

Lecture Note 7

Dynamic Programming

draft of September 29, 2026, by [N. Young](#)

© 2018–2026 [Neal E. Young](#). Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

7.1	Pascal’s rule for computing n choose k (with long-form proof)	1
7.2	Paths in DAGs	3
7.2.1	Counting Paths in a Grid	3
7.2.1.1	Memoizing	4
7.2.1.2	Iterating over the subproblems	4
7.2.2	Shortest Path in a Grid	5
7.2.3	Shortest Path in a DAG	6
7.2.4	Longest Path in a DAG	7
7.3	Making Change	8
7.3.1	Reducing to Shortest Path in a DAG	8
7.4	Four steps for presenting a dynamic-programming algorithm	10
7.5	Subset Sum	11
7.6	Knapsack (with long-form proof)	13
7.7	Weighted Activity Selection	15
7.8	Longest Common Subsequence (with long-form proof)	18
7.9	Edit Distance and Sequence Alignment	21
7.10	Longest Palindromic Subsequence (with long-form proof)	23
7.11	Paragraph Layout	25
7.12	Minimum-Weight Polygon Triangulation	26
7.13	Optimal Binary Search Tree	27
7.14	Single-Source Shortest Paths with Negative Weights	28
7.15	All Pairs Shortest Paths / Floyd-Warshall Alg. (w/ l. f. proof)	30

7.16	External sources on dynamic programming	33
7.17	Python code	34

7.1 Pascal's rule for computing n choose k (with long-form proof)

Given non-negative integers n and k , the number of size- k subsets of $\{1, 2, \dots, n\}$ is called n choose k , and denoted $\binom{n}{k}$. Consider the problem of computing it:

Computing n choose k

input: non-negative integers n and k

output: $\binom{n}{k}$.

To design an algorithm, first we'll prove Pascal's rule:

Lemma 7.1 (Pascal's rule).
$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } k > 0 \text{ and } n = 0, \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise.} \end{cases}$$

Proof (long form).

1. For $m, \ell \geq 0$, let $\mathcal{C}(m, \ell)$ be the collection of all size- ℓ subsets of $\{1, 2, \dots, m\}$, so $\binom{m}{\ell} = |\mathcal{C}(m, \ell)|$.
2. If $k = 0$, the empty set is the only size- k subset of $\{1, 2, \dots, n\}$, so the rule holds in this case.
3. If $k > 0$ and $n = 0$, there is no size- k subset of $\{1, 2, \dots, n\}$, so the rule holds in this case.
4. So assume $n, k > 0$ and consider $\mathcal{C}(n, k)$.
5. Each set in $\mathcal{C}(n, k)$ either contains n or it doesn't.
6. The sets in $\mathcal{C}(n, k)$ that *don't* contain n are just the sets in $\mathcal{C}(n-1, k)$.
7. By definition of $\mathcal{C}$, there are $\binom{n-1}{k}$ of these.
8. So, the number of sets in $\mathcal{C}(n, k)$ that don't contain n is $\binom{n-1}{k}$.
9. To finish, we show that the number of sets in $\mathcal{C}(n, k)$ that *do* contain n is $\binom{n-1}{k-1}$.
10. For each set S in $\mathcal{C}(n, k)$ that does contain n , let $\phi(S) = S \setminus \{n\}$ be the set with n removed.
11. Then ϕ is a bijection (a one-to-one mapping) between the sets in $\mathcal{C}(n, k)$ that contain n and the sets in $\mathcal{C}(n-1, k-1)$. (That is, $\phi(S)$ is in $\mathcal{C}(n-1, k-1)$, and, for each $S' \in \mathcal{C}(n-1, k-1)$, there is exactly one set $S \in \mathcal{C}(n, k)$ that contains n such that $\phi(S) = S'$ —namely $S = S' \cup \{n\}$.)
12. Because there is such a bijection, the number of sets in $\mathcal{C}(n, k)$ that contain n must equal the number of sets in $\mathcal{C}(n-1, k-1)$. By definition this number is $\binom{n-1}{k-1}$.
13. By this and Step 8, the size of $\mathcal{C}(n, k)$ is $\binom{n-1}{k} + \binom{n-1}{k-1}$. □

The proof of Pascal's rule illustrates the first main principle of dynamic programming. Namely, to reason about a collection of objects (in this case the size- k subsets of $\{1, \dots, n\}$) the proof partitions the collection into a few types—in this case two:

- (A) Those subsets not containing the element n .
- (B) Those subsets containing the element n .

Then it handles each type by noting that, restricted to objects of that particular type, the problem at hand is somehow equivalent to a smaller instance of the original problem.

In the current example, the proof observes that the sets of type (A) are just the size- k subsets of $\{1, \dots, n-1\}$, so counting the type-(A) sets is equivalent to the problem of counting the size- k subsets of $\{1, \dots, n-1\}$. This is a smaller instance of the original problem.

Similarly, the sets of type (B) correspond bijectively (via the deletion of element n) to the size- $(k-1)$ subsets of $\{1, \dots, n-1\}$. So counting the type-(B) sets is equivalent to counting the size- $(k-1)$ subsets of $\{1, \dots, n-1\}$. Again, this is a smaller instance of the original problem.

This understanding of the recursive structure of the objects in question (subsets) yields a recurrence relation (in this case, Pascal's rule). This in turn yields a recursive, divide-and-conquer algorithm for the problem. In this case, for computing $\binom{n}{k}$:

countSubsets(n, k): — $n, k \geq 0$

1. return $\begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } k > 0 \text{ and } n = 0, \\ \text{countSubsets}(n - 1, k) + \text{countSubsets}(n - 1, k - 1) & \text{otherwise.} \end{cases}$

Correctness of the algorithm follows by induction and Lemma 7.1.

This seems like a classic *divide-and-conquer* approach. But beware! This implementation is not efficient—indeed, its run time is exponential! Consider its recursion tree, which has branching factor two, and depth between k and n , so somewhere between 2^k and 2^n nodes!

There are two standard work-arounds for this obstacle. We'll discuss them in the next example.

Exercises

Exercise 7.1. (*example with solution*) The proof of Lemma 7.1 uses a bijection ϕ between those sets in $\mathcal{C}(n, k)$ that contain n and the sets in $\mathcal{C}(n - 1, k - 1)$. Show this bijection for $n = 5$ and $k = 3$. For reference, here is all of $\mathcal{C}(5, 3)$, and its subset $\mathcal{C}(4, 3)$:

$$\mathcal{C}(5, 3) = \left\{ \begin{array}{l} \{1, 2, 3\} \\ \{1, 2, 4\} \\ \{1, 3, 4\} \\ \{2, 3, 4\} \\ \{1, 2, 5\} \\ \{1, 3, 5\} \\ \{1, 4, 5\} \\ \{2, 3, 5\} \\ \{2, 4, 5\} \\ \{3, 4, 5\} \end{array} \right\}$$

$\left. \begin{array}{l} \{1, 2, 3\} \\ \{1, 2, 4\} \\ \{1, 3, 4\} \\ \{2, 3, 4\} \end{array} \right\} \mathcal{C}(4, 3)$

Solution.

sets in $\mathcal{C}(5, 3)$ that contain 5	ϕ $\longleftrightarrow$	sets in $\mathcal{C}(4, 2)$
$\{1, 2, 5\}$		$\{1, 2\}$
$\{1, 3, 5\}$		$\{1, 3\}$
$\{1, 4, 5\}$		$\{1, 4\}$
$\{2, 3, 5\}$		$\{2, 3\}$
$\{2, 4, 5\}$		$\{2, 4\}$
$\{3, 4, 5\}$		$\{3, 4\}$

External sources

- https://en.wikipedia.org/wiki/Pascal%27s_rule

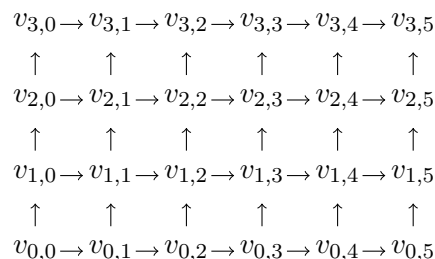
7.2 Paths in DAGs

7.2.1 Counting Paths in a Grid

Here's a toy problem to consider: Counting Paths in a Grid.

input: Non-negative integers i and j .

output: The number of paths from the origin $v_{0,0}$ to the node $v_{i,j}$ at location (i, j) in the 2D grid, as illustrated to the right, assuming each step in the path moves one unit up or one unit to the right.



For example, how many paths are there from $v_{0,0}$ to $v_{3,5}$ in the graph to the right? Let $P(i, j)$ denote the set of paths from $v_{0,0}$ to $v_{i,j}$. We want to count the size of $P(3, 5)$ (denoted $|P(3, 5)|$). The next lemma gives a divide-and-conquer approach:

Lemma 7.2. For $i, j \geq 0$, $|P(i, j)| = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ |P(i, j - 1)| + |P(i - 1, j)| & \text{otherwise} \end{cases}$.

Proof. When $i = 0$ or $j = 0$, the lemma holds because (by inspection of the grid) there is only one path. So assume $i > 0$ and $j > 0$. Partition the set $P(i, j)$ of paths from $v_{0,0}$ to $v_{i,j}$ into two subsets:

- (A) The paths that enter $v_{i,j}$ through $v_{i,j-1}$.
- (B) The paths that enter $v_{i,j}$ through $v_{i-1,j}$.

By inspection of the grid, every path into $v_{i,j}$ enters either through $v_{i,j-1}$ or through $v_{i-1,j}$, so the two subsets (A) and (B) partition $P(i, j)$.

The number of paths entering $v_{i,j}$ through $v_{i-1,j}$ equals the number of paths to $v_{i-1,j}$, because each path to $v_{i-1,j}$, followed by the edge from there to $v_{i,j}$, yields a unique path into $v_{i,j}$. Similarly, the number of paths entering $v_{i,j}$ through $v_{i,j-1}$ equals the number of paths to $v_{i,j-1}$. $\square$

The lemma gives this recursive algorithm for Counting Paths in a grid:

$\text{countPaths}(i, j)$ 1. return $\begin{cases} 1 \\ \text{countPaths}(i, j - 1) + \text{countPaths}(i - 1, j) \end{cases}$	— return the number of paths from $v_{0,0}$ to $v_{i,j}$ if $i = 0$ or $j = 0$ otherwise
---	--

There is one problem with this algorithm: it takes time exponential in i and j . To see why, consider the recursion tree for $\text{countPaths}(n, n)$. In the tree, each leaf has depth at least n , and each non-leaf node has two children, so the number of nodes in each level $i \leq n$ is 2^i . So executing $\text{countPaths}(n, n)$ results in more than 2^n recursive calls altogether!

Two ways to avoid solving subproblems multiple times. In the execution of $\text{countPaths}(i, j)$, exponentially many recursive calls can arise. But each recursive call that arises is of the form $\text{countPaths}(a, b)$ where $0 \leq a \leq i$ and $0 \leq b \leq j$. And there are only $(i + 1) \cdot (j + 1)$ such pairs (a, b) (one for each point lying to the left of and below point (i, j)). So, the number of *different* subproblems solved during the execution is only $\Theta(ij)$.

The reason that the recursive implementation above takes exponential time is that *it solves each distinct subproblem many times*. You can see this by trying it out on the example, or diagramming

the recursion tree. This redundancy is unnecessary. *It's enough to solve each distinct subproblem just once.* There are two standard ways to do this.

7.2.1.1 Memoizing

Memoizing caches the value returned by each call, to avoid computing it again:

countPaths(i, j): — return the number of paths from $(0, 0)$ to (i, j) in the 2d grid

1. maintain global “cache” array $C[]$, initially empty
2. if $C[i, j]$ has not previously been set: (store the value in the cache)
3. set $C[i, j] = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ \text{countPaths}(i, j - 1) + \text{countPaths}(i - 1, j) & \text{otherwise} \end{cases}$
4. return $C[i, j]$

Consider executing **countPaths**(i, j) with this modification. Now, in the recursion tree, there will be just one non-leaf node for each point (a, b) with $0 < a \leq i$ and $0 < b \leq j$, representing the first time **countPaths**(a, b) is called. Subsequent calls with identical parameter values (a, b) will return the cached value, without recursing. So the total number of non-leaf nodes is $\Theta(ij)$. And the number of leaf nodes is at most twice the number of non-leaf nodes (since each non-leaf node has two children). So the total number of nodes is also $\Theta(ij)$. The work associated with each node is constant, so the total time is $\Theta(ij)$, instead of exponential in i and j .

7.2.1.2 Iterating over the subproblems

The second standard fix is to avoid recursion altogether, by simply iterating over the distinct subproblems:

countPaths(i, j): — return the number of paths from $(0, 0)$ to (i, j) in the 2d grid

1. for $a = 0, 1, 2, \dots, i$:
- 2.1. for $b = 0, 1, 2, \dots, j$:
- 2.2. $N[a, b] = \begin{cases} 1 & \text{if } a = 0 \text{ or } b = 0 \\ N[a, b - 1] + N[a - 1, b] & \text{otherwise} \end{cases}$
3. return $N[i, j]$

With this approach, *the iteration order matters!!!* Each $N[a, b]$ must be computed *after* $N[a, b - 1]$ and $N[a - 1, b]$. This requirement is easy to satisfy for this problem, but is more complicated to satisfy for some others.

With either approach, correctness of the resulting algorithm follows by induction on the problem size and Lemma 7.2. This implementation requires (by inspection) $\Theta(ij)$ additions, but the values involved can be roughly as large as 2^{i+j} , so large-integer arithmetic on $(i+j)$ -bit integers is required. Then each addition takes time $O(i + j)$, yielding the following theorem:

Theorem 7.1. *Given integers $i, j \geq 0$, we can compute, in $O((i + j)ij)$ time, the number of paths in the 2d grid graph (defined earlier) from the origin to $v_{i,j}$.*

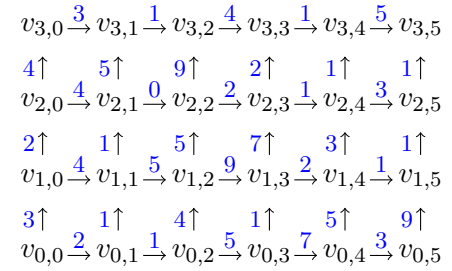
Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section 7.17).

7.2.2 Shortest Path in a Grid

The same reasoning easily extends to solve other problems on paths in a grid, such as finding a shortest path.

input: Non-negative integers i and j , and a weight for each edge in the 2d grid graph.

output: The minimum, over all paths from $v_{0,0}$ to $v_{i,j}$, of the total weight of edges on the path.



To the right is the graph with some edge weights added (in blue).

Recall that $P(i, j)$ denotes the set of paths from $v_{0,0}$ to $v_{i,j}$. Our goal now is to compute the minimum, over all paths in this set, of the total weight of edges in the path.

Define $M(i, j)$ to be this minimum weight. Let $w(v_{a,b}, v_{c,d})$ denote the weight of the edge from $v_{a,b}$ to $v_{c,d}$ (if the edge is present, that is, if $(c, d) = (a + 1, b)$ or $(c, d) = (a, b + 1)$).

$$\textbf{Lemma 7.3.} \text{ For } i, j \geq 0, M(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0, \\ M(i - 1, j) + w(v_{i-1,j}, v_{i,j}) & \text{if } i > 0 \text{ and } j = 0, \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) & \text{if } i = 0 \text{ and } j > 0, \\ \min \left(\begin{array}{l} M(i - 1, j) + w(v_{i-1,j}, v_{i,j}), \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) \end{array} \right) & \text{if } i > 0 \text{ and } j > 0. \end{cases}$$

Proof sketch. When $i = 0$ and $j = 0$, the lemma holds because the only path is the empty path.

Next we give the proof just for the case $i > 0$ and $j > 0$. (The remaining cases are easier.) As before, partition the set $P(i, j)$ of paths from $v_{0,0}$ to $v_{i,j}$ into two types:

- (A) The paths that enter $v_{i,j}$ from $v_{i-1,j}$.
- (B) The paths that enter $v_{i,j}$ from $v_{i,j-1}$.

By inspection of the grid, every path into $v_{i,j}$ enters either through $v_{i,j-1}$ or through $v_{i-1,j}$, so the two subsets (A) and (B) partition $P(i, j)$. It follows that the minimum weight of any path in $P(i, j)$ is the minimum weight of any path of type (A), or the minimum weight of any path of type (B), whichever is smaller.

The paths of type (A) can be obtained by taking any path to $v_{i-1,j}$ and adding the edge from $v_{i-1,j}$ to $v_{i,j}$. So the minimum weight of any path of type (A) is the minimum weight of any path into $v_{i-1,j}$, plus the weight of that edge. That is, the minimum weight of any type-(A) path is $M(i - 1, j) + w(v_{i-1,j}, v_{i,j})$.

By similar reasoning, the minimum weight of any type-(B) path is $M(i, j - 1) + w(v_{i,j-1}, v_{i,j})$.

It follows (when $i > 0$ and $j > 0$), that $M(i, j) = \min \left(\begin{array}{l} M(i - 1, j) + w(v_{i-1,j}, v_{i,j}), \\ M(i, j - 1) + w(v_{i,j-1}, v_{i,j}) \end{array} \right)$. □

Here is the resulting algorithm (the iterative implementation):

```

shortestPath( $i, j, w$ ):                                — return the minimum weight of any path from  $v_{0,0}$  to  $v_{i,j}$ 
1. for  $a = 0, 1, 2, \dots, i$ :
2.1. for  $b = 0, 1, 2, \dots, j$ :
2.2.    $M[a, b] = \begin{cases} 0 & \text{if } a = 0 \text{ and } b = 0, \\ M[a-1, b] + w(v_{a-1,b}, v_{a,b}) & \text{if } a > 0 \text{ and } b = 0, \\ M[a, b-1] + w(v_{a,b-1}, v_{a,b}) & \text{if } a = 0 \text{ and } b > 0, \\ \min \begin{cases} M[a-1, b] + w(v_{a-1,b}, v_{a,b}) \\ M[a, b-1] + w(v_{a,b-1}, v_{a,b}) \end{cases} & \text{if } a > 0 \text{ and } b > 0. \end{cases}$ 
3. return  $M[i, j]$ 

```

Correctness of the algorithm follows from the lemma and induction on the problem size.

By inspection the algorithm runs in time $\Theta(ij)$. This proves the following theorem:

Theorem 7.2. *Given integers $i, j \geq 0$ and edge weights for the 2d grid graph (defined earlier), we can compute, in $O(ij)$ time, the minimum weight of any path from the origin to $v_{i,j}$.*

7.2.3 Shortest Path in a DAG

The previous reasoning for grid graphs extends easily to any directed acyclic graph (DAG). To illustrate this, consider finding shortest paths in an edge-weighted DAG:

input: DAG $G = (V, E)$ with edge weights; source vertex s , destination vertex w

output: The shortest-path distance from s to w

Fix any such input. We start with the following recurrence relation:

Lemma 7.4. *For any vertex $w \in V$:*

$$\text{dist}(s, w) = \begin{cases} 0 & \text{if } w = s \\ \min \{ \text{dist}(s, u) + \text{wt}(u, w) : (u, w) \in E \} & \text{otherwise.} \end{cases}$$

Proof sketch. In the case that $w = s$, the lemma holds because the only path is the empty path.

Consider the case $w \neq s$. The shortest path from s to w is the shortest, over all possible edges (u, w) into w , of the shortest among the paths ending in that edge (u, w) . Among the paths from s to w ending in a given edge (u, w) , a shortest one can be obtained by taking any shortest path from s to u , and appending edge (u, w) . Hence the shortest path from s to w ending in (u, w) has weight $\text{dist}(s, u) + \text{wt}(u, w)$. $\square$

This gives the following recursive algorithm:

```

shortestPath( $G = (V, E), s, w$ ):                        return the shortest-path distance from  $s$  to  $w$  in DAG  $G$ 
1. return  $\begin{cases} 0 & \text{if } w = s \\ \min \{ \text{shortestPath}(G, s, u) + \text{wt}(u, w) : (u, w) \in E \} & \text{otherwise.} \end{cases}$ 

```

If w has no in-neighbors, so the set is empty, the minimum has value ∞ .

Correctness follows from Lemma 7.4 and induction. (Show by induction on the recursion depth that $\text{shortestPath}(G, s, w) = \text{dist}(s, w)$ for all $w \in V$.) We memoize the algorithm. Then the recursion tree will have at most one non-leaf node for each vertex $v \in V$, and the work associated with that node will be proportional to the in-degree of v (the number of edges into v), so the total time will be $\Theta(n + m)$.

If we want instead an iterative algorithm, we have to explicitly compute an order in which to evaluate the subproblems. We can only evaluate a given subproblem $\text{shortestPath}(G, s, w)$ *after* the subproblem for each in-neighbor u of w is done. This is exactly what topological ordering is for! The algorithm first computes a topological ordering of the vertices (in linear time using DFS, as described in [Lecture Note 6](#)), then considers the vertices in that order.

```

shortestPath( $G = (V, E), s, w$ ):           return the shortest-path distance from  $s$  to  $w$  in DAG  $G$ 
1. let  $v_1, v_2, \dots, v_n$  be a topological ordering of the vertices (computed using DFS; see Lecture Note 6)
2. for  $j = 1, 2, \dots, n$ :
3.   set  $d[v_j] = \begin{cases} 0 & \text{if } v_j = s \\ \min \{d[v_i] + \text{wt}(v_i, v_j) : (v_i, v_j) \in E\} & \text{otherwise.} \end{cases}$ 
4. return  $d[w]$ 
    
```

In Step 3 if v_j has no in-neighbors, so the set is empty, the minimum has value ∞ .

Correctness follows from Lemma 7.4 and induction. (Show by induction on j that that $d[v_j] = \text{dist}(s, v_j)$ for all $j \in \{1, 2, \dots, n\}$.) The time to evaluate the right-hand side in Step 3 is $\Theta(1 + \text{in-degree}(v_j))$. Summing over all the vertices, the total time is $\Theta(n + m)$. We conclude with a theorem that summarizes what we’ve figured out:

Theorem 7.3. *There is a linear-time algorithm for Shortest Path in a DAG.*

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section 7.17).

Exercise 7.2. (*implementation*) Implement the algorithm and test your implementation on Hackerrank.com, at <https://www.hackerrank.com/neaileyong-algs101-dag-paths-and-topological-sort> (the “Shortest-path distances in a DAG” challenge; there, edge weights can be negative).

Solution. Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section 7.17).

7.2.4 Longest Path in a DAG

The same reasoning yields a linear-time algorithm for Longest Path in a DAG (computing the *maximum* weight of any path from s to w). The algorithm and its derivation are exactly the same as for Shortest Path in a DAG, but each “min” is replaced by a “max”. We leave the details as an exercise.

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section 7.17).

7.3 Making Change

input: A pair $I = (D, T)$ where $D = \{d_1, d_2, \dots, d_k\}$ is a set of *denominations*, and T is a target (all non-negative integers, with each $d_i \geq 1$ and $1 \in D$).

output: The minimum size of any sequence $C = (c_1, \dots, c_\ell)$ s.t. $c_i \in D$ for all i and $\sum_i c_i = T$.

Lecture Note 5 described a greedy algorithm, and proved that it worked for denominations $D = \{1, 5, 10, 25\}$. However, that algorithm doesn't work for all denominations. For example, it fails for $D = \{1, 3, 4\}$ and target $T = 6$. Next we give a dynamic-programming algorithm that works in all cases. Fix any input instance $(D = \{d_1, d_2, \dots, d_k\}, T)$.

Our first goal is to come up with some divide-and-conquer (recursive or inductive) understanding of the possible ways of making change for T .

To this end, partition the ways of making change for T into up to k types, one for each denomination d_i that could be used first, which can be any denomination d_i that does not exceed T . Among ways to make change for T by first using a coin of denomination d_i , the best uses that coin d_i , and is followed by the best possible way of making change for $T - d_i$. The best way of making change overall is the best among these (up to) k types. So, we could find the best way overall if we knew the best way to make change for $T - d_i$, for each $d_i \leq T$.

Applying this reasoning recursively, for $t \in \{0, 1, \dots, T\}$, define $M(t)$ to be the minimum number of coins needed to make change for t . Then $1 + M(t - d_i)$ is the minimum number of coins needed to make change for t if the first coin is d_i . This reasoning gives the following recurrence relation:

Lemma 7.5. $M(0) = 0$. For $t \geq 1$, $M(t) = \min\{1 + M(t - d_i) : d_i \in D, d_i \leq t\}$.

This recurrence relation naturally gives a recursive algorithm, which we would memoize for efficiency. Alternatively it gives the following iterative algorithm:

```
makeChange( $D = \{d_1, d_2, \dots, d_k\}, T$ ):
1. For  $t = 0, 1, \dots, T$ :
2.    $m[t] = \begin{cases} 0 & \text{if } t = 0 \\ \min\{1 + m[t - d_i] : d_i \in D, d_i \leq t\} & \text{otherwise.} \end{cases}$ 
3. return  $m[T]$ 
```

Correctness follows by the lemma and induction on t , showing that $m[t]$ as computed by the algorithm equals $M(t)$ as defined earlier. Each of the $T + 1$ subproblems ($m[t]$ for each t) takes time $O(k)$ to evaluate, so the total time is $O(kT)$.

Let's summarize what we've figured out in a theorem:

Theorem 7.4. *There is an $O(kT)$ -time algorithm for Making Change.*

Python code: [dynamic_programming/making_change.py](#) (Section 7.17).

7.3.1 Reducing to Shortest Path in a DAG

Many dynamic-programming problems reduce to finding an optimal path in some implicit DAG. For example, we can obtain the above algorithm for Making Change this way. Given an instance (D, T) of Making Change, consider the following instance $(G = (V, E), s, t)$ of Shortest Path in a DAG.

Take $V = \{0, 1, \dots, T\}$. That is, there is a vertex for each subproblem $M(t)$. For each t , add an edge from $t - d$ to t for each $d \in D$ with $d \leq t$. So $E = \{(t - d, t) : t \in V; d \in D; d \leq t\}$. (There is

an edge $(t-d, t)$ if subproblem $t-d$ occurs in the right-hand side of the recurrence for subproblem t .) Give each edge weight 1.

Then each path p in G from 0 to T corresponds to a way of making change totaling T , and vice versa. Specifically, if a path p from 0 to T goes through a sequence of vertices such as $0, c_1, c_1 + c_2, \dots, c_1 + c_2 + \dots + c_n = T$, then it corresponds to making change totaling T using coins in $c = (c_1, c_2, \dots, c_n)$. The length of the path p is the number of coins in c . So, the minimum number of coins needed to make change totaling T equals the distance $\text{dist}(0, T)$ from 0 to T in G , and another algorithm for Making Change would work as follows:

makeChange2(D, T) (*return the min. num. of coins needed to make change for T with denominations in D*)

1. construct the weighted DAG G described above for (D, T)
2. return **shortestPath**($G, 0, T$)

This algorithm is essentially equivalent to the previous dynamic-programming algorithm for Making Change.

Python code: [dynamic_programming/dynamic_programming_paths.py](#) (Section 7.17).

How could you use this idea to solve the following problem? *Given (D, T) , compute the number of ways to make change totaling T using coins with denominations in D .* (There is an ambiguity here: do we consider two ways of making change different if they differ only in the order of their coins? Is $c = (1, 1, 3)$ different than $c' = (3, 1, 1)$? First answer the question assuming that order matters (so these c and c' are different). Then solve it assuming order doesn't matter. Hint: restrict to sequences c such that, say $c_i \geq c_{i+1}$ for each i .)

7.4 Four steps for presenting a dynamic-programming algorithm

In the rest of these notes, we use the following four steps to present each dynamic-programming algorithm:

- 1. Define the subproblems.** Define the subproblems to be solved for a given instance, and state how the final answer is computed from these subproblems. For example, to present the dynamic-programming algorithm for Making Change, the first step would be as follows:

1. Given the instance (D, T) of Making Change, define the following subproblems. For each $t \in \{0, 1, \dots, T\}$, define $M(t)$ to be the minimum number of coins needed to make change (using denominations in D) of total value t . The solution to the problem is $M(T)$.

This tells the reader exactly what subproblems the algorithm will solve, and also how (once they are solved) the algorithm will compute the output for the given instance.

- 2. State the recurrence relation.** The recurrence relation is a divide-and-conquer equation that can be used to compute the value for each subproblem from the values of some smaller subproblems. The left-hand side of the equation is the value of the subproblem. The right-hand side is an expression built from the values of smaller subproblems. For Making Change, the recurrence relation is in Lemma 7.5:

$$2. M(t) = \begin{cases} 0 & \text{if } t = 0 \\ \min\{1 + M(t - d_i) : d_i \in D, d_i \leq t\} & \text{otherwise.} \end{cases}$$

The recurrence relation does not *define* $M(t)$! Rather, it is an identity that holds for $M(t)$ as M is defined in the first step. (To prove correctness of the algorithm, we'll have to prove that the recurrence relation holds.)

The first and second step together determine the algorithm: the algorithm iterate (or recurses) over the set of subproblems to be solved. It solves each subproblem using the recurrence relation, evaluating the right-hand side to determine the value of the left-hand side.

For readers who are familiar with dynamic programming, the first two steps are enough to implicitly define the algorithm, so it is usually unnecessary to give any further definition.

- 3. Prove correctness.** The third step is to prove that (or explain why) the recurrence relation holds. As long as it holds, correctness of the algorithm (iterative, or recursive and memoized) will follow by induction.
- 4. Bound the run time.** Analyze the runtime of the algorithm. This is the sum, over all of the subproblems, of the time to evaluate the right-hand side of the recurrence relation (assuming the smaller subproblems are already solved). For Making Change, this would be enough:

4. There are $T + 1$ subproblems (one for each $t \in \{0, 1, \dots, T\}$; subproblem t is to compute $M(t)$). For each, the time to evaluate the right-hand side of the recurrence for $M(t)$ is $O(k)$ (the number of denominations). So the total time for the algorithm is $O(Tk)$.

7.5 Subset Sum

input: A sequence $x = (x_1, x_2, \dots, x_n)$ of non-negative integers, and an integer target T .

output: True if some subsequence of x sums to T , otherwise False.

(That is, True if there is a subset S of $\{1, 2, \dots, n\}$ such that $\sum_{i \in S} x_i = T$.)

The problem differs from Making Change in that each integer can be used at most once. In what follows, for any subset $S \subseteq \{1, 2, \dots, n\}$, we say the subset *yields* t if the corresponding subsequence of x sums to t , that is, if $\sum_{i \in S} x_i = t$. The underlying collection of objects we are working with is the collection of subsets of $\{1, 2, \dots, n\}$. For our divide-and-conquer approach, we partition this collection into two types of subsets:

- (A) Subsets that don't contain n . That is, subsets of $\{1, 2, \dots, n-1\}$.
- (B) Subsets that do contain n . That is, subsets of the form $S = S' \cup \{n\}$ where $S' \subseteq \{1, 2, \dots, n-1\}$.

Using this, we reduce our problem to two smaller problems:

- (i) There is a type-A subset yielding T if and only if some subset of $\{1, \dots, n-1\}$ yields T .
- (ii) There is a type-B subset yielding T if and only if some subset of $\{1, \dots, n-1\}$ yields $T - x_n$.

Applying this reasoning recursively gives us the algorithm. Here it is, using the four steps:

subproblems and recurrence relation. Fix any input $x = (x_1, \dots, x_n)$ with target T . For each $i \in \{0, 1, \dots, n\}$ and each target $t \in \{0, 1, \dots, T\}$, define

$$S(i, t) = \begin{cases} \text{True} & \text{if there is a subset of } \{1, 2, \dots, i\} \text{ that yields } t, \\ \text{False} & \text{otherwise.} \end{cases}$$

The final answer is given by $S(n, T)$. The recurrence relation is

$$S(i, t) = \begin{cases} \text{True} & \text{if } t = 0 \text{ and } i = 0, \\ \text{False} & \text{if } t > 0 \text{ and } i = 0, \\ S(i-1, t) \text{ or } S(i-1, t-x_i) & \text{if } i > 0 \text{ and } x_i \leq t, \\ S(i-1, t) & \text{if } i > 0 \text{ and } x_i > t. \end{cases}$$

correctness. The base cases of the recurrence relation are true by inspection. The rest holds by the reasoning above. That is, by definition $S(i, t)$ holds if and only if there is a subset of $\{1, \dots, i\}$ yielding t . This occurs if and only if (A) there is such a subset that *doesn't* contain i , or (B) there is one that *does* contain i . (A) occurs if and only if there is a subset of $\{1, \dots, i-1\}$ yielding t , that is, if and only if $S(i-1, t)$ holds. (B) occurs if and only if there is a subset of $\{1, \dots, i-1\}$ yielding $t - x_i$, that is, if and only if $x_i \leq t$ and $S(i-1, t - x_i)$ holds.

run time. There are $O(nT)$ subproblems, and for each the right-hand-side of the recurrence can be evaluated in constant time, so the total time for the algorithm (iterative, or recursive and memoized) is $O(nT)$.

Theorem 7.5. *There is an $O(nT)$ -time algorithm for Subset Sum.*

reduction to finding a path in a graph. Consider the graph with a vertex $v(i, t)$ for each subproblem $S(i, t)$, with edges representing the recurrence-relation dependencies as usual. The problem is then equivalent to asking whether this graph has a path from $v(0, 0)$ to $v(n, T)$.

Python code: [dynamic_programming/subset_sum.py](#) (Section 7.17).

7.6 Knapsack (with long-form proof)

input: A triple $I = (v, w, L)$ where $v = (v_1, v_2, \dots, v_n)$ is a sequence of non-negative *values*, $w = (w_1, w_2, \dots, w_n)$ is a sequence of positive integer *weights*, and L is a non-negative integer *weight limit*. We call each integer $i \in \{1, 2, \dots, n\}$ an *item* of value v_i and weight w_i .

output: The maximum value of any subset $S \subseteq \{1, 2, \dots, n\}$ having weight at most L . (The value of a subset S is $\sum_{i \in S} v_i$. The weight of S is $\sum_{i \in S} w_i$.) In other words, output

$$\max \left\{ \sum_{i \in S} v_i \mid S \subseteq \{1, 2, \dots, n\}, \sum_{i \in S} w_i \leq L \right\}.$$

Here is the algorithm. Fix any input instance $I = (v, w, L)$.

subproblems and recurrence. For each $i \in \{0, 1, 2, \dots, n\}$ and $\ell \in \{0, 1, 2, \dots, L\}$, define $V(i, \ell)$ to be the maximum value achievable using only items $\{1, 2, \dots, i\}$ and total weight at most ℓ . The given instance I has solution $V(n, L)$. The recurrence relation is

$$V(i, \ell) = \begin{cases} 0 & \text{if } i = 0, \\ \max(V(i-1, \ell), V(i-1, \ell - w_i) + v_i) & \text{if } i > 0 \text{ and } w_i \leq \ell, \\ V(i-1, \ell) & \text{if } i > 0 \text{ and } w_i > \ell. \end{cases}$$

correctness. The intuition is as follows. If $i = 0$, then the only subset is the empty set, of value zero. For $i > 0$, consider the subsets for $V(i, \ell)$ —that is, subsets of $\{1, \dots, i\}$ having total weight at most ℓ . Partition these into two types: those that don't include i , and those that do. Those that don't include i are just the subsets for $V(i-1, \ell)$, so the maximum value of any subset of this type is $V(i-1, \ell)$. Those that do include i (which exist only if $w_i \leq \ell$) are of the form $S \cup \{i\}$ where S is any subset for $V(i-1, \ell - w_i)$ (having total weight at most $\ell - w_i$). The maximum value of any such subset $S \cup \{i\}$ is $V(i-1, \ell - w_i) + v_i$. A long-form proof of correctness is on the next page.

time. There are $O(nL)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in constant time. So the total time to compute $V(n, L)$ is $O(nL)$.

Theorem 7.6. *There is an $O(nL)$ -time algorithm for Knapsack*

reduction to Longest Path in a DAG. Alternatively, we can reduce the problem to an instance of Longest Path in a DAG. The graph has a vertex $v(i, \ell)$ for each subproblem. If $i > 0$ then $v(i, \ell)$ has an incoming edge from $v(i-1, \ell)$ of weight zero, and (if $\ell \geq w_i$) an incoming edge from $v(i-1, \ell - w_i)$ of weight v_i . If $i = 0$ and $\ell > 0$, then give $v(i, \ell)$ an incoming edge of weight zero from $v(0, 0)$. Then the longest path from $v(0, 0)$ to $v(i, \ell)$ corresponds to a maximum-value subset of $\{1, 2, \dots, i\}$ of total weight at most ℓ .

(continued)

Long-form proof of correctness for Knapsack

Here is a long-form proof of correctness for the recurrence relation. For brevity in the proof, for any $S \subseteq \{1, \dots, n\}$, define $v(S) = \sum_{j \in S} v_j$ to be the total value of items in S . Likewise, define $w(S) = \sum_{j \in S} w_j$ to be the total weight of items in S . So, using this notation, $V(i, \ell) = \max \{v(S) : S \subseteq \{1, 2, \dots, i\}, w(S) \leq \ell\}$.

Proof (long form).

1. Consider any $i \in \{0, \dots, n\}$ and $\ell \in \{0, 1, \dots, L\}$.
2. Recall that $V(i, \ell) = \max \{v(S) : S \subseteq \{1, 2, \dots, i\}, w(S) \leq \ell\}$.
 - 3.1. Case 1. In the case $i = 0$, the set S must be empty, so $V(i, \ell) = 0$, so the recurrence holds.
 - 4.1. Case 2. Next consider the case when $i > 0$ and $w_i \leq \ell$.
 - 4.2. Consider the subsets $S \subseteq \{1, 2, \dots, i\}$ such that $w(S) \leq \ell$.
 - 4.3. Partition these into two types: (A) those that don't include i , and (B) those that do.
 - 4.4. $V(i, \ell)$ is the maximum value of any subset of either type.
 - 4.5. The type-A subsets are just the subsets of $\{1, \dots, i-1\}$ with $w(S) \leq \ell$.
 - 4.6. So (by definition of $V(i-1, \ell)$) the maximum value of any type-A subset is $V(i-1, \ell)$.
 - 4.7. The type-B subsets are those of the form $S' \cup \{i\}$ where $S' \subseteq \{1, \dots, i-1\}$ has $w(S') \leq \ell - w_i$.
 - 4.8. The maximum value of such a set $S' \cup \{i\}$ is achieved when S' itself has maximum total value.
 - 4.9. So, the maximum value of any type-B subset $S' \cup \{i\}$ is $V(i-1, \ell - w_i) + v_i$.
(Here we implicitly use that $w_i \leq \ell$, so that $V(i-1, \ell - w_i)$ is well-defined.)
 - 4.10. By Steps 4.4, 4.6 and 4.9, $V(i, \ell) = \max(V(i-1, \ell), V(i-1, \ell - w_i) + v_i)$.
 - 4.11. So the recurrence holds in this case.
 - 5.1. Case 3. Finally consider the remaining case, when $i > 0$ and $w_i > \ell$.
 - 5.2. Each w_j is non-negative, so no subset of $\{1, \dots, i\}$ that contains i has total weight at most ℓ .
 - 5.3. So the subsets $S \subseteq \{1, \dots, i\}$ with $w(S) \leq \ell$ are just the $S \subseteq \{1, \dots, i-1\}$ with $w(S) \leq \ell$.
 - 5.4. So $V(i, \ell) = V(i-1, \ell)$, and the recurrence holds in this case.
6. By the case analysis (Cases 1–3) above, the recurrence holds. □

Python code: [dynamic_programming/knapsack.py](#) (Section 7.17).

7.7 Weighted Activity Selection

Warm-up—counting pairwise-disjoint subsets

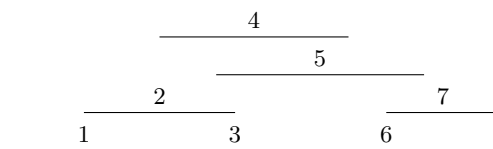
input: A collection $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ of intervals.

For convenience we assume $e_1 \leq e_2 \leq \dots \leq e_n$. (If not, sort them.)

output: The number of subsets $C' \subseteq C$ of C that are pairwise disjoint.

(These include the subsets of size one or zero.)

Here is an example input, C , consisting of seven intervals. (The interval heights are irrelevant.)



Following Activity Selection, call each interval in C a *job*, and interpret each job $[s_i, e_i]$ as an activity that lasts from start time s_i to end time e_i . A pairwise-disjoint subset of C is then just a collection of jobs can be done without doing more than one job at any time.

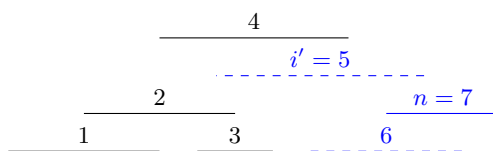
Our first goal is to come up with some recursive or inductive understanding of the collection of pairwise-disjoint subsets. To start, partition the pairwise-disjoint subsets into two types:

- (A) Pairwise-disjoint subsets of C that don't contain the last job $[s_n, e_n]$.
- (B) Pairwise-disjoint subsets of C that do contain the last job $[s_n, e_n]$.

The subsets of type (A) are just the pairwise-disjoint subsets of the first $n - 1$ jobs, that is, of the collection $C_A = \{[s_1, e_1], [s_2, e_2], \dots, [s_{n-1}, e_{n-1}]\}$.

What about the subsets of type (B)? Any subset of type (B) contains the job $[s_n, e_n]$, so can't contain any job that overlaps with job $[s_n, e_n]$, that is, can't contain any job $[s_i, e_i]$ with $i \geq i'$, where $i' = \min\{i : e_i \geq s_n\}$ is the index of the first job that overlaps job $[s_n, e_n]$.

In the example, $n = 7$ and $i' = 5$. The jobs overlapping the last job $[s_n, e_n]$ are shown in blue:



So, the subsets of type (B) consist of $[s_n, e_n]$, together with any pairwise-disjoint subset of the first $i' - 1$ jobs, that is, of the collection $C_B = \{[s_1, e_1], [s_2, e_2], \dots, [s_{i'-1}, e_{i'-1}]\}$. The collection C_B can be empty. (In the example, C_B contains the first four jobs.)

This approach gives the following dynamic-programming algorithm.

subproblems and recurrence. For $j \in \{0, 1, 2, \dots, n\}$, define $N(j)$ to be the number of pairwise-disjoint subsets of $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$. The output will be $N(n)$. The recurrence relation is

$$N(j) = \begin{cases} 1 & \text{if } j = 0 \\ N(j-1) + N(i'-1), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$$

correctness. The recurrence relation follows from the preceding discussion. We don't give a careful proof here.

run time. There are $n + 1$ subproblems. The time to solve each depends on how we compute i' . Here is the iterative version of the algorithm:

```

countSchedules( $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ ):
1. for  $j = 0, 1, \dots, n$ :
2.   set  $M[j] = \begin{cases} 1 & \text{if } j = 0 \\ M[j - 1] + M[i' - 1], \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0 \end{cases}$ 
3. return  $M[n]$ 

```

— iterative version

If we assume constant-time arithmetic (which is not realistic here!) the algorithm can be implemented to run in $\Theta(n \log n)$ time. The main obstacle is computing i' in each iteration. Since the jobs are sorted by end times, i' can be computed in each iteration in $O(\log n)$ time using binary search. Then the n iterations take total time $\Theta(n \log n)$, as does the sort.

But the value of $M[i]$ can be as large as 2^i , so in the worst case requires n bits to store. Then each addition takes $\Theta(n)$ time, so each of the n iterations takes $\Theta(n)$ time, for a total time of $\Theta(n^2)$. Here is what we have figured out:

Theorem 7.7. *Given a collection C of n intervals, the number of pairwise-disjoint subsets of C can be computed in $O(n^2)$ time.*

Weighted Activity Selection

input: A collection $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$ of intervals (called *jobs*), and, for each job $[s_i, e_i]$, an associated value $v_i \geq 0$. Again assume $e_1 \leq e_2 \leq \dots \leq e_n$.

output: The maximum, over all pairwise-disjoint subsets C' of C , of the total value of intervals in C' , that is $\sum_{[s_i, e_i] \in C'} v_i$.

subproblems and recurrence relation. Fix an instance. For $j \in \{0, 1, \dots, n\}$, define $V(j)$ to be the maximum value of any pairwise-disjoint subset from the first j intervals $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$. The final answer is $V(n)$. The recurrence relation is

$$V(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(V(j-1), V(i'-1) + v_j), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$$

correctness. As before, we divide the pairwise-disjoint subsets of $\{[s_1, e_1], [s_2, e_2], \dots, [s_j, e_j]\}$ into two types:

- (A) The pairwise-disjoint subsets that don't include the last job $[s_j, e_j]$.
- (B) The pairwise-disjoint subsets that do include the last job $[s_j, e_j]$.

Each pairwise-disjoint subset is of type (A) or type (B), so $V(j)$ is the maximum value of any subset of type (A), or the maximum value of any subset of type (B), whichever is larger.

As before, the subsets of type (A) are just the pairwise-disjoint subsets from the first $j - 1$ jobs, so the maximum value of any subset of type (A) is $V(j - 1)$.

And, also as before, each subset of type (B) consists of $[s_j, e_j]$, together with any pairwise-disjoint subset from the first $i' - 1$ jobs, where i' is the index of the first job that overlaps job j . That

is, where $i' = \min\{i : e_i \geq s_j\}$. It follows that the maximum value of any subset of type (B) is $V(i' - 1) + v_j$. This gives the recurrence relation.

Correctness of the algorithm (iterative, or recursive and memoized) follows from the recurrence relation by induction on j (that the algorithm does in fact set $W[j]$ to $V(j)$ as previously defined).

run time. Here is the iterative algorithm.

```

maxSchedule( $C = \{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}, v$ ):
1. for  $j = 0, 1, 2, \dots, n$ :
2.   set  $W[j] = \begin{cases} 0 & \text{if } j = 0 \\ \max(W[j-1], W[i'-1] + v_j), \text{ where } i' = \min\{i : e_i \geq s_j\} & \text{if } j > 0. \end{cases}$ 
3. return  $W[n]$ 
    
```

— iterative version

In each iteration, the index i' can be found using binary search on the jobs (which are ordered by end times) in $O(\log n)$ time. Then the n iterations take total time $\Theta(n \log n)$. This proves the following theorem:

Theorem 7.8. *There is an $O(n \log n)$ -time algorithm for Weighted Activity Selection.*

reduction to Longest Path. Consider the DAG that has a vertex for each subproblem $V(j)$, with edges for each dependency (of weight zero or v_j) in the recurrence relation. The problem is equivalent to finding a max-weight path in this graph from the vertex for $V(0)$ to the vertex $V(n)$.

7.8 Longest Common Subsequence (with long-form proof)

input: A pair $I = (A[1..m], B[1..n])$ of sequences.

output: The length of the longest common subsequence of A and B . (That is, the longest sequence that is a subsequence of both A and B .)

Here is the algorithm. Fix any input instance $(A[1..m], B[1..n])$.

subproblems and recurrence. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $L[i, j]$ to be the length of the longest common subsequence between $A[1..i]$ and $B[1..j]$. (The first i elements in A , and the first j elements in B .) The final answer is $L(m, n)$. Here is the recurrence relation:

$$L(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + L(i - 1, j - 1) & \text{if } i, j > 0 \text{ and } A[i] = B[j], \\ \max(L(i - 1, j), L(i, j - 1)) & \text{if } i, j > 0 \text{ and } A[i] \neq B[j]. \end{cases}$$

correctness. The intuition is as follows. In the case that $A[i] = B[j]$, there is a longest common subsequence of $A[1..i]$ and $B[1..j]$ that ends in the common final element $A[i]$, and, before that final element, consists of a longest common subsequence of $A[1..i - 1]$ and $B[1..j - 1]$ (of length $L(i - 1, j - 1)$). So in that case, $L(i, j) = 1 + L(i - 1, j - 1)$.

In the case that $A[i] \neq B[j]$, every common subsequence y of $A[1..i]$ and $B[1..j]$ either (i) doesn't end with $A[i]$, or (ii) doesn't end with $B[j]$ (or both). In Case (i), y is a common subsequence of $A[1..i - 1]$ and $B[1..j]$. In Case (ii), y is a common subsequence of $A[1..i]$ and $B[1..j - 1]$. Thus, the set of common subsequences of $A[1..i]$ and $B[1..j]$ is the union of two (not necessarily disjoint) subsets:

- (i) the set of common subsequences of $A[1..i - 1]$ and $B[1..j]$, and
- (ii) the set of common subsequences of $A[1..i]$ and $B[1..j - 1]$.

So the length of the longest common subsequence of $A[1..i]$ and $B[1..j]$ is the length of the longest sequence in these two subsets. That is, $L(i, j) = \max(L(i - 1, j), L(i, j - 1))$. A long-form proof of correctness is on the next page.

time. There are $(n + 1) \cdot (m + 1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So algorithm (iterative, or recursive and memoized) takes time $\Theta(nm)$.

Theorem 7.9. *There is an $O(nm)$ -time algorithm for Longest Common Subsequence.*

reduction to Longest Path. The problem is equivalent an instance of Longest Path on a graph with a vertex v_{ij} for each subproblem $L(i, j)$, and an edge for each dependency in the recurrence relation (with appropriate weight, 1 or 0).

(continued)

Long-form proof of correctness for Longest Common Subsequence

Here's a long-form proof that the recurrence relation is correct.

Proof (long form).

1. Consider any pair (i, j) with $i \in \{0, \dots, m\}$ and $j \in \{0, \dots, n\}$.
 - 2.1. Case 1. Consider the case that $0 \in \{i, j\}$. ($A[1..i]$ or $B[1..j]$ is the empty sequence.)
 - 2.2. The only common subsequence is the empty one, so $L(i, j) = 0$, and the recurrence is correct.
 - 3.1. Case 2. Otherwise $i \geq 1$ and $j \geq 1$.
 - 3.2.1. Case 2.1. First consider the sub-case when $A[i] = B[j]$.
 - 3.2.2. We first show that $L(i, j) \geq L(i - 1, j - 1) + 1$.
 - 3.2.3. Let $x = (x_1, x_2, \dots, x_\ell)$ be a common subsequence of $A[1..i - 1]$ and $B[1..j - 1]$, of length $L(i - 1, j - 1)$ (it exists by definition of L).
 - 3.2.4. Let $x' = (x_1, x_2, \dots, x_\ell, A[i])$ be obtained by appending $A[i]$ to x .
 - 3.2.5. Since $A[i] = B[j]$, x' is a subsequence of both $A[1..i]$ and $B[1..j]$.
 - 3.2.6. So $L(i, j) \geq |x'| = L(i - 1, j - 1) + 1$.
 - 3.2.7. Next we show $L(i, j) \leq L(i - 1, j - 1) + 1$.
 - 3.2.8. Let $y = (y_1, y_2, \dots, y_k)$ be a common subsequence of $A[1..i]$ and $B[1..j]$, of length $L(i, j)$.
 - 3.2.9. Let $y' = (y_1, y_2, \dots, y_{k-1})$ be obtained from y by removing the last element.
 - 3.2.10. Sequence y is a subsequence of $A[1..i]$, so y' is a subsequence of $A[1..i - 1]$. (Verify!)
 - 3.2.11. Similarly, y' is a subsequence of $B[1..j - 1]$.
 - 3.2.12. So y' is a common subsequence of $A[1..i - 1]$ and $B[1..j - 1]$.
 - 3.2.13. So $L(i - 1, j - 1)$ is at least $|y'| = L(i, j) - 1$. Rearranging gives $L(i, j) \leq L(i - 1, j - 1) + 1$.
 - 3.2.14. By this and Step 3.2.6, $L(i, j) = L(i - 1, j - 1) + 1$.
 - 3.2.15. So $L(i, j)$ equals the right-hand side of the recurrence in Case 2.1 (when $A[i] = B[j]$).
 - 3.3.1. Case 2.2. In the remaining sub-case $A[i] \neq B[j]$.
 - 3.3.2. First we show $L(i, j) \geq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.3. Let z be a common subsequence of $A[1..i - 1]$ and $B[1..j]$ of length $L(i - 1, j)$.
 - 3.3.4. Then z is also a common subsequence of $A[1..i]$ and $B[1..j]$.
 - 3.3.5. So $L(i, j) \geq |z| = L(i - 1, j)$.
 - 3.3.6. By a symmetric argument, $L(i, j) \geq L(i, j - 1)$. So $L(i, j) \geq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.7. Next we show that $L(i, j) \leq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.8. Let $y = (y_1, y_2, \dots, y_\ell)$ be a common subsequence of $A[1..i]$ and $B[1..j]$ of length $L(i, j)$.
 - 3.3.9.1. Case 2.2.1 First consider the case that $y_\ell \neq A[i]$.
 - 3.3.9.2. Then y must be a subsequence of $A[1..i - 1]$ (as well as $B[1..j]$).
 - 3.3.9.3. So $L(i - 1, j) \geq |y| = L(i, j)$. That is $L(i, j) \leq L(i - 1, j)$.
 - 3.3.9.4. So $L(i, j) \leq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.10.1. Case 2.2.2 Otherwise (since $A[i] \neq B[j]$) it must be that $y_\ell \neq B[j]$.
 - 3.3.10.2. By an argument symmetric to the one in Case 2.2.1, $L(i, j) \leq L(i, j - 1)$.
 - 3.3.10.3. So $L(i, j) \leq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.11. By Blocks 3.3.9 and 3.3.10, $L(i, j) \leq \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.12. By this and Step 3.3.6, $L(i, j) = \max(L(i - 1, j), L(i, j - 1))$.
 - 3.3.13. So $L(i, j)$ equals the right-hand side of the recurrence in Case 2.2 ($A[i] \neq B[j]$).
 - 3.4. By Blocks 3.2 and 3.3, $L(i, j)$ equals the right-hand side of the recurrence in Case 2 ($0 \notin \{i, j\}$).
4. By Blocks 2 and 3, $L(i, j)$ equals the right-hand side of the recurrence (for any i, j). $\square$

Python code: [dynamic_programming/longest_common_subsequence.py](#) (Section 7.17).

7.9 Edit Distance and Sequence Alignment

input: A pair $I = (A[1..m], B[1..n])$ of strings.

output: The *edit distance* between A and B . This is the minimum cost of any *alignment* between A and B . An alignment is a matching between characters of A and characters of B such that the matching doesn't *cross*: that is, there are no two matched pairs $(A[i], B[j])$ and $(A[i'], B[j'])$ where $i < i'$ but $j > j'$. The cost of an alignment is the total number of unmatched characters (in both sides) plus the number of matched pairs $(A[i], B[j])$ where $A[i] \neq B[j]$.

Equivalently, the edit distance is the minimum number of edit operations that transforms A into B , where each edit operation either deletes a character, inserts a character, or replaces any one character by some other character.

intuition. Fix any input instance $(A[1..m], B[1..n])$. The collection of objects we are working with is the set of alignments between A and B . To derive the algorithm, we classify these alignments into three types:

- (A) Those that don't match $A[m]$ at all.
- (B) Those that don't match $B[n]$ at all.
- (C) Those that match $A[m]$ to $B[n]$.

Every alignment is of one of these types, because an alignment cannot both match $A[m]$ to some character in $B[1..n-1]$ and match $B[n]$ to some character in $A[1..m-1]$. (This is because in such a matching those matched pairs would cross.) It is possible for an alignment to be both type (A) and type (B), but this is not a problem for the algorithm. Next we consider the three types of alignments, and for each we observe that the problem of finding a minimum-cost alignment of that type reduces to a smaller subproblem:

- (A) An alignment of type (A) is just an alignment of $A[1..m-1]$ and $B[1..n]$, so the minimum cost of any alignment of type (A) is just the minimum cost of any alignment between $A[1..m-1]$ and $B[1..n]$.
- (B) Likewise, the minimum cost of any alignment of type (B) is just the minimum cost of any alignment between $A[1..m]$ and $B[1..n-1]$.
- (C) Any alignment of type (C) matches $A[m]$ to $B[n]$, and then the rest of the alignment is just an alignment between $A[1..m-1]$ and $B[1..n-1]$. So, the minimum cost of any alignment of type (C) is just the minimum cost of any alignment between $A[1..m-1]$ and $B[1..n-1]$, plus the cost of matching $A[m]$ to $B[n]$ (1 if $A[m] \neq B[n]$, otherwise 0).

When we apply this reasoning recursively, each subproblem that arises is to compute the minimum cost of any alignment between some prefix of A and some prefix of B . That is, between $A[1..i]$ and $B[1..j]$ for some i and j . Here is the algorithm:

subproblems and recurrence relation. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $M(i, j)$ to be the minimum cost of any alignment between $A[1..i]$ and $B[1..j]$. The final answer is $M(m, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} \max(i, j) & \text{if } i = 0 \text{ or } j = 0, \\ \min(M(i-1, j) + 1, M(i, j-1) + 1, M(i-1, j-1)) & \text{if } i, j > 0 \text{ and } A[i] = B[j], \\ \min(M(i-1, j) + 1, M(i, j-1) + 1, M(i-1, j-1) + 1) & \text{if } i, j > 0 \text{ and } A[i] \neq B[j]. \end{cases}$$

correctness. Correctness follows from the ideas sketched above under “intuition.”

time. There are $(n + 1) \cdot (m + 1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) take $O(nm)$.

Theorem 7.10. *There is an $O(nm)$ -time algorithm for Edit Distance.*

Python code: [dynamic_programming/edit_distance.py](#) (Section 7.17).

Extending the algorithm to Sequence Alignment

KT Section 6.6 considers a generalization of Edit Distance called Sequence Alignment:

input: A pair $I = (A[1..m], B[1..n])$ of strings, *gap penalty* $\delta > 0$, and *mismatch cost function* α defining, for each character a in A and b in B , a *mismatch cost* $\alpha(a, b) \geq 0$.

output: The minimum cost of any alignment of A and B , where an alignment is a non-crossing matching (as defined for Edit Distance), but the cost of the alignment is δ times the total number of unmatched characters (in both sides), plus the sum over the matched pairs $(A[i], B[j])$ of $\alpha(A[i], B[j])$.

Edit Distance is the special case when $\delta = 1$ and $\alpha(a, b) = 1$ for $a \neq b$, and $\alpha(a, b) = 0$ for $a = b$. Following the same reasoning as for the algorithm for Edit Distance gives the following algorithm. Fix any input instance $(A[1..m], B[1..n])$.

subproblems and recurrence relation. For each pair (i, j) with $i \in \{0, 1, \dots, m\}$ and $j \in \{0, 1, \dots, n\}$, define $M(i, j)$ to be the minimum cost of any alignment between $A[1..i]$ and $B[1..j]$. The final answer is $M(m, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} \delta \max(i, j) & \text{if } i = 0 \text{ or } j = 0 \\ \min(M(i-1, j) + \delta, M(i, j-1) + \delta, M(i-1, j-1) + \alpha(A[i], B[j])) & \text{if } i, j > 0. \end{cases}$$

correctness. Correctness follows by reasoning similar to the reasoning for the correctness of the recurrence relation for Edit Distance.

time. There are $(n + 1) \cdot (m + 1) = O(nm)$ subproblems, and for each, the right-hand side of the recurrence takes constant time to evaluate (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) takes time $\Theta(nm)$.

Theorem 7.11. *There is an $O(nm)$ -time algorithm for Sequence Alignment.*

7.10 Longest Palindromic Subsequence (with long-form proof)

input: A string $X[1..n]$ (a sequence of characters).

output: The maximum length of any subsequence of X that is a palindrome. (A palindrome is a sequence that is the same as its reverse. The subsequence need not be contiguous. For example, “yobananaboy” is a palindromic subsequence of “your urban cabana bat toy”.)

Here is the algorithm. Fix any input instance $X[1..n]$.

subproblems and recurrence. For each i and j with $1 \leq i \leq n+1$ and $i-1 \leq j \leq n$, define $M(i, j)$ to be the maximum length of any palindromic subsequence of $X[i..j]$. (When $j = i-1$, $X[i..j]$ is the empty string, so $M(i, i-1) = 0$.) The given instance X has solution $M(1, n)$. The recurrence relation is

$$M(i, j) = \begin{cases} 0 & \text{if } j < i, \\ 1 & \text{if } j = i, \\ \max(M(i, j-1), M(i+1, j), 2 + M(i+1, j-1)) & \text{if } j > i \text{ and } X[i] = X[j], \\ \max(M(i, j-1), M(i+1, j)) & \text{if } j > i \text{ and } X[i] \neq X[j]. \end{cases}$$

(We can simplify this somewhat per the comment at the end below.)

correctness. The base cases hold by inspection. Here’s the intuition for the remaining cases. Consider the palindromic subsequences of $X[i..j]$. Categorize these into three types: (A) those that include $X[i]$ and $X[j]$ (necessarily starting with $X[i]$ and ending with $X[j]$), (B) those that don’t include $X[i]$, (C) those that don’t include $X[j]$. Every subsequence is of one (or more) of these types. So the maximum length of any such subsequence is the maximum length of any type-A, type-B, or type-C subsequence.

First consider the case that $X[i] = X[j]$. Then those of type A are of the form $X[i], S, X[j]$ where S is any palindromic subsequence of $X[i+1..j-1]$; so the longest of type A has length $2 + M(i+1, j-1)$. Those of type B are just palindromic subsequences of $X[i+1..j]$; the longest of these has length $M(i+1, j)$. Those of type C are just palindromic subsequences of $X[i..j-1]$; the longest of these has length $M(i, j-1)$. So the recurrence holds in this case.

In the remaining case ($X[i] \neq X[j]$), there are none of type A, because any palindromic subsequence has to start and end with the same character. The analysis of the other two types is the same as in the case $X[i] = X[j]$. So the recurrence holds in this case. A long-form proof of correctness is on the next page.

time. There are $O(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in constant time. So the algorithm (iterative, or memoized recursive) takes time $\Theta(n^2)$.

Theorem 7.12. *There is an $O(n^2)$ -time algorithm for Longest Palindromic Subsequence.*

reduction to Longest Path. Consider the DAG with a node for each subproblem and a directed edge for each dependency (with appropriate weight 0 or 2), and additional directed edges from the node for $M(0, 0)$ to the nodes for each $M(i, i)$ with $1 \leq i \leq n$ (of weight 1) and to the nodes for each $M(i+1, i)$ with $0 \leq i \leq n$ (the empty substrings, of weight 0). The problem is equivalent to finding a longest path in this DAG from the node for $M(0, 0)$ to the node for $M(1, n)$.

simplifying the recurrence. In the case that $X[i] = X[j]$, for any type-B or type-C subsequence, there is always a type-A subsequence that is at least as long. (We can use a “greedy” exchange argument to prove this; we omit the details.) So the maximum in the recurrence for this case is always achieved by the third term, $2 + M(i + 1, j - 1)$, and the recurrence simplifies to

$$M(i, j) = \begin{cases} 0 & \text{if } j < i, \\ 1 & \text{if } j = i, \\ 2 + M(i + 1, j - 1) & \text{if } j > i \text{ and } X[i] = X[j], \\ \max(M(i, j - 1), M(i + 1, j)) & \text{if } j > i \text{ and } X[i] \neq X[j]. \end{cases}$$

Long-form proof of correctness

Here is a long-form proof that the first recurrence is correct.

Proof (long form).

1. Consider any i and j with $1 \leq i \leq j \leq n$. Note that $j - i + 1$ is the length of the string $X[i..j]$.
 - 2.1. Case 1. In the case that $j - i + 1 \leq 1$, the string $X[i..j]$ has length one or zero.
 - 2.2. So it is its own longest palindromic subsequence, $M(i, j) = j - i + 1$, and the recurrence holds.
 - 3.1. Case 2. Next consider the case that $X[i] = X[j]$ (and $j - i + 1 \geq 2$).
 - 3.2. Consider all possible palindromic subsequences of $X[i..j]$.
 - 3.3. Categorize these into three types:
 - (A) those that include $X[i]$ and $X[j]$,
 - (B) those that don't include $X[i]$, and
 - (C) those that don't include $X[j]$.
 - 3.4. Each palindromic subsequence of $X[i..j]$ is of one (or more!) of these types.
 - 3.5. So $M(i, j)$ is the maximum length of any type-A, type-B, or type-C subsequence.
 - 3.6. The type-A's are of the form $X[i], s, X[j]$, where s is any palindromic subseq. of $X[i + 1..j - 1]$.
 - 3.7. The length of such a subsequence $X[i], s, X[j]$ is maximized when the length of s is maximized.
 - 3.8. By definition of $M(i + 1, j - 1)$, the maximum length of any such s is $M(i + 1, j - 1)$.
 - 3.9. So the maximum length of any type-A subsequence is $2 + M(i + 1, j - 1)$.
 - 3.10. The type-B subsequences are just palindromic subsequences of $X[i + 1..j]$.
 - 3.11. So (by definition of $M(i + 1, j)$) the maximum length of any type-B subsequence is $M(i + 1, j)$.
 - 3.12. Similarly, the maximum length of any type-C subsequence is $M(i, j - 1)$.
 - 3.13. By Steps 3.5, 3.9, 3.11, and 3.12, $M(i, j) = \max(2 + M(i + 1, j - 1), M(i + 1, j), M(i, j - 1))$.
 - 3.14. So the recurrence holds in this case.
 - 4.1. Case 3. Finally consider the remaining case, when $X[i] \neq X[j]$ and $j - i + 1 \geq 2$.
 - 4.2. Since $X[i] \neq X[j]$, no palindromic subsequence of $X[i..j]$ starts with $X[i]$ and ends with $X[j]$.
 - 4.3. So, the reasoning in Case 2 applies, except that there are no type-A subsequences.
 - 4.4. And by that reasoning, $M(i, j) = \max(M(i + 1, j), M(i, j - 1))$.
 - 4.5. So the recurrence holds in this case.
5. By the case analysis (Cases 1–3) above, the recurrence holds. □

Python code: [dynamic_programming/longest_palindrome.py](#) (Section 7.17).

7.11 Paragraph Layout

input: A pair $I = (N, w[1..n])$ where N is positive integer, and $w[1..n]$ is an array of words (strings), each of length at most N .

output: A *minimum-cost layout* of w , of width at most N .

A layout breaks the sequence $w[1], w[2], \dots, w[n]$ of words into *lines*, by inserting line breaks after some of the words. For example, suppose $N = 6$ and the given sequence of words is $W[1] = \text{aaa}$, $W[2] = \text{bb}$, $W[3] = \text{cc}$, $W[4] = \text{dddd}$.

Two possible layouts (each with three lines) are shown to the right. The *width* of each line is the number of characters in the line, including the spaces that separate the words. The three lines in Layout 1 have widths 6, 2, and 4. The three lines in Layout 2 have widths 3, 5, and 4.

layout 1	layout 2
aaa.bb	aaa...
cc....	bb.cc.
dddd..	dddd..

Each line in the layout must have width at most N . We define the *cost* of a layout as follows. For each line, define the *gap* of the line to be N minus the width. The three lines in Layout 1 have gaps 0, 4, and 2. The three lines in Layout 2 have gaps 3, 1, and 2. The cost of a layout is the sum, over its lines, of the gap of the line *squared*. Layout 1 has cost $0^2 + 4^2 + 2^2 = 20$. Layout 2 has cost $3^2 + 1^2 + 2^2 = 14$. The goal is to compute a layout of minimum cost.

Minimizing the cost tends to reduce the variation in the gaps, giving a better-looking layout.

Here is the algorithm. Fix any instance (N, w) .

subproblems. For each $i \in \{0, 1, \dots, n\}$, let $M(i)$ be the optimal cost for the subproblem $(N, w[1..i])$ formed by the first i words. The solution to the given problem $(N, w[1..n])$ is $M(n)$.

recurrence. First, define $\text{width}(w[i..j])$ (for $i \leq j$) to be the width of a line formed by words $w[i], w[i+1], \dots, w[j]$. That is, $\text{width}(w[i..j]) = \sum_{h=i}^j (|w[h]| + 1) - 1$. Then the gap of such a line is $N - \text{width}(w[i..j])$. Here is the recurrence for M :

$$M(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min \left\{ M(i-1) + (N - \text{width}(w[i..j]))^2 \mid i \in \{1, \dots, j\}, \text{width}(w[i..j]) \leq N \right\} & \text{if } j > 0. \end{cases}$$

correctness. The intuition is as follows. Each possible layout for words $w[1..j]$ consists (for some i) of a layout for the words in $w[1..i-1]$, followed by one last line with the words in $w[i..j]$. Any $i \in \{1, 2, \dots, j\}$ with $\text{width}(w[i..j]) \leq N$ gives such a layout.

time. There are $n+1 = \Theta(n)$ subproblems. For each, evaluating the right-hand side of the recurrence takes time proportional to the maximum number of words that fit in the line. This is at worst $O(\min(n, N))$. So the algorithm (iterative, or recursive and memoized) takes time $O(n \min(n, N))$.

Theorem 7.13. *There is an $O(n \min(n, N))$ -time algorithm for Paragraph Layout.*

The problem is equivalent to finding a shortest path, in the DAG with vertices for each subproblem and appropriate edge costs, from the node for $M(0)$ to the node for $M(n)$.

Python code: [dynamic_programming/paragraph_layout.py](#) (Section 7.17).

7.12 Minimum-Weight Polygon Triangulation

input: A sequence $p[1..n]$ of points in the plane, forming (in clockwise order) the vertices $p[1], p[2], \dots, p[n]$ of a convex polygon.

output: The minimum weight of any *triangulation* of p . A triangulation of p is a set of *chords* (diagonals—straight-line segments connecting pairs of vertices) that subdivides the convex polygon into triangular faces. The weight of the triangulation is the sum of the lengths of the chords. We include the boundary segments (with endpoints p_i, p_{i+1}) as chords.

subproblems and recurrence relation. Fix any input instance $p[1..n]$. For each pair (i, j) with $1 \leq i < j \leq n$, and $j - i \geq 1$, define $T(i, j)$ to be the optimal cost for the subproblem $p[i..j]$ formed by the $j - i + 1 \geq 2$ points $p[i], p[i+1], \dots, p[j]$. The final answer is $T(1, n)$. The recurrence relation is

$$T(i, j) = \begin{cases} d(p[i], p[j]) & \text{if } j = i + 1, \\ \min \left\{ T(i, k) + T(k, j) + d(p[i], p[j]) \mid k \in \{i + 1, i + 2, \dots, j - 1\} \right\} & \text{if } j > i + 1. \end{cases}$$

Above, $d(p[i], p[j])$ is the Euclidean distance between points $p[i] = (x_i, y_i)$ and $p[j] = (x_j, y_j)$, that is, $d(p[i], p[j]) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

correctness. The intuition is as follows. Each triangulation of $p[i..j]$ can be obtained as follows for some $k \in \{i + 1, \dots, j - 1\}$: take any triangulation of $p[i..k]$, and any triangulation of $p[k..j]$, and add the chord with endpoints $p[i]$ and $p[j]$. (The latter chord must be in the triangulation, and point $p[k]$ is the third vertex on the triangle that has that chord as one side. Draw a picture and consider some examples for more intuition. Make sure to consider the boundary cases when $k \in \{i + 1, j - 1\}$.)

time. There are $\Theta(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in time $O(n)$ (assuming the smaller subproblems are already solved). So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.14. *There is an $O(n^3)$ -time algorithm for Min-Weight Polygon Triangulation.*

does not reduce to Shortest Paths! The standard trick of creating a graph with one vertex per subproblem $T(i, j)$ doesn't work, because, in the right-hand side of the recurrence for $T(i, j)$, for each k , the term includes *two* subproblems $T(i, k)$ and $T(k, j)$.

Python code: [dynamic_programming/min_wt_triangulation.py](#) (Section 7.17).

7.13 Optimal Binary Search Tree

input: A sequence $p[1..n]$ of *frequencies*, where $p[j]$ is the frequency of *key* j .

output: The minimum cost of any *binary search tree* T for the n keys. A binary search tree is a rooted binary tree with the n keys $\{1, 2, \dots, n\}$ as nodes, *in search-tree order*. (So, at each node, the key is larger than the keys in the node's left subtree, and smaller than the keys in the node's right subtree.) The cost of T is the weighted average depth of the keys (where the root has depth 1, so $\text{depth}_T(j)$ is the number of nodes visited when searching for key j):

$$\text{cost}(T) = \sum_{j=1}^n p[j] \text{depth}_T(j).$$

For any such tree T and key j , define T_j to be the subtree whose root node has key j , containing j and all its descendants.

Observation 7.1. *For any tree T and key j , the set of keys in the subtree T_j is some contiguous sequence $\{i, i+1, \dots, k\}$, where $i \leq j \leq k$. So any such subtree T_j is a binary search tree for the Optimal Binary Search Tree instance $p[i..k]$.*

The cost of T_j for its instance $p[i..k]$ is the weighted average depth of keys in T_j , that is, $\text{cost}(T_j) = \sum_{h=i}^k p[h] \text{depth}_{T_j}(h)$. For any such subproblem $p[i..k]$, define $\text{wt}(p[i..k]) = \sum_{h=i}^k p[h]$ to be the sum of the frequencies in the instance $p[i..k]$.

Observation 7.2. *Consider any subtree T_j . Let $p[i..k]$ be the subproblem it solves. Let T_ℓ and T_r be its left and right subtrees. Then $\text{cost}(T_j) = \text{wt}(p[i..k]) + \text{cost}(T_\ell) + \text{cost}(T_r)$.*

Also, the subproblems solved by T_ℓ and T_r are $p[i..j-1]$ and $p[j+1..k]$.

Here is the algorithm. Fix any input instance $p[1..n]$.

subproblems and recurrence relation. For each $i \in \{1, 2, \dots, n+1\}$ and $k \in \{0, 1, \dots, n\}$ with $i \leq k+1$, define $M(i, k)$ to be the minimum cost of any binary search tree for the subproblem $p[i..k]$ (which is empty if $i = k+1$). The given instance $p[1..n]$ has solution $M(1, n)$. Here's the recurrence relation:

$$M(i, k) = \begin{cases} 0 & \text{if } i > k \\ \min \left\{ \text{wt}(p[i..k]) + M(i, j-1) + M(j+1, k) \mid j \in \{i, i+1, \dots, k\} \right\} & \text{if } i \leq k. \end{cases}$$

correctness. The intuition is that an optimal tree for $p[i..k]$ can be obtained by choosing some key $j \in \{i, i+1, \dots, k\}$ for the root, then choosing any optimal tree T_ℓ for subproblem $p[i..j-1]$ as the left subtree, and any optimal tree T_r for subproblem $p[j+1..k]$ as the right subtree. The subtree for $p[i..k]$ is then a subtree T_j , and, by Observation 7.2, its cost is $\text{cost}(T_j) = \text{wt}(p[i..k]) + \text{cost}(T_\ell) + \text{cost}(T_r) = \text{wt}(p[i..k]) + M(i, j-1) + M(j+1, k)$.

time. There are $O(n^2)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in time $O(n)$. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.15. *There is an $O(n^3)$ -time algorithm for Optimal Binary Search Tree.*

does not reduce to Shortest Paths! The standard trick of creating a graph with one vertex per subproblem $M(i, k)$ doesn't work, because, in the right-hand side of the recurrence for $M(i, k)$, for each j , the term includes *two* subproblems $M(i, j-1)$ and $M(j+1, k)$.

Python code: [dynamic_programming/optimal_binary_search_tree.py](#) (Section 7.17).

7.14 Single-Source Shortest Paths with Negative Weights

input: An edge-weighted digraph $G = (V, E)$, and source vertex s . (The weights can be negative.)

output: For each vertex $v \in V$, the shortest-path distance from s to v .

subproblems and recurrence relation. Fix an input instance $G = (V, E)$. For each $w \in V$ and $i \in \{0, 1, 2, \dots, n-1\}$ define $M(w, i)$ to be the minimum weight of any s -to- w path in G having exactly i edges. (Notes: (i) If there is no such path, then $M(w, i) = \infty$. (ii) The path doesn't have to be simple.) The final answer is an array d where $d[w] = \min \{M(w, i) : i \in \{0, 1, \dots, n-1\}\}$ for all $w \in V$. (If no negative-weight cycle is reachable from s , then by [Observation 5.7](#) of [Lecture Note 5](#) (applied to the subgraph of G reachable from s) there is a shortest path from s to w that is simple, so has at most $n-1$ edges, so this will be the true shortest-path distance.) Here is the recurrence relation:

$$M(w, i) = \begin{cases} 0 & \text{if } i = 0 \text{ and } w = s, \\ \infty & \text{if } i = 0 \text{ and } w \neq s, \\ \min \{M(u, i-1) + \text{wt}(u, w) \mid (u, w) \in E\} & \text{if } i > 0. \end{cases}$$

As usual we define the minimum of an empty set to be ∞ .

correctness. For the first two (base) cases, the recurrence holds by inspection. For the main case of the recurrence ($i > 0$), the intuition is as follows. $M(w, i)$ is the minimum weight of any i -edge path from s to w . As $i > 0$, each such path consists of an $(i-1)$ -edge path from s to u , followed by the edge (u, w) , for some in-neighbor u of w . For a given final edge (u, w) , among the i -edge paths ending in (u, w) , the shortest consists of a shortest $(i-1)$ -edge path to u , followed by the edge (u, w) , so has weight $M(u, i-1) + \text{wt}(u, w)$.

time. For a given $M(w, i)$, the time to evaluate the right-hand side of the recurrence is proportional to the in-degree of w , plus 1. Summing over the subproblems, the total time is proportional to $\sum_{i=0}^{n-1} \sum_{w \in V} 1 + \text{in-degree}(w) = \sum_{i=0}^{n-1} n + m = n(n+m)$. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n(n+m))$, where $n = |V|$ and $m = |E|$. We can assume without loss of generality that $m = \Omega(n)$ —if not we can preprocess the graph to remove vertices that have no incident edges, after which $m \geq n/2$. Then this is $\Theta(nm)$.

Theorem 7.16. *There is an $O(nm)$ -time algorithm for Single-Source Shortest Paths with Neg. Wts.*

Detecting negative cycles. The algorithm can be modified to detect negative-weight cycles reachable from s , if there are any, as follows. Recall that $d[v] = \min_{i \leq n-1} M(v, i)$ is the s -to- v distance as computed by the algorithm. The graph has a negative-weight cycle reachable from s if and only if there is an edge (u, w) such that $d[w] > d[u] + \text{wt}(u, w)$.¹

¹By the reasoning above, if there is no such cycle, the algorithm computes distances correctly, that is, $d[v] = \text{dist}(s, v)$ for all v . This implies $d[w] \leq d[u] + \text{wt}(u, w)$, because $\text{dist}(s, w) \leq \text{dist}(s, u) + \text{wt}(u, w)$ for all (u, w) .

Conversely, suppose $d[w] \leq d[u] + \text{wt}(u, w)$ for all edges (u, w) . Then $\text{wt}(u, w) \geq d[w] - d[u]$. For any cycle C reachable from s , summing this inequality over the edges in C gives

$$\text{wt}(C) = \sum_{(u,w) \in C} \text{wt}(u, w) \geq \sum_{(u,w) \in C} d[w] - d[u] = 0.$$

Bellman-Ford algorithm. The well-known and closely related Bellman-Ford algorithm achieves the same time bound. One variant of Bellman-Ford is as follows:

```
Bellman-Ford( $G = (V, E), s$ )
1. initialize  $d[s] = 0$  and  $d[v] = \infty$  for  $v \neq s$ 
2. for  $j = 1, 2, \dots, n - 1$ :
3.   for  $w \in V$ :
4.      $d[w] = \min(d[w], \min\{d[u] + \text{wt}(u, w) : (u, w) \in E\})$ 
5. return  $d$ 
```

To show correctness, we can use a similar argument to show by induction on j that, after each iteration j , for all vertices $v \in V$, (i) $d[v]$ is *at most* $\min_{i \leq j} M(v, i)$ and (ii) $d[v]$ is the weight of some s -to- v path. It follows that, if the graph has no negative cycles reachable from s , then, after iteration $n - 1$, $d[v] = \text{dist}(s, v)$ for all $v \in V$.

Python code: [dynamic_programming/Bellman_Ford.py](#) (Section 7.17).

7.15 All Pairs Shortest Paths / Floyd-Warshall Alg. (w/ l. f. proof)

input: An edge-weighted digraph $G = (V, E)$ with no negative-weight cycles. To ease notation assume $V = \{1, 2, \dots, n\}$.

output: An array of shortest-path distances d such that for all $u, w \in V$, $d[u, w] = \text{dist}(u, w)$ is the minimum shortest-path distance in G from u to w .

subproblems and recurrence relation. Fix an input instance $G = (V, E)$. For each $u, w \in V$ and $i \in \{0, 1, 2, \dots, n\}$ define $M(u, w, i)$ to be the minimum weight of any u -to- w path in G whose interior vertices are all in $\{1, 2, \dots, i\}$. (If there is no such path, then $M(u, w, i) = \infty$.) The final answer is an array d where $d[u, w] = M(u, w, n)$ for all $u, w \in V$. Here is the recurrence relation:

$$M(u, w, i) = \begin{cases} 0 & \text{if } u = w \\ \text{wt}(u, w) & \text{if } u \neq w, i = 0, \text{ and } (u, w) \in E \\ \infty & \text{if } u \neq w, i = 0, \text{ and } (u, w) \notin E \\ \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1)) & \text{if } u \neq w \text{ and } i \neq 0 \end{cases}$$

correctness. The intuition for the main case of the recurrence (when $i \neq 0$ and $u \neq w$) is as follows. Say a path p from u to w in G is *allowed for* $M(u, w, i)$ if the interior vertices of p (the vertices in p other than the endpoints u and w) are all in $\{1, 2, \dots, i\}$. Then $M(u, w, i)$ is the minimum weight of any allowed path for $M(u, w, i)$. Partition the allowed paths for $M(u, w, i)$ into three types: (A) those that don't have i as an interior vertex, (B) those on which i occurs once as an interior vertex, and (C) those on which i occurs more than once as an interior vertex.

The type-A paths are just those that are allowed for $M(u, w, i-1)$. So the minimum weight of any type-A path is $M(u, w, i-1)$.

The type-B paths are those paths p that can be obtained as follows. Let p_1 be any allowed path for $M(u, i, i-1)$. Let p_2 be any allowed path for $M(i, w, i-1)$. Obtain path $p = p_1 \cup p_2$ by appending p_2 to p_1 , so $\text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2)$. A minimum weight type-B path p can be obtained by taking p_1 to be a minimum-weight allowed path for $M(u, i, i-1)$, and taking p_2 to be a minimum-weight allowed path for $M(i, w, i-1)$, so p has weight $M(u, i, i-1) + M(i, w, i-1)$.

We can ignore any type-C path p , because by removing cycles (from i to i) from p , we can find a type-B path with $\text{wt}(p') \leq \text{wt}(p)$. A long-form proof of correctness is below.

time. There are $\Theta(n^3)$ subproblems. For each, the right-hand side of the recurrence can be evaluated in $\Theta(1)$ time. So the algorithm (iterative, or recursive and memoized) takes time $\Theta(n^3)$.

Theorem 7.17. *There is an $O(n^3)$ -time algorithm for All-Pairs Shortest Paths.*

Long-form proof of correctness for the Floyd-Warshall recurrence relation

Here's a long-form proof that the recurrence is correct.

Proof (long form).

1. Following the explanation, for any $u, w \in V$ and $i \in \{0, \dots, n\}$, say a path p from u to w is *allowed for* $M(u, w, i)$ if all interior vertices of p are in $\{1, 2, \dots, i\}$. Then $M(u, w, i)$ is the minimum weight of any allowed path for $M(u, w, i)$.
- 2.1. Consider any $u, w \in V$ and $i \in \{0, 1, \dots, n\}$.
- 2.2. First we consider the boundary cases of the recurrence.

- 2.3.1. Case 1. Suppose $u = w$.
- 2.3.2. The empty path (of weight zero) is a u -to- w path.
- 2.3.3. There is no path of less weight (as the graph has no negative cycles).
- 2.3.4. So $M(u, w, i) = 0$, and the recurrence holds in Case 1.
- 2.4.1. Case 2. Suppose $u \neq w$, $i = 0$, and $(u, w) \in E$.
- 2.4.2. As $i = 0$, the allowed paths for $M(u, w, i)$ are the u -to- w paths with no interior vertices.
- 2.4.3. The only such path consists of just the edge (u, w) , with weight $\text{wt}(u, w)$.
- 2.4.4. So $M(u, w, i) = \text{wt}(u, w)$, and the recurrence holds in Case 2.
- 2.5.1. Case 3. Suppose $u \neq w$, $i = 0$, and $(u, w) \notin E$.
- 2.5.2. As $i = 0$, the allowed paths for $M(u, w, i)$ are the u -to- w paths with no interior vertices.
- 2.5.3. Since (u, w) is not an edge, there is no such allowed path.
- 2.5.4. So $M(u, w, i) = \infty$, and the recurrence holds in Case 3.
- 2.6. Next we consider the main case of the recurrence.
- 2.7.1. Case 4. In the remaining case, $u \neq w$, and $i \neq 0$.
- 2.7.2. We show that $M(u, w, i) = \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.3. Every allowed path for $M(u, w, i-1)$ is also allowed for $M(u, w, i)$.
- 2.7.4. So $M(u, w, i) \leq M(u, w, i-1)$.
- 2.7.5. Next we show $M(u, w, i) \leq M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.6. If $M(u, i, i-1) + M(i, w, i-1) = \infty$, then the above inequality holds, so assume otherwise.
- 2.7.7. Let p_1 be an allowed path for $M(u, i, i-1)$ with $\text{wt}(p_1) = M(u, i, i-1)$.
(It exists, as $M(u, i, i-1)$ is finite.)
- 2.7.8. Let p_2 be an allowed path for $M(i, w, i-1)$ with $\text{wt}(p_2) = M(i, w, i-1)$.
(It exists, as $M(i, w, i-1)$ is finite.)
- 2.7.9. Then $p = p_1 \cup p_2$ is a path from u to w whose interior vertices are all in $\{1, \dots, i\}$.
- 2.7.10. That is, p is an allowed path for $M(u, w, i)$.
- 2.7.11. So $M(u, w, i) \leq \text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2) = M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.12. By this and Step 2.7.4, $M(u, w, i) \leq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.13. Next we show $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.14. If $M(u, w, i)$ is infinite, then the inequality above holds, so assume otherwise.
- 2.7.15. Let p be an *acyclic* allowed path for $M(u, w, i)$ with $\text{wt}(p) = M(u, w, i)$.
(It exists, because $M(u, w, i)$ is finite, and the graph has no negative-weight cycles.)
- 2.7.16.1. Case 4.1 First consider the case that i is not an interior vertex of p .
- 2.7.16.2. Then p is allowed for $M(u, w, i-1)$, so $M(u, w, i-1) \leq \text{wt}(p) = M(u, w, i)$.
- 2.7.16.3. So $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.17.1. Case 4.2 Otherwise i occurs as an interior vertex of p , just once since p is acyclic.
- 2.7.17.2. Let p_1 be the prefix of p ending at i . Let p_2 be the suffix of p starting at i .
- 2.7.17.3. Since i occurs just once as an interior vertex of p , i is not an interior vertex of p_1 or p_2 .
- 2.7.17.4. So p_1 is an allowed path for $M(u, i, i-1)$, and p_2 is an allowed path for $M(i, w, i-1)$.
- 2.7.17.5. So $\text{wt}(p_1) \geq M(u, i, i-1)$, and $\text{wt}(p_2) \geq M(i, w, i-1)$.
- 2.7.17.6. So $M(u, w, i) = \text{wt}(p) = \text{wt}(p_1) + \text{wt}(p_2) \geq M(u, i, i-1) + M(i, w, i-1)$.
- 2.7.17.7. So $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.18. By Blocks 2.7.16 and 2.7.17, $M(u, w, i) \geq \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.19. By this and Step 2.7.12, $M(u, w, i) = \min(M(u, w, i-1), M(u, i, i-1) + M(i, w, i-1))$.
- 2.7.20. So the recurrence holds in Case 4.
- 2.8. By Blocks 2.3-2.7, the recurrence holds for $M(u, w, i)$. □

Python code: [dynamic_programming/Floyd_Warshall.py](#) (Section [7.17](#)).

7.16 External sources on dynamic programming

	JE	DPV	KT	KT slides	CLRS	MIT video	Python code
intro	3	6.1	6.1	-	15	19 (2011)	-
paths in DAGS	8.5	4.7	-	-	24.2	-	py
Making Change	Ex. 1; Ch. 3	Ex. 6.17–6.19	-	-	-	-	py
Subset Sum and Knapsack	3.8	6.4	6.4	6.4	-	21 (2011)	py, py
Weighted Activity Selection	-	-	6.1	6.1	-	-	-
Longest Common Subsequence	-	-	-	-	15.4	15 (2005)	py
Edit Distance	3.7	6.3	6.6	6.6	-	21 (2011)	py
Longest Palindromic Subseq.	-	-	-	-	Prob. 15-2	10 (2015)	py
Paragraph Layout	-	-	-	-	Prob. 15–4	20 (2011)	py
Min-Wt Polygon Triangulation	-	Prob. 6.12	-	-	-	-	py
Optimal Binary Search Tree	3.9	-	-	-	15.5	-	py
Bellman-Ford	8.7	4.6	6.8	6.8	24.1	18 (2005)	py
Floyd-Warshall	9.8	6.6	-	-	25.2	11 (2015)	py
Segmented Least Squares	-	-	6.3	6.3	-	-	-

The table entries in blue above (for the slides, videos, and code) are hyperlinked to their sources. In case you have a printed copy of these notes, you can find them via the following URLs:

KT slides: <https://www.cs.princeton.edu/~wayne/kleinberg-tardos/>

2005 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>

2011 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/>

2015 videos: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/>

Python code: <https://github.com/nealeyoung/algs101/tree/main/src/code>

7.17 Python code

This section lists the Python code for this lecture note. Click on a file's name to open the file on GitHub. Copying code from GitHub is more reliable than copying it from this PDF, since some PDF viewers drop the leading spaces (the indentation) of copied lines.

[dynamic_programming/graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable, Iterator
from typing import Any

type Edge = tuple[int, int]

def out_neighbor_lists[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> dict[Vertex, list[Vertex]]:
    out_neighbors: dict[Vertex, list[Vertex]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> dict[Vertex, list[Vertex]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[Edge]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[Edge]:
        if acyclic:
            priority = random.sample(range(n), n)

        edges: set[Edge] = set()
        while len(edges) < n_edges:
            u, w = random.choice(vertices), random.choice(vertices)
            if u != w:
                if acyclic and priority[u] < priority[w]:
```

```

        u, w = w, u
        edges.add((u, w))
    return edges

    if m <= n_pairs / 2:
        edges = random_edge_set(m)
    else:
        # if most edges are present, faster to generate the complement
        all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
        assert len(all_pairs) == n_pairs
        edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[Edge]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

def random_weighted_digraph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge], dict[Edge, int]]:
    import random

    V, E = random_digraph(*args, **kwargs)
    weights = {e: random.choice(range(10000)) for e in E}
    return V, E, weights

def random_weighted_graph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[Edge], dict[Edge, int]]:
    V, E, weights = random_weighted_digraph(*args, **kwargs, acyclic=True)
    E += [(w, u) for u, w in E]
    for u, w in E:
        weights[(w, u)] = weights[(u, w)]
    return V, E, weights

def depth_first_search[Vertex: Hashable](
    V: Iterable[Vertex],
    E: Iterable[tuple[Vertex, Vertex]],
    previsit: Callable[[Vertex, Vertex | None], object] | None = None,
    postvisit: Callable[[Vertex, Vertex | None], object] | None = None,
    source: Vertex | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[Vertex] = set()

    def DFS1(u: Vertex, parent: Vertex | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
```

```

        postvisit(u, parent)

    if source is not None:
        DFS1(source)
    else:
        for v in V:
            if v not in visited:
                DFS1(v)

def breadth_first_search_parents[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]], source: Vertex | None = None
) -> Iterator[tuple[Vertex, Vertex]]:
    from collections import deque

    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[Vertex] = deque()
    queue.append(source)
    visited = {source}

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in visited:
                visited.add(w)
                queue.append(w)
                yield (w, u)

    # for v in V:
    #     if v not in d:
    #         d[v] = inf

    # return d, parents

def topological_sort[Vertex: Hashable](
    V: Iterable[Vertex], E: Iterable[tuple[Vertex, Vertex]]
) -> list[Vertex]:
    finished: list[Vertex] = []

    def postvisit(u: Vertex, parent: Vertex | None) -> None:
        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    return list(reversed(finished))

```

[dynamic_programming/dynamic_programming_paths.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf
from typing import Any

```

```

from graph_library import (
    in_neighbor_lists,
    random_digraph,
    random_weighted_digraph,
    topological_sort,
)

def count_paths[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> int:
    in_neighbors = in_neighbor_lists(V, E)

    N: Any # a memoized function if recursively, else a table (dict)
    if recursively:
        @memoize
        def N(w: Vertex) -> int:
            return 1 if w == s else sum(N(u) for u in in_neighbors[w])

        return N(t)
    else:
        N = {}
        for w in topological_sort(V, E):
            N[w] = 1 if w == s else sum(N[u] for u in in_neighbors[w])
        return N[t]

def test_count_paths(n: int = 10, m: int | None = None) -> None:
    V, E = random_digraph(n, m, acyclic=True)
    n1 = count_paths(V, E, 0, n - 1)
    n2 = count_paths(V, E, 0, n - 1, recursively=True)
    assert n1 == n2

def shortest_path[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> float:
    in_neighbors = in_neighbor_lists(V, E)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:
        @memoize
        def D(w: Vertex) -> float:
            return (
                0 if w == s else min((D(u) + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

        return D(t)

```

```

    else:
        D = {}
        for w in topological_sort(V, E):
            D[w] = (
                0 if w == s else min((D[u] + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

        return D[t]

def longest_path[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
    t: Vertex,
    recursively: bool = False,
) -> float:
    in_neighbors = in_neighbor_lists(V, E)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def D(w: Vertex) -> float:
            return (
                0 if w == s else max((D(u) + weights[u, w] for u in in_neighbors[w]), default=-inf)
            )

        return D(t)

    else:
        D = {}
        for w in topological_sort(V, E):
            D[w] = (
                0 if w == s else max((D[u] + weights[u, w] for u in in_neighbors[w]), default=-inf)
            )

        return D[t]

def test_shortest_path(n: int = 10, m: int | None = None) -> None:
    V, E, weights = random_weighted_digraph(n, m, acyclic=True)
    d3 = shortest_path(V, E, weights, 0, n - 1)
    d4 = shortest_path(V, E, weights, 0, n - 1, recursively=True)
    assert d3 == d4

def test_longest_path(n: int = 10, m: int | None = None) -> None:
    V, E, weights = random_weighted_digraph(n, m, acyclic=True)
    d1 = longest_path(V, E, weights, 0, n - 1)
    d2 = longest_path(V, E, weights, 0, n - 1, recursively=True)

    weights = {e: -weights[e] for e in weights}
    d3 = shortest_path(V, E, weights, 0, n - 1)
    d4 = shortest_path(V, E, weights, 0, n - 1, recursively=True)

    assert d1 == d2 == -d3 == -d4

def make_change(denominations: Collection[int], target: int, recursively: bool = False) -> float:

```

```

assert isinstance(target, int) and target >= 0
assert all(isinstance(d, int) and d >= 1 for d in denominations)

M: Any # a memoized function if recursively, else a table (dict)
if recursively:

    @memoize
    def M(i: int) -> float:
        return (
            0 if i == 0 else min((1 + M(i - d) for d in denominations if d <= i), default=inf)
        )

    return M(target)

else:
    M = {}
    for i in range(target + 1):
        M[i] = (
            0 if i == 0 else min((1 + M[i - d] for d in denominations if d <= i), default=inf)
        )
    return M[target]

def make_change_via_shortest_paths(denominations: Collection[int], target: int) -> float:
    assert isinstance(target, int) and target >= 0
    assert all(isinstance(d, int) and d >= 1 for d in denominations)

    V = range(target + 1)
    E = [(i, i + d) for i in V for d in denominations if i + d in V]
    weights = {e: 1 for e in E}
    return shortest_path(V, E, weights, 0, target)

def test_making_change(n: int = 10) -> None:
    import random

    assert n >= 1
    denominations = set(random.randrange(1, 25) for _ in range(5))
    target = random.randrange(n, 2 * n)
    d1 = make_change(denominations, target)
    d2 = make_change(denominations, target, recursively=True)
    d3 = make_change_via_shortest_paths(denominations, target)

    assert d1 == d2 == d3

if __name__ == "__main__":
    for _ in range(20):
        test_count_paths(50)
        test_shortest_path(50)
        test_longest_path(50)
        test_making_change()

    print("test passed")

```

[dynamic_programming/making_change.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection

```

```

from functools import cache as memoize
from math import inf
from typing import Any

def make_change(D: Collection[int], T: int, recursively: bool = False) -> float:
    assert all(d > 0 and isinstance(d, int) for d in D)
    assert T >= 0 and isinstance(T, int)

    m: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def m(t: int) -> float:
            """min coins to make change for t"""
            return 0 if t == 0 else min((1 + m(t - d) for d in D if d <= t), default=inf)

        return m(T)
    else:
        m = {}
        for t in range(T + 1):
            m[t] = 0 if t == 0 else min((1 + m[t - d] for d in D if d <= t), default=inf)
        return m[T]

def test_make_change(n: int = 10, T: int = 100) -> None:
    import random

    D = [random.randrange(1, 100) for _ in range(n)]
    m1 = make_change(D, T)
    m2 = make_change(D, T, recursively=True)
    assert m1 == m2

if __name__ == "__main__":
    for _ in range(20):
        test_make_change()

    print("test passed")

```

[dynamic_programming/subset_sum.py](#)

```

#!/usr/bin/env python3

from collections.abc import Collection
from functools import cache as memoize
from typing import Any

def subset_sum(X: Collection[int], T: int, recursively: bool = False) -> bool:
    assert all(x >= 0 and isinstance(x, int) for x in X)
    assert T >= 0 and isinstance(T, int)

    X = list(X)
    n = len(X)

    S: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize

```

```

def S(i: int, t: int) -> bool:
    """is there a subset of the first j items summing to t ?"""
    return (
        True
        if t == 0 and i == 0
        else False
        if t > 0 and i == 0
        else S(i - 1, t) or S(i - 1, t - X[i - 1])
        if i > 0 and X[i - 1] <= t
        else S(i - 1, t)
    )

    return S(n, T)
else:
    S = {}
    for i in range(n + 1):
        for t in range(T + 1):
            S[i, t] = (
                True
                if t == 0 and i == 0
                else False
                if t > 0 and i == 0
                else S[i - 1, t] or S[i - 1, t - X[i - 1]]
                if i > 0 and X[i - 1] <= t
                else S[i - 1, t]
            )
    return S[n, T]

def test_subset_sum(n: int = 10, T: int = 100) -> None:
    import random

    X = [random.randrange(1, 100) for _ in range(n)]
    s1 = subset_sum(X, T)
    s2 = subset_sum(X, T, recursively=True)
    assert s1 == s2

if __name__ == "__main__":
    for _ in range(20):
        test_subset_sum()

    print("test passed")

```

dynamic_programming/knapsack.py

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def knapsack(w: Sequence[int], v: Sequence[float], L: int, recursively: bool = False) -> float:
    assert all(wi > 0 and isinstance(wi, int) for wi in w)
    assert all(vi >= 0 for vi in v)
    assert L >= 0 and isinstance(L, int)
    assert len(w) == len(v)

    n = len(w)

```

```

V: Any # a memoized function if recursively, else a table (dict)
if recursively:

    @memoize
    def V(i: int, ell: int) -> float:
        """max value achievable
        using first items {0,1,2,...,i-1}
        and total weight <= ell"""
        return (
            0
            if i == 0 or ell == 0
            else V(i - 1, ell)
            if w[i - 1] > ell
            else max(V(i - 1, ell), V(i - 1, ell - w[i - 1]) + v[i - 1])
        )

    return V(n, L)
else:
    V = {}
    for i in range(n + 1):
        for ell in range(L + 1):
            V[i, ell] = (
                0
                if i == 0 or ell == 0
                else V[i - 1, ell]
                if w[i - 1] > ell
                else max(V[i - 1, ell], V[i - 1, ell - w[i - 1]] + v[i - 1])
            )
    return V[n, L]

def test_knapsack(n: int = 10, L: int = 100) -> None:
    import random

    w = [random.randrange(1, 100) for _ in range(n)]
    v = [random.randrange(0, 100) for _ in range(n)]
    k1 = knapsack(w, v, L)
    k2 = knapsack(w, v, L, recursively=True)
    assert k1 == k2, (w, v, L, k1, k2)

if __name__ == "__main__":
    for _ in range(20):
        test_knapsack()

    print("test passed")

```

[dynamic_programming/longest_increasing_subsequence.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize

from longest_common_subsequence import longest_common_subsequence

def longest_increasing_subsequence(A: Sequence[float], recursively: bool = False) -> int:

```

```

n = len(A)

if recursively:

    @memoize
    def LIS(j: int) -> int:
        return 1 + max((LIS(i) for i in range(j) if A[i] < A[j]), default=0)

    return max((LIS(i) for i in range(n)), default=0)

else:
    L: dict[int, int] = {}
    for j in range(n):
        L[j] = 1 + max((L[i] for i in range(j) if A[i] < A[j]), default=0)
    return max((L[i] for i in range(n)), default=0)

def LIS_via_LCS(A: Sequence[float]) -> float:
    B = sorted(set(A)) # distinct values, so common subsequences are strictly increasing
    return longest_common_subsequence(A, B)

if __name__ == "__main__":

    def test_LIS(n: int = 10) -> None:
        import random

        A = [random.randrange(n) for _ in range(n)]
        L1 = longest_increasing_subsequence(A)
        L2 = longest_increasing_subsequence(A, recursively=True)
        L3 = LIS_via_LCS(A)
        assert L1 == L2 == L3

    for _ in range(20):
        test_LIS(50)

    print("test passed")

```

dynamic_programming/longest_common_subsequence.py

```

#!/usr/bin/env python3
# mypy: allow-redefinition

from collections.abc import Sequence
from functools import cache as memoize

from dynamic_programming_paths import longest_path

def longest_common_subsequence(
    A: Sequence[object], B: Sequence[object], recursively: bool = False
) -> int:
    m, n = len(A), len(B)

    if recursively:

        @memoize
        def LCS(i: int, j: int) -> int:
            if 0 in {i, j}:

```

```

        return 0
    elif A[i - 1] == B[j - 1]:
        return 1 + LCS(i - 1, j - 1)
    else:
        return max(LCS(i, j - 1), LCS(i - 1, j))

    return LCS(m, n)

else:
    L: dict[tuple[int, int], int] = {}
    for i in range(m + 1):
        for j in range(n + 1):
            if 0 in {i, j}:
                L[i, j] = 0
            elif A[i - 1] == B[j - 1]:
                L[i, j] = 1 + L[i - 1, j - 1]
            else:
                L[i, j] = max(L[i, j - 1], L[i - 1, j])
    return L[m, n]

def LCS_via_longest_path(A: Sequence[object], B: Sequence[object]) -> float:
    m, n = len(A), len(B)

    vertices = [(i, j) for i in range(m + 1) for j in range(n + 1)]
    edges = (
        [(i - 1, j), (i, j), 0] for i, j in vertices
    ) + (
        [(i, j - 1), (i, j), 0] for i, j in vertices
    ) + [
        ((i - 1, j - 1), (i, j), 1)
        for i, j in vertices
        if i > 0 and j > 0 and A[i - 1] == B[j - 1]
    ]
    )
    weights = {(u, v): wt for u, v, wt in edges if min(u) >= 0}
    edges = weights.keys()
    return longest_path(vertices, edges, weights, (0, 0), (m, n))

if __name__ == "__main__":

    def test_LCS(m: int = 10, n: int = 10) -> None:
        import random

        A = [random.randrange(m) for _ in range(m)]
        B = [random.randrange(n) for _ in range(n)]
        L1 = longest_common_subsequence(A, B)
        L2 = longest_common_subsequence(A, B, recursively=True)
        L3 = LCS_via_longest_path(A, B)
        assert L1 == L2 == L3

    for _ in range(20):
        test_LCS(50, 40)

    print("test passed")

```

[dynamic_programming/edit_distance.py](#)

```

#!/usr/bin/env python3
# mypy: allow-redefinition

```

```

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

from dynamic_programming_paths import shortest_path

def edit_distance(A: Sequence[object], B: Sequence[object], recursively: bool = False) -> int:
    m, n = len(A), len(B)

    D: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def D(i: int, j: int) -> int:
            return (
                max(i, j)
                if i == 0 or j == 0
                else D(i - 1, j - 1)
                if A[i - 1] == B[j - 1]
                else 1 + min(D(i, j - 1), D(i - 1, j), D(i - 1, j - 1))
            )

        return D(m, n)

    else:
        D = {}
        for i in range(m + 1):
            for j in range(n + 1):
                D[i, j] = (
                    max(i, j)
                    if i == 0 or j == 0
                    else D[i - 1, j - 1]
                    if A[i - 1] == B[j - 1]
                    else 1 + min(D[i, j - 1], D[i - 1, j], D[i - 1, j - 1])
                )
        return D[m, n]

def edit_distance_via_shortest_path(A: Sequence[object], B: Sequence[object]) -> float:
    m, n = len(A), len(B)

    vertices = [(i, j) for i in range(m + 1) for j in range(n + 1)]

    def match(i: int, j: int) -> bool:
        return i > 0 and j > 0 and A[i - 1] == B[j - 1]

    edges = (
        [((i - 1, j - 1), (i, j), int(not match(i, j))) for i, j in vertices]
        + [((i - 1, j), (i, j), 1) for i, j in vertices if not match(i, j)]
        + [((i, j - 1), (i, j), 1) for i, j in vertices if not match(i, j)]
    )
    weights = {(u, v): wt for u, v, wt in edges if min(u) >= 0}
    edges = weights.keys()
    return shortest_path(vertices, edges, weights, (0, 0), (m, n))

if __name__ == "__main__":

    def test_edit_distance(m: int = 10, n: int = 10) -> None:

```

```

import random

A = [random.randrange(m) for _ in range(m)]
B = [random.randrange(n) for _ in range(n)]
D1 = edit_distance(A, B)
D2 = edit_distance(A, B, recursively=True)
D3 = edit_distance_via_shortest_path(A, B)
assert D1 == D2 == D3, (D1, D2, D3)

for _ in range(15):
    test_edit_distance(30, 40)

A, B = "SNOWY", "SUNNY"
print("edit distance between", A, "and", B, "is", edit_distance(A, B))
print("test passed")

```

[dynamic_programming/longest_palindrome.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def longest_palindromic_subsequence(X: Sequence[object], recursively: bool = False) -> int:
    n = len(X)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(i: int, j: int) -> int:
            return (
                0
                if j < i
                else 1
                if j == i
                else 2 + M(i + 1, j - 1)
                if j > i and X[i - 1] == X[j - 1]
                else max(M(i, j - 1), M(i + 1, j))
            )

        return M(1, n)

    else:
        M = {}
        for j in range(-1, n + 1):
            for i in range(j + 1, 0, -1):
                M[i, j] = (
                    0
                    if j < i
                    else 1
                    if j == i
                    else 2 + M[i + 1, j - 1]
                    if j > i and X[i - 1] == X[j - 1]
                    else max(M[i, j - 1], M[i + 1, j])
                )
        return M[1, n]

```

```

if __name__ == "__main__":

    def test(n: int = 10) -> None:
        import random

        X = [random.randrange(10) for _ in range(n)]
        l1 = longest_palindromic_subsequence(X)
        l2 = longest_palindromic_subsequence(X, recursively=True)
        assert l1 == l2

    def brute_force(X: Sequence[object]) -> int:
        from itertools import combinations

        return max(len(c) for r in range(len(X) + 1) for c in combinations(X, r) if c == c[::-1])

    def test_small(n: int = 8) -> None:
        import random

        X = [random.randrange(3) for _ in range(random.randrange(n + 1))]
        assert longest_palindromic_subsequence(X) == brute_force(X), X

    for _ in range(15):
        test(30)
    for _ in range(100):
        test_small()

    print("test passed")

```

[dynamic_programming/paragraph_layout.py](#)

```

#!/usr/bin/env python3

from collections.abc import Iterator, Sequence
from functools import cache as memoize
from typing import Any

def paragraph_layout(N: int, w: Sequence[str], recursively: bool = False) -> int:
    """return min cost of any width-N layout of w"""

    # note: w is indexed from 0..(n-1) instead of 1..n

    assert all(len(wi) <= N for wi in w)

    n = len(w)

    @memoize
    def width(i: int, j: int) -> int:
        """return width of line with words w[i..j]"""
        assert 0 <= i <= j <= n - 1
        return len(w[i]) + (0 if i == j else 1 + width(i + 1, j))

    def i_range(j: int) -> Iterator[int]:
        """return range of i's s.t. width(i, j) <= N, in time O(j-i)"""
        assert 0 <= j <= n - 1
        i = j
        while i >= 0 and width(i, j) <= N:
            yield i
            i -= 1

```

```

M: Any # a memoized function if recursively, else a table (dict)
if recursively:

    @memoize
    def M(j: int) -> int:
        """min cost of layout of w[0..j]"""
        return 0 if j == -1 else min(M(i - 1) + (N - width(i, j)) ** 2 for i in i_range(j))

    return M(n - 1)
else:
    M = {}
    M[-1] = 0
    for j in range(n):
        M[j] = min(M[i - 1] + (N - width(i, j)) ** 2 for i in i_range(j))
    return M[n - 1]

def test_paragraph_layout(N: int = 15, n: int = 10) -> None:
    import random

    alphabet = "abcdefghijklmnopqrstuvwxyz"
    words = [random.choice(alphabet) * random.randrange(2, 6) for _ in range(n)]
    c1 = paragraph_layout(N, words)
    c2 = paragraph_layout(N, words, recursively=True)
    assert c1 == c2

    c1 = paragraph_layout(6, ["aaa", "bb", "cc", "dddd"])
    assert c1 == 14, c1

if __name__ == "__main__":
    for _ in range(20):
        test_paragraph_layout()

    print("test passed")

```

dynamic_programming/min_wt_triangulation.py

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def min_wt_triangulation(p: Sequence[tuple[float, float]], recursively: bool = False) -> float:
    # note: p is indexed from 0..(n-1) instead of 1..n

    n = len(p)

    @memoize
    def d(i: int, j: int) -> float:
        return sum((p[i][k] - p[j][k]) ** 2 for k in (0, 1)) ** 0.5

    T: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def T(i: int, j: int) -> float:

```

```

        return (
            d(i, j) if j == i + 1 else min(T(i, k) + T(k, j) + d(i, j) for k in range(i + 1, j))
        )

    return T(0, n - 1)
else:
    T = {}
    for j_i in range(1, n):
        for i in range(n - j_i):
            j = i + j_i
            T[i, j] = (
                d(i, j)
                if j == i + 1
                else min(T[i, k] + T[k, j] + d(i, j) for k in range(i + 1, j))
            )
    return T[0, n - 1]

def test_MWT() -> None:
    p1 = ((0, 0), (11, 1), (9, 8), (2, 6))
    p2 = ((0, 0), (1, 0), (1, 1), (0, 1))

    for p in (p1, p2):
        t1 = min_wt_triangulation(p)
        t2 = min_wt_triangulation(p, recursively=True)
        assert t1 == t2

    assert abs((t1 - 4) ** 2 - 2) <= 0.001

if __name__ == "__main__":
    for _ in range(1):
        test_MWT()

    print("test passed")

```

[dynamic_programming/optimal_binary_search_tree.py](#)

```

#!/usr/bin/env python3

from collections.abc import Sequence
from functools import cache as memoize
from typing import Any

def optimal_BST(p: Sequence[float], recursively: bool = False) -> float:
    # note: p is indexed from 0..(n-1) instead of 1..n

    n = len(p)

    @memoize
    def wt(i: int, k: int) -> float:
        """return p[i-1] + p[i] + ... + p[k-1]"""
        return 0 if i > k else p[k - 1] + wt(i, k - 1)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(i: int, k: int) -> float:

```

```

        return (
            0 if i > k else min(wt(i, k) + M(i, j - 1) + M(j + 1, k) for j in range(i, k + 1))
        )

    return M(1, n)
else:
    M = {}
    for k in range(n + 1):
        for i in range(k + 1, 0, -1):
            M[i, k] = (
                0
                if i > k
                else min(wt(i, k) + M[i, j - 1] + M[j + 1, k] for j in range(i, k + 1))
            )
    return M[1, n]

def test_BST(n: int = 10) -> None:
    import random

    p = [random.randrange(n) for _ in range(n)]
    t1 = optimal_BST(p)
    t2 = optimal_BST(p, recursively=True)
    assert t1 == t2

if __name__ == "__main__":
    for _ in range(40):
        test_BST()

    print("test passed")

```

dynamic_programming/Bellman_Ford.py

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf
from typing import Any

from graph_library import in_neighbor_lists

def negative_cycles[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    d: Mapping[Vertex, float],
) -> bool:
    return any(d[w] > d[u] + weights[u, w] for (u, w) in E if d[u] != inf)

def check_negative_cycles[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    d: dict[Vertex, float],
) -> dict[Vertex, float] | None:
    return None if negative_cycles(V, E, weights, d) else d

```

```

def shortest_paths[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
    recursively: bool = False,
) -> dict[Vertex, float] | None:
    n = len(V)
    in_neighbors = in_neighbor_lists(V, E)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(w: Vertex, i: int) -> float:
            return (
                0
                if i == 0 and w == s
                else inf
                if i == 0 and w != s
                else min((M(u, i - 1) + weights[u, w] for u in in_neighbors[w]), default=inf)
            )

        d = {w: min(M(w, i) for i in range(n)) for w in V}
        return check_negative_cycles(V, E, weights, d)

    else:
        M = {}
        for i in range(n):
            for w in V:
                M[w, i] = (
                    0
                    if i == 0 and w == s
                    else inf
                    if i == 0 and w != s
                    else min(
                        (M[u, i - 1] + weights[u, w] for u in in_neighbors[w]),
                        default=inf,
                    )
                )

        d = {w: min(M[w, i] for i in range(n)) for w in V}
        return check_negative_cycles(V, E, weights, d)

def Bellman_Ford[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    s: Vertex,
) -> dict[Vertex, float] | None:
    n = len(V)
    in_neighbors = in_neighbor_lists(V, E)

    d: dict[Vertex, float] = {v: inf for v in V}
    d[s] = 0

    for _ in range(n):
        for w in V:
            d[w] = min(d[w], min((d[u] + weights[u, w] for u in in_neighbors[w]), default=inf))

```

```

        return check_negative_cycles(V, E, weights, d)

def check_shortest_path_distances(Vertex: Hashable)(
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    source: Vertex,
    d: Mapping[Vertex, float],
) -> bool:
    in_neighbors = in_neighbor_lists(V, E)

    return d[source] == 0 and all(
        d[w] == min((d[u] + weights[(u, w)] for u in in_neighbors[w]), default=inf)
        for w in V
        if w != source
    )

def test_shortest_paths(n: int, m: int, m_neg: int) -> int:
    import random

    from graph_library import random_weighted_digraph

    V, E, weights = random_weighted_digraph(n, m)
    for e in random.sample(E, m_neg):
        weights[e] *= -1
    source = 0

    d1 = Bellman_Ford(V, E, weights, source)
    d2 = shortest_paths(V, E, weights, source)
    d3 = shortest_paths(V, E, weights, source, recursively=True)

    assert d1 == d2 == d3, (d1, d2, d3)

    if d1:
        assert check_shortest_path_distances(V, E, weights, source, d1)
        return 1
    else:
        return 0

if __name__ == "__main__":
    n = 200
    m = int(n**1.25)
    T = 10
    count = 0
    for t in range(T):
        count += test_shortest_paths(n, m, int(0.15 * n * t / T))

    print("test passed, with", count, "of", T, "having no negative cycle")

```

dynamic_programming/Floyd_Warshall.py

```

#!/usr/bin/env python3

from collections.abc import Collection, Hashable, Mapping
from functools import cache as memoize
from math import inf

```

```

from typing import Any

def Floyd_Warshall[Vertex: Hashable](
    V: Collection[Vertex],
    E: Collection[tuple[Vertex, Vertex]],
    weights: Mapping[tuple[Vertex, Vertex], float],
    recursively: bool = False,
) -> dict[tuple[Vertex, Vertex], float]:
    n = len(V)
    V = list(V)
    E = set(E)

    M: Any # a memoized function if recursively, else a table (dict)
    if recursively:

        @memoize
        def M(u: Vertex, w: Vertex, i: int) -> float:
            return (
                0
                if u == w
                else weights[u, w]
                if i == 0 and (u, w) in E
                else inf
                if i == 0
                else min(M(u, w, i - 1), M(u, V[i - 1], i - 1) + M(V[i - 1], w, i - 1))
            )

        return {(u, w): M(u, w, n) for u in V for w in V}

    else:
        M = {}

        for i in range(n + 1):
            for u in V:
                for w in V:
                    M[u, w, i] = (
                        0
                        if u == w
                        else weights[u, w]
                        if i == 0 and (u, w) in E
                        else inf
                        if i == 0
                        else min(M[u, w, i - 1], M[u, V[i - 1], i - 1] + M[V[i - 1], w, i - 1])
                    )
        return {(u, w): M[u, w, n] for u in V for w in V}

def test_FW(n: int = 10, m: int | None = None) -> None:
    from graph_library import random_weighted_digraph

    V, E, weights = random_weighted_digraph(n, m)
    d1 = Floyd_Warshall(V, E, weights)
    d2 = Floyd_Warshall(V, E, weights, recursively=True)
    assert d1 == d2

if __name__ == "__main__":
    for _ in range(20):
        test_FW(50)

```

```
print("test passed")
```