortest path. If we find that it is x, then we continue by finding the next vertex on the path from x to v, etc.

So initialize $\texttt{next}[u, v] = v$ if there is an edge (u, v) and otherwise $\texttt{next}[u, v] = \perp$. Whenever we make a modification $d[u, v, k] = d[u, k, k-1] + d[k, v, k-1]$, we can set

$$\texttt{next}[u, v] = \texttt{next}[u, k]$$

since the shortest path to v proceeds by first taking the shortest path to k.

In this case, reconstruction is even easier:

1. Add u to the path.
2. Starting at u, let $x =$ $\texttt{next}[u, v]$.
3. Add x to the path.
4. If $x = v$, stop.
5. Otherwise, let $x =$ $\texttt{next}[x, v]$ and go to step 3.

6 Recap and Variants of DP Algorithms

To recap, the elements of a dynamic programming solution are:

- Definition of a subproblem.
- How to compute the final solution from the subproblem solutions.
- Recurrence for solving a subproblem:
  - Base cases.
  - Inductive case: solving a generic subproblem given previous ones.
- (Optional) Reconstructing an object witnessing the solution.

Usually, the subproblem solutions are arranged in an array, where a subproblem depends on ones earlier in the array. Here are several variants to be aware of:

- The subproblems are in a one-dimensional array; or a multidimensional array.
- A subproblem is the same as the original problem, just on a smaller input; or the subproblem is slightly different. Similarly, computing the final solution is just returning the final subproblem solution; or it requires more computation depending on the subproblem.
- The subproblems correspond directly to prefixes of the input; or we create extra index variables to break down the problem in an artificial way. Examples of the latter: single-use knapsack, Floyd-Warshall.