CS331: Algorithms and Complexity
Part III: Dynamic Programming

Kevin Tian

1 Introduction

In these notes, we introduce our second main paradigm in designing algorithms: *dynamic programming* (DP).[1] DP is a variant of recursion that uses a technique called *memoization*[2] to reduce runtimes. To motivate it, observe that most recursive algorithms are implicitly written in a top-down fashion, e.g., RecursiveAlgo (Algorithm 2, Part II) on large inputs calls RecursiveAlgo on smaller inputs multiple times. However, correctness of recursive algorithms is actually established bottom-up; we first make sure that the base case is solved correctly, and then repeatedly obtain correctness of larger calls using our solutions on smaller inputs. Notably, multiple larger calls can depend on correctness of the same smaller call. In such cases, it is less wasteful to remember which subproblems we have already solved, and reuse them when needed.

Memoization makes this observation algorithmic: whenever we have solved a recursive subproblem, we explicitly store the solution so we never have to compute it again. As we will see, this simple strategy can lead to drastic runtime savings for many problems, that can require solving subproblems exponentially many times if implemented less carefully. As an example, consider the $\mathsf{Fib}(n)$ problem from Section 5.3, Part II. Algorithm 1 gives a naïve implementation.

Algorithm 1: $\mathsf{FibNaive}(n)$

```
1 if n ≤ 1 then
2 |   return n + 1
3 end
4 return FibNaive(n − 2) + FibNaive(n − 1)
```

Unfortunately, while it outputs the correct solution, the top-down recursion used by FibNaive takes exponential time to execute. This is because each recursive call in Line 4 blows up the number of calls to $\mathsf{FibNaive}(1)$ or $\mathsf{FibNaive}(0)$ by a constant factor. To see this, we can prove inductively that the number of calls to $\mathsf{FibNaive}(1)$ or $\mathsf{FibNaive}(0)$ used by $\mathsf{FibNaive}(n)$ is F_{n-1}, following Eq. (24), Part II. This number grows exponentially in n by Lemma 14, Part I.

However, a moment's thought should convince you that $\mathsf{Fib}(n)$ is implementable in polynomial time using a smarter recursion strategy. There are only n distinct subproblems we ever need to solve in the computation path for returning $\mathsf{Fib}(n)$: namely, $\mathsf{Fib}(i)$ for all $0 \leq i \leq n-1$. Therefore, rather than starting over from scratch each time, we should implement our recursion from the bottom-up, and remember any earlier solutions. This is carried out in Algorithm 2. As discussed in Section 5.3, Part II, Algorithm 2 only requires $O(n^2)$ time under standard arithmetic models, further improvable via matrix multiplication. This is an exponential speedup of Algorithm 1.

We give many example applications of DP in these notes. In each case, to design a DP algorithm, we follow a very similar template to how we came up with Algorithm 2.

1. First, come up with any recursive algorithm that solves the problem correctly.

[1] This term was coined by Richard Bellman when working at RAND Corporation under the Air Force. Bellman supposedly chose this name in an attempt to convince his supervisors that he was not doing math at his job.

[2] To my knowledge, this word (a synonym of the confusingly similar "memorization") is only used in DP contexts. It is attributed to Donald Michie, and signifies turning subproblem solutions into "memos" for future use.

Algorithm 2: FibBasic(n)

```
L ← Array.Init(n + 1)        // L[i] will later store the (i − 1)st Fibonacci number, for all i ∈ [n + 1].
L[1] ← 1
L[2] ← 2
for 3 ≤ i ≤ n + 1 do
    L[i] ← L[i − 2] + L[i − 1]
end
return L[n + 1]
```

2. Next, examine the structure of the recursive algorithm when working from the bottom-up. Can you reuse subproblem solutions to shortcut future computations? If so, memoize them.
3. Finally, look for ways to further shortcut the recursion. Is there a more concise way to define the subproblems you care about, that lessens the number of recursive calls?

This framework seems very straightforward, but as we will see, there can be significant ingenuity involved in applying it. The two main degrees of freedom we have when designing a DP solution are: how do we define the subproblems we care about (the basic unit of memoization), and in what order do we solve them? It can be surprisingly tricky to answer these questions in a way that yields the fastest algorithm. In these notes, we focus on giving examples of applications that are representative of common strategies for solving DP problems as efficiently as possible.

2 Arrays

As an introduction, we present three simple but very common DP problems on arrays. The first two problems admit straightforward polynomial-time solutions that can be sped up significantly via memoization. For the last, it may not even be clear at first glance that the problem is solvable in polynomial time; we will see how DP can be applied to design a solution.

2.1 Largest jump

In the *largest jump* problem, we are given as input L, an Array of n numbers in $\mathbb{R}$. The goal is to output the maximum possible value of $L[j] - L[i]$, where indices (i, j) satisfy $1 \leq i \leq j \leq n$. For example, the entries of L could represent stock prices on n different days, and the largest jump problem corresponds to the largest profit margin one can achieve by buying stocks on day i and selling on day j. Note that we require $j \geq i$ (e.g., you must have bought stocks to sell them), and also that the largest jump is always nonnegative (since we allow taking $j = i$).

Let us instantiate our plan from Section 1. We first observe that there is a simple $O(n^2)$ time solution: we can compute $L[j] - L[i]$ for all pairs $1 \leq i \leq j \leq n$ and remember the maximum.

Next, let us re-examine these $\binom{n}{2} + n = \Theta(n^2)$ subproblems to search for repeated structure. Many of them have a common endpoint: for example, we need to solve all the subproblems $L[j] - L[1], L[j] - L[2], \ldots, L[j] - L[j]$. The purpose of solving these subproblems can be summarized as: suppose the largest jump ended on index $j \in [n]$. Then, we need to find the smallest entry before $L[j]$, $\min_{i \in [j]} L[i]$. Solving this problem tells us the starting index of the largest jump.

We thus have a new set of $O(n)$ subproblems describing the entire largest jump problem. Let S be an Array of n numbers in $\mathbb{R}$, where

$$S[j] = \max_{i \in [j]} L[j] - L[i] = L[j] - \min_{i \in [j]} L[i] \tag{1}$$

stores the largest jump ending on index j. If we can compute S in $O(n)$ time, then we can pass over S to determine the largest jump. Moreover, to compute $S[j]$, we just need $\min_{i \in [j]} L[i]$.

Now we can describe the full algorithm. We first iterate over L and maintain a running minimum value, storing $\min_{i \in [j]} L[i]$ for all $j \in [n]$. We can then use these running minimum values in the formula (1) to compute each $S[j]$ in $O(1)$ time. The overall runtime is just $O(n)$.

Roughly speaking, the idea was to define a set of n "special" subproblems, encoded by the entry of S, out of all of the $\Theta(n^2)$ possibilities. These special subproblems have the property that if they are computed in a certain order, each can be solved in $O(1)$ time. Specifically, this was done by maintaining a running minimum $\min_{i \in [j]} L[i]$, and using the formula (1). The key observation is that $\min_{i \in [j]} L[i]$ can be recomputed in $O(1)$ time as we increment j.

Building upon this observation, there is a simpler, direct recursive formula for the entries $S[j]$:

$$\begin{aligned} S[j] &= \max_{i \in [j]} L[j] - L[i] = \max\left(0, \max_{i \in [j-1]} L[j] - L[i]\right) \\ &= \max\left(0, L[j] - L[j-1] + \left(\max_{i \in [j-1]} L[j-1] - L[i]\right)\right) \\ &= \max\left(0, L[j] - L[j-1] + S[j-1]\right). \end{aligned} \tag{2}$$

The first line treated the two cases $i = j$ and $i \in [j-1]$ separately, the second added and subtracted $L[j-1]$, and the third used the definition of $S[j-1]$. Applying (2) repeatedly gives a one pass algorithm for computing the largest jump in $O(n)$ time, stated as Algorithm 3.

Algorithm 3: LargestJump(L)

```
Input: L, an Array instance containing n := |L| numbers in ℝ
S ← Array.Init(n)
S[1] ← 0
jump ← 0
for 2 ≤ j ≤ n do
    S[j] ← max(0, L[j] − L[j − 1] + S[j − 1])
    jump ← max(jump, S[j])
end
return jump
```

To summarize, we arrived at this solution by noticing that, although the largest jump is most naturally defined as a maximum of $\binom{n}{2}+n$ numbers that all take $O(1)$ time to compute, it is also the maximum of n (potentially more complicated) numbers: the entries of S in Algorithm 3. Therefore, we can treat $S[j]$ as our new basic subproblem. Naïvely computing each $S[j] = \max_{i \in [j]} L[j] - L[i]$, defined as a maximum over j numbers, would require $O(j) = O(n)$ time. However, we observed that the recursive formula (2) casts $S[j]$ as the maximum of two numbers, rather than j numbers as before. These two numbers can be computed in $O(1)$ time if we proceed over the subproblems S in order, via memoization of the earlier solution $S[j-1]$.

Situations like this one, where there are natural "special" subproblems solvable using a previous subproblem and a small additional amount of effort, are excellent targets for DP solutions.

2.2 Largest subsequence sum

In the *largest subsequence sum* problem, we are again given as input L, an Array of n numbers in $\mathbb{R}$. The goal is to output the maximum possible value of $\sum_{k=i}^{j} L[k]$, where indices (i, j) satisfy $1 \le i \le j \le n$. This is a classic algorithms interview question, and has a fun history rooted in computer vision [Wik24b]. The naïve algorithm computes the subsequence $\sum_{k=i}^{j} L[k]$ in $O(n)$ time for each of the $\binom{n}{2} + n = O(n^2)$ index pairs (i, j), and takes $O(n^3)$ time in total.

We can obtain significantly faster algorithms using DP. A basic idea is to treat each subproblem as computing a subsequence sum $\sum_{k=i}^{j} L[k]$, and see if we can reuse solutions. Indeed,

$$\sum_{k=i}^{j} L[k] = L[j] + \sum_{k=i}^{j-1} L[k],$$

so we can compute the former in $O(1)$ time if given the latter. Moreover, all the base cases (i, i), i.e., $L[i] = \sum_{k=i}^{i} L[k]$, take $O(1)$ time to compute. There are many ways to solve subproblems so that $(i, j-1)$ occurs before (i, j), whenever $j \ge i + 1$. For example, we can first solve all base

cases, and then solve all of the pairs $(1, j)$ for $j \geq 2$ in order, all of the pairs $(2, j)$ for $j \geq 3$ in order, and so on. This already brings down the runtime to $O(n^2)$.

We present a linear-time solution attributed to Jay Kadane. The idea is essentially the same as in Section 2.1. Let S be an Array of n numbers, such that $S[j]$ is the largest subsequence sum that ends on $L[j]$. We can instead treat computation of each $S[j]$ as one of our DP subproblems, so there are n subproblems. Does knowledge of $S[j-1]$ help with computing $S[j]$?

There are two cases: a subsequence ending on $L[j]$ either does not include $L[j-1]$, or it does. There is only one subsequence with the former property: $L[j]$ itself. Moreover, every subsequence sum of the latter type is just the sum of $L[j]$ and a subsequence sum that ends on $L[j-1]$. The largest subsequence sum ending on $L[j-1]$ is $S[j-1]$ by definition, leading to the formula

$$S[j] = \max\left(L[j], S[j-1] + L[j]\right). \tag{3}$$

Therefore, we can compute each $S[j]$ in $O(1)$ time if we loop over the $j \in [n]$ incrementally. This gives an $O(n)$ time algorithm for largest subsequence sum, as claimed. Again, the key insight leading to this solution was that the largest subsequence sum can be written as the maximum of just $O(n)$ numbers, each of which can be efficiently computed via memoization.

2.3 Balanced parentheses

In the *balanced parentheses problem*, we are given as input L, an Array of n characters that are each either an '(' (an open parenthesis) or a ')' (a closed parenthesis). The goal is to determine whether the characters can be matched into $\frac{n}{2}$ pairs, such that each open parenthesis is matched to a closed parenthesis that comes after it (if n is odd, then this is always impossible). We want to design an algorithm BalancedParentheses(L) that returns **True** if such a matching is possible, and **False** if not. The following are possible inputs to BalancedParentheses when $n = 8$:

$$)(())(, \quad ((()())), \quad (()()(((.$$

The middle example should return **True**, and the first and last should return **False** (indeed, the last example does not even have a balanced number of parentheses).

At first glance, it is not even clear this problem is solvable in polynomial time. For example, the naïve strategy of "try every matching" fails here: the number of possible matchings is $\binom{n}{2} \cdot \binom{n-2}{2} \ldots \binom{4}{2} \cdot \binom{2}{2}$, which scales as $\Theta(n)^n = \exp(\Theta(n \log(n)))$.

Luckily, the balanced parentheses problem has a highly-recursive structure, so this is an apt opportunity for the DP framework. For indices $1 \leq i \leq j \leq n$, let $L[i:j]$ denote the contiguous subarray of L that starts at $L[i]$ and ends at $L[j]$. We can define a subproblem $S[i][j]$ for each of the $\binom{n}{2} + n = \Theta(n^2)$ pairs of possible $1 \leq i \leq j \leq n$ as follows:

$$S[i][j] = \text{BalancedParentheses}(L[i:j]).$$

Our goal is thus to return $S[1][n]$. How can we use memoization to recursively solve for all $S[i][j]$? In other words, what recursively computed information would make $S[i][j]$ easy to derive? One strategy is to consider what character $L[i]$ is matched to, in a balancing of $L[i:j]$.

If $L[i]$ is matched with $L[j]$, then a balancing is possible iff the remaining $L[i+1:j-1]$ can be balanced. Otherwise, $L[i]$ is matched to some character $L[k]$, for $i < k < j-1$. In this case, we must recursively balance $L[k+1:j]$, as well as $L[i:k]$. Intuitively, this case splits $S[i][j]$ into two subproblems $S[i][k]$ and $S[k+1][j]$. In summary, $S[i][j] =$ **True** if either of the following holds.

1. $L[i] =$ '(', $L[j] =$ ')', and $S[i+1][j-1] =$ **True**.
2. $S[i][k] = S[k+1][j] =$ **True** for some $i < k < j-1$.

In what order should we solve the subproblems so that the information we need is memoized, by the time we need to compute $S[i][j]$? Observe that regardless of which case we are in, we only need the solutions to subproblems on *shorter subarrays*. Thus, a reasonable order is to start by solving subproblems on length-2 subarrays, and then solve length-4 subarrays, and so on. We remark that if $j - i$ is even (so that $L[i][j]$ contains an odd number of elements), then balancing the subarray is clearly impossible. We present pseudocode implementing this solution.

Algorithm 4: BalancedParentheses(L)

```
1  Input: L, an Array instance containing n := |L| characters, each either ( or )
2  if n%2 == 1 then
3  |  return False
4  end
5  S ← two-dimensional Array with dimensions n × n    // Implementable using an Array of size n, with
     Array instances of size n as its entries.
6  ℓ ← 2                                              // Length of subarrays currently considered, initialized at 2.
7  while ℓ ≤ n do
8  |  for 1 ≤ i ≤ n − ℓ + 1 do
9  |  |  j ← i + ℓ − 1                                // This loop computes the value of S[i][j], where L[i][j] has length ℓ.
10 |  |  if ℓ = 2 then
11 |  |  |  S[i][j] ← L[i] == '(' and L[j] == ')'
12 |  |  end
13 |  |  else
14 |  |  |  S[i][j] ← L[i] == '(' and L[j] == ')' and S[i + 1][j − 1]
15 |  |  |  for i + 1 ≤ k ≤ j − 2 do
16 |  |  |  |  S[i][j] ← S[i + 1][k] and S[k + 1][j]
17 |  |  |  end
18 |  |  end
19 |  end
20 |  ℓ ← ℓ + 2
21 end
22 return S[1][n]
```

What is the runtime of Algorithm 4? It considers $O(n)$ possible lengths (values of ℓ), each containing two nested for loops (the outer loop being Lines 8 to 21, and the inner loop being Lines 15 to 17). Thus, the runtime is $O(n^3)$. Another way to arrive at this conclusion is that there are $O(n^2)$ subproblems $S[i][j]$, and it takes $O(n)$ time to solve each subproblem, even after memoization. This is because our recursion for $S[i][j]$ considers each possible character that $S[i]$ could be paired to, and there are $O(n)$ possibilities in general (e.g., once ℓ becomes large).

The balanced parentheses problem illustrates a more advanced use of the DP framework. Solving each subproblem $S[i][j]$ efficiently relies on having recursively solved multiple smaller subproblems, each corresponding to a subarray of $L[i][j]$. this is in contrast with our earlier recursions in (2) and (3), each of which only required the previous subproblem's solution.

In fact, it is possible to significantly improve upon the runtime presented here; in Part IV of the notes, we will see an $O(n)$-time solution using a greedy strategy. However, the DP-based solution is often the most straightforward for establishing a polynomial-time baseline.

3 Sets

In Section 2, we saw two applications in arrays where DP leads to a polynomial factor runtime savings over the straightforward algorithm (e.g., $O(n)$ vs. $O(n^2)$ time). We also saw an example where it was less clear how to design even a polynomial-time algorithm without DP.

In this section, we give several examples of DP applications in sets, where runtime savings are quite dramatic. In each case, the goal is to return a subset of a set of size n, that is optimal according to some criterion. The straightforward approach would try every subset as a candidate, but this quickly becomes intractable for moderate n, as there are 2^n candidate subsets.

By more carefully defining subproblems of interest and using DP, we show how to design efficient polynomial-time algorithms for these problems. In each case, the key insight is to recursively eliminate large portions of the candidate solution space (i.e., all possible subsets) by imposing additional structure on the subproblems we consider, so that there are much fewer subproblems.

3.1 Scheduling

In the *scheduling* problem, we are given as input a set L, an Array of n tuples. The i^{th} tuple is denoted $(\ell_i, r_i) \in \mathbb{R}^2$ and satisfies $\ell_i < r_i$, for all $i \in [n]$. Each tuple specifies the endpoints of a closed interval $[\ell_i, r_i] \subset \mathbb{R}$. The goal is to the maximum size $|T|$ of a subset $T \subseteq [n]$ with no overlapping intervals, i.e., for all $i, j \in T$ with $i \neq j$, we have $[\ell_i, r_i] \cap [\ell_j, r_j] = \emptyset$.

To motivate the problem, suppose you are a professor and are advising n students. The i^{th} student wants to meet from time ℓ_i to time r_i, but you can only meet with one student at a time. What is the maximum number of your students that you can meet with?

At first, it is unclear that scheduling is even solvable in polynomial time. The straightforward solution checks each subset $T \subseteq [n]$ for whether it contains overlaps, and if it does not, compares $|T|$ to a running maximum. This takes $O(2^n)$ time due to the number of subsets.

Following our plan from Section 1, let us try to define a small (polynomial-sized) set of recursive subproblems which correctly solves scheduling. Based on our development in Section 2, a reasonable attempt is to let S be a length-n Array where $S[i]$ stores the maximum size of a non-overlapping subset that includes the i^{th} interval $[\ell_i, r_i]$. However, this strategy immediately runs into two major issues. First, how do we recursively compute $S[i]$? If an optimal subset includes the i^{th} interval, all we can can conclude is that it does not contain any other intervals that overlap with $[\ell_i, r_i]$. This information is not contained in any of the other subproblems $S[j]$, and seems to require defining more subproblems, which quickly can blow up into an exponential number of subproblems. Second, there is no natural order that we should recursively solve these subproblems in.

A slight modification to our subproblem definitions addresses both issues simultaneously. Recall from Section 2 that our DP subproblems were naturally ordered: each subproblem was the optimal solution on a prefix of the array. We can employ a similar prefix-based solution here, after first sorting the input list L to induce an ordering. There is a design decision to be made: should we sort intervals in L by their left endpoint (i.e., start time) or right endpoint (i.e., end time)? The answer is clearly whichever lets us recursively solve subproblems most efficiently. As we will explain shortly, it turns out to be more useful to sort entries by their right endpoint.

We are ready to explain how to solve scheduling via DP. Suppose first that L has been sorted by right endpoint, so that $r_i \leq r_j$ for each pair of intervals with $1 \leq i < j \leq n$. This takes $O(n \log(n))$ time using, e.g., MergeSort (Algorithm 5, Part II). Moreover, suppose that for each $i \in [n]$, we have computed $P[j]$, the largest index i such that $r_i < \ell_j$ (if there are no such indices, we let $P[j] = 0$). In other words, $P[j]$ is the last interval in L that has ended before the j^{th} interval begins. We can compute P in $O(n \log(n))$ time using binary search, since for each ℓ_j, we can determine which two indices r_i, r_{i+1} it fits in between using $O(\log(n))$ time, using the sorted list of right endpoints.

We now define our DP subproblems. Let S be a length-n Array, such that for all $j \in [n]$, $S[j]$ stores the maximum size of a non-overlapping subset of $L[:j]$, which we use to denote the prefix subarray consisting of the first j intervals in L. We claim that

$$S[j] = \max\left(S[j-1], 1 + S[P[j]]\right), \tag{4}$$

for all $j \in [n]$, where $S[0] := 0$. To see why, consider the optimal scheduling solution over $L[:j]$. There are two cases, depending on whether the largest subset includes $S[j]$. If it does not contain $S[j]$, then the solution equals $S[j-1]$, i.e., the best solution in $L[:j-1]$, the first case in (4).

Otherwise, the optimal solution contains $S[j]$, and hence it cannot include any intervals that overlap with $[\ell_j, r_j]$. We should therefore drop any such intervals from consideration in this case. However, we know exactly which intervals are not dropped: they are precisely the prefix $L[:P[j]]$, i.e., the intervals with indices $1, 2, \ldots, P[j]$! This is because the intervals are sorted by their right endpoint, and we know $P[j]$ is the last interval whose right endpoint occurs before ℓ_j. This also explains why we sorted L by right endpoint values: otherwise, the set of intervals ending before $L[j]$ begins is not a prefix of L, whereas here, we recursively only need to consider solutions in $L[:P[j]]$.

In conclusion, our DP solution has three steps: sorting L by right endpoint, computing the indices P, and recursively applying the formula (4) to solve our subproblems. The answer to our original scheduling problem is contained in $S[n]$, as it is optimal for the first n intervals in L, i.e., the entire list. We provide pseudocode in Algorithm 5, assuming for simplicity L has been pre-sorted.

Algorithm 5: Scheduling(L)

1 **Input:** L, an Array instance containing $n := |L|$ tuples $\{(\ell_i, r_i)\}_{i \in [n]} \subset \mathbb{R}^2$ with $\ell_i < r_i$ for all $i \in [n]$, sorted by right endpoint so that $r_i \leq r_j$ for all $1 \leq i < j \leq n$
2 $P \leftarrow$ Array.Init(n)
3 $S \leftarrow$ Array.Init(n)
4 **for** $j \in [n]$ **do**
5 | $P[j] \leftarrow i$ where $i \in [n]$ is the largest index such that $r_i < \ell_j$, or $i = 0$ if $\ell_j \leq r_1$
6 **end**
7 **for** $j \in [n]$ **do**
8 | $S[j] \leftarrow \max(S[j-1], 1 + S[P[j]])$, where $S[0] := 0$
9 **end**
10 **return** $S[n]$

Let us now recap the runtime analysis of Algorithm 5. Pre-sorting the list L using MergeSort takes $O(n \log(n))$ time, and each loop of Lines 4 to 6 takes $O(\log(n))$ time, since we need to search for where to insert ℓ_j into the sorted list of $\{r_i\}_{i \in [n]}$, solvable via binary search. Finally, each loop of Lines 7 to 9 takes $O(1)$ time, since we have precomputed $P[j] < j$ and $S[j-1]$. Thus, the overall runtime is $O(n \log(n))$, dominated by the cost of sorting L and computing P.

This is a ridiculously large improvement to a problem which originally appeared to take exponential time. In hindsight, the way we obtained this improvement is by severely restricting the kinds of "special" subsets we are interested in: the largest size non-overlapping subsets restricted to each of the prefixes $L[:j]$. This seems like a strong ask, but fortunately the largest prefix-restricted sizes can be recursively computed using (4). This formula, which encodes a decision on whether to include $L[j]$ in the optimal subset, luckily also results in a natural efficient algorithm.

Weighted scheduling. Algorithm 5 extends straightforwardly to solve the *weighted scheduling* problem, where each entry $i \in [n]$ of the input L contains a weight w_i in addition to two endpoints of an interval $[\ell_i, r_i]$. The goal in weighted scheduling is to output a subset T of L with maximum weight $w(T) := \sum_{i \in T} w_i$, such that no two intervals in T overlap. For example, weighted scheduling captures the problem of maximizing the total length of all the intervals in our subset, by defining weights $w_i = r_i - \ell_i$ for all $i \in [n]$. Moreover, weighted scheduling generalizes the standard scheduling problem, which corresponds to weights $w_i = 1$ for all $i \in [n]$.

The idea is essentially identical to the unweighted case, where we first assume that L has been sorted by right endpoint. Next, we let each subproblem $S[j]$ for $j \in [n]$ correspond to the maximum weight of a non-overlapping subset of the first j intervals in L. Again, to solve subproblem $S[j]$, we take the better of the case where the j^{th} interval is not included (so the maximum weight achievable is $S[j-1]$), and the case where it is included (so the maximum weight achievable is $w_j + S[P[j]]$). This gives rise to the following recurrence, generalizing (4):

$$S[j] = \max(S[j-1], w_j + S[P[j]]).$$

Replacing Line 8 of Algorithm 5 with the above solves weighted scheduling in $O(n \log(n))$ time.

Recovering an optimal solution. Our algorithm so far has only produced the value of the optimal solution (i.e., the size or total weighted value of T), rather than the solution itself (i.e., the set T). There is a simple way to augment Algorithm 5 to reconstruct the optimal solution.

First, observe that we can store in each $S[j]$, in addition to the value of the optimal prefix solution, enough information to remember which subproblem we built upon to achieve this optimum. Specifically, we can let each $S[j]$ store two fields: (val, case), where $S[j]$.val contains the optimal value as before. Moreover, $S[j]$.case $= 1$ or 2, depending on whether $S[j] = S[j-1]$ (case 1) or $S[j] = 1 + S[P[j]]$ (case 2). Using this extra information, we can work backwards from $S[n]$ to recover the optimal set, by either including $S[j]$ or not (if $S[j]$.case $= 2$ or 1, respectively), and continuing to either $S[j-1]$ or $S[P[j]]$ to continue building the solution.

In fact, we do not even need to store this extra information in $S[j]$.case, because we can check which case we are in "on the fly" by recomputing the two values in (4) and checking which equals the value

in $S[j]$. We provide pseudocode which uses this simpler reconstruction strategy in Algorithm 6, where the input consists of memoized subproblem solutions P and S from Algorithm 5.

Algorithm 6: RecoverSchedule(P, S)

```
Input: P, S, the contents of variables initialized on Lines 2 and 3 after Algorithm 5 returns
T ← ∅
j ← n
while j ≠ 0 do
    if S[j] == S[j − 1] then
        j ← j − 1                  // New goal: recover the subset leading to the subproblem solution S[j − 1].
    end
    else
        T ← T ∪ {i}
        j ← P[j]                   // New goal: recover the subset leading to the subproblem solution S[P[j]].
    end
end
return T
```

3.2 Longest increasing subsequence

In the *longest increasing subsequence* problem, we are given as input L, an Array of n numbers in $\mathbb{R}$. The goal is to output the maxmium size of a subset $T \subseteq [n]$, such that for all $i, j \in T$ with $i < j$, we have $L[i] \leq L[j]$. In other words, we want to find the maximum length of a subsequence of L that is in sorted order, when L itself may be unsorted. This fundamental problem is commonly used as a subroutine in other applications, see e.g., discussion in [Wik24a].

It is not obvious that this problem is solvable in polynomial time, as the straightforward approach is to try all subsets of $[n]$. However, again we will be able to use DP to dramatically speed up our solution. Analogously to the scheduling problem, it is natural to define a set of n subproblems on prefixes. Specifically, let S be a length-n Array, such that for all $j \in [n]$, $S[j]$ stores the largest length of an increasing subsequence that ends on (and includes) $L[j]$. Once we have computed S, we can take a single pass and return $\max_{j \in [n]} S[j]$ in $O(n)$ time.

However, computing $S[j]$ involves its own recursive choice of which subproblem to skip to. That is, how should we continue building the longest increasing subsequence once $L[j]$ is included? In each of (2), (3), and (4), this recursion only involved two cases. In scheduling, for example, the optimal solution including the j^{th} interval must necessarily exclude all of the overlapping intervals $P[j]+1, P[j]+2, \ldots, j-1$, so we can skip from $S[j]$ directly to $S[P[j]]$. In the longest increasing subsequence problem, it is less clear which subproblem we should skip to after including $L[j]$, as any of the $L[i] \leq L[j]$ with $i < j$ could potentially be used to continue growing the subsequence, with different tradeoffs. Taking a larger value of the index i may be a bad idea if $L[i]$ itself is small, whereas taking a smaller i would exclude more entries from consideration.

Fortunately, if we solve subproblems $S[j]$ sequentially (i.e., one at a time while incrementing j), we can simply try all of the possible continuations. Indeed, we claim:

$$S[j] = 1 + \max_{\substack{i \in [j-1] \\ L[i] \leq L[j]}} S[i], \tag{5}$$

for all $j \in [n]$ (the max evaluates to 0 if $L[j]$ is minimal in $L[:j]$). To see this, the longest increasing subsequence ending on $S[j]$ must have second-to-last entry $L[i]$, for some $i < j$ satisfying $L[i] \leq L[j]$. Moreover, if we know that the second-to-last entry is $L[i]$, then the longest subsequence we can make just appends $L[j]$ to the longest subsequence ending on $L[i]$. Therefore, $S[j] = 1 + S[i]$ for whichever $S[i]$ is largest among the $L[i]$ that could be included, exactly as (5) computes.

When computing $S[j]$ sequentially, we have already memoized all of the $S[i]$ for all $i \in [j-1]$. Thus, evaluating the formula (5) takes $O(j) = O(n)$ time, so looping over all $j \in [n]$ requires $O(n^2)$ time total. Our algorithm thus far only returns a length rather than a subsequence, but a strategy similar to Algorithm 6 recovers the longest subsequence once we have computed S.

Using multiple intermediate subproblems to solve a single new subproblem is reminiscent of our balanced parentheses recursion in Section 2.3. As before, the difficulty is that there are multiple potential "branches" that our recursion could take in computing an optimal solution. The key is that the subproblems have a natural recursive order. Here, the order was given by nested prefixes, and in Section 2.3, the order was given by lengths of subarrays. This ordering lets us make sure we have memoized all of the subproblems that we needed before attacking the next one.

Schensted's algorithm. In fact, the longest increasing subsequence problem can be solved in $O(n \log(n))$ time, by implementing (5) more cleverly. We give an algorithm due to [Sch61] which proceeds in n iterations, indexed by an iteration count $j \in [n]$. We maintain a subproblem solution array S, with the following invariant: after iteration $j \in [n]$ is done, $S[k]$ contains the smallest possible last value in a length-k increasing subsequence of $L[:j]$. This guarantee holds for all $k \in [K]$, where K is the length of the longest increasing subsequence of $L[:j]$.

We now give some intuition for how one could come up with this definition. When evaluating (5), we consider $j-1$ potential increasing subsequences (the longest ones ending on each $L[i]$), but the formula (5) does not need to know very much about these subsequences. In particular, it only cares about their lengths and last entry values. This fact motivates our new subproblem: the claim is that it suffices to keep the last value as small as possible for a subsequence of a given length. As we will see, this dramatically speeds up each iteration of the algorithm.

There is a helpful way to visualize this process as maintaining K distinct stacks, each containing the current $S[k]$ values at their heads. First, $K \leftarrow 1$, and we place $L[1]$ on the first stack. Next, in the j^{th} step for $j \geq 2$, we iterate over the stacks and check if it is possible to create a length-k increasing subsequence of $L[:j]$ with a smaller last value. If so, we place the new smallest value on top of the stack, and replace $S[k]$. If K increments (i.e., it is newly possible to create a length-$K+1$ increasing subsequence), we create a new stack with the last value in the subsequence. Observe that S is always in sorted order, because given a length-k increasing subsequence of $L[j]$, we can remove its k^{th} term to create a length-$(k-1)$ increasing subsequence with a smaller last value.

It turns out that this stack-growing process admits a very simple recursion, though this is not obvious. We claim that in step j, all that happens is we place $L[j]$ on top of stack $k+1$, where k is the index satisfying $S[k] \leq L[j] < S[k+1]$. If $L[j] < S[1]$, we place it on the first stack, and if it satisfies $L[j] \geq S[K]$, we increment K and create a new stack with $S[K+1] = L[j]$ at the top. We can implement this stack-growing step in $O(\log(n))$ time by binary searching for the index k with $S[k] \leq L[j] < S[k+1]$, since we argued earlier that S is always sorted. Therefore, if we prove this stack-growing process correctly preserves the definition of S, we can solve longest increasing subsequence in $O(n \log(n))$ time, by outputting the final value of K.

We now prove correctness. Consider what could change in step j: i.e., after allowing $L[j]$ to be included, which k could possibly have subsequences of that length with smaller last entries? One simple observation is that any subsequence of $L[:j]$ that includes $L[j]$ ends with $L[j]$, so the only thing that can happen is that $L[j]$ becomes the new head of some stacks. Our claim is essentially that this uniquely happens to the $(k+1)^{\text{th}}$ stack, where $S[k] \leq L[j] < S[k+1]$.

We can prove this in three cases. First, for any earlier stack $i \leq k$, we cannot replace its head with $L[j]$, since $L[j] \geq S[k] \geq S[i]$, i.e., $L[j]$ is not smaller than the previous head $S[i]$. Second, for any later stack $i > k+1$, it cannot be the case that there is a length-i subsequence of $L[:j]$ ends on $L[j]$. This would imply there is a length-$(i-1)$ subsequence of $L[:j-1]$ with smaller last term than $L[j]$, but $S[i-1] \geq S[k+1] > L[j]$. Finally, we can create a smaller last term in a length-$(k+1)$ subsequence by taking the length-k subsequence ending in $S[k]$, and appending $L[j]$ at the end. Thus, we should replace the head of $S[k+1]$ only, and this completes the proof.

3.3 Subset sum

In this final subsection on sets, we introduce a family of related problems. The simplest variant in this family is the *subset sum* problem. In this problem, we are given as input L, an Array of n natural numbers, and a target value $V \in \mathbb{N}$. The goal is to determine whether or not there is a subset $T \subseteq [n]$ whose corresponding entries in L sum to V, i.e., has

$$V = \sum_{i \in T} L[i].$$

Again, there is a straightforward exponential-time solution that computes the sum of each of the 2^n subsets of L. Our goal is to improve this result substantially via DP.

In Sections 3.1 and 3.2, we overcame the potentially exponential hardness of two subset selection problems by defining subproblems with a natural ordering. For example, in scheduling, we first induced an ordering by sorting the right endpoints. We then defined our subproblems on prefixes. In the longest increasing subsequence problem, we again let subproblems correspond to prefixes.

For subset sum, such a straightforward approach fails. For example, we could try letting $S[j]$ compute the value of subset sum for the prefix $L[:j]$. The trouble is that earlier subproblems appear to yield little information for later subproblems. Indeed, if all earlier prefixes were unable to produce a subset summing to V, that says essentially nothing about the current prefix.

However, it turns out that there is still a way of using DP to attack subset sum, if we extend our subproblems along another dimension: the *value* of a subset $T \subseteq [n]$, i.e., the actual sum $\sum_{i \in T} L[i]$. This is intuitive, because to understand whether a target sum v is achievable after committing to including some entry $L[j]$, we can reduce to a subproblem with target sum $v - L[j]$. Therefore, it seems natural to index subproblems on the value of a subset sum.

We can now explain our DP solution. We define nV different subproblems $S[j][v]$, indexed by $j \in [n]$ and $v \in [V]$. Thus, there are two axes, thought of as a "prefix index" j and a "target value" v. We let $S[j][v]$ store either **True** (if there is a subset of the prefix $L[:j]$ summing to v) or **False** (if no such subset of $L[:j]$ exists). Note that $S[n][V]$ contains the answer to subset sum.

We claim that

$$S[j][v] = S[j-1][v - L[j]] \textbf{ or } S[j-1][v]. \tag{6}$$

Here, we let $S[j-1][v - L[j]] =$ **False** by default if $v - L[j]$ is not a valid target, i.e., if $L[j] > v$, and we let $S[j][0] =$ **True** for all $j \in [n]$, since we can always hit a target of 0 by taking nothing.

Let us unpack the formula (6). By our subproblem definitions, the left-hand side asks to compute whether it is possible for a subset of $L[:j]$ to sum to v. Suppose there were such a subset of $L[:j]$. There are two cases: either the subset contains the last entry $L[j]$, or it does not.

If it contains $L[j]$, then the other entries in the subset must sum to $v - L[j]$. These other entries all belong to the prefix $L[:j-1]$. Therefore, in this case, $S[j-1][v - L[j]] =$ **True**.

In the other case, $L[j]$ is not included in the subset, so the entire subset belongs to the prefix $L[:j-1]$ and sums to v. Therefore, in this case, $S[j-1][v] =$ **True**.

We have enumerated the only two possibilities for how $S[j][v]$ can be **True** and the reader can verify that the formula (6) indeed implements this recursion. To finish our DP solution, we also need to specify an order to solve the subproblems (6) in, and analyze its runtime. Luckily, the formula (6) suggests a natural order: proceed one index j at a time.

It is helpful to visualize our subproblems as being memoized in a two-dimensional array, where the horizontal axis is the value $v \in [V]$ and the vertical axis is the index $j \in [n]$. Computation of the j^{th} row using (6) only requires having memoized the values of the $(j-1)^{\text{th}}$ row. Thus, we can proceed row by row, filling in each row from left to right, as described in Algorithm 7.

Lines 3 to 10 of Algorithm 7 implement the base case of our DP, i.e., filling out the first row of S. The only possible target value v attainable using subsets of $L[:1]$ is $v = L[1]$, since this is the only entry of the prefix. Thus, we set $S[1][L[1]] \leftarrow$ **True** if $L[1] \in [V]$, and we set all other entries of the first row in S to **False**. Finally, Lines 11 to 23 fill out the rest of S one row at a time. Each row only relies on the previous row, which has been memoized. It is clear that Algorithm 7 uses $O(1)$ time to compute each of the nV entries in S, so the overall runtime is $O(nV)$.

This is good if V is moderately small, e.g., if $V \approx n$ then the runtime of Algorithm 7 is $\approx n^2$. However, this algorithm is not well-suited to subset sum instances with large V. This is for good reason: our difficulty grappling with $V \gg n$ is believed to be inherent to all efficient algorithms. We will revisit this point in our complexity theory unit, in Part VIII of the notes.

0-1 **knapsack.** There are many variants of subset sum, admitting corresponding variants of the solution template in Algorithm 7. Most only involve small modifications to the basic solution.

Algorithm 7: SubsetSum(L, V)

```
Input: L, an Array instance containing n := |L| numbers in ℕ, target value V ∈ ℕ
S ← two-dimensional Array with dimensions n × V
                // Implementable using an Array of size n, with Array instances of size V as its entries.
for v ∈ [V] do
    if v == L[1] then
        S[1][v] ← True
    end
    else
        S[1][v] ← False
    end
end
for 2 ≤ j ≤ n do
    for v ∈ [V] do
        if L[j] < v then
            S[j][v] ← S[j − 1][v − L[j]] or S[j − 1][v]
        end
        else if L[j] = v then
            S[j][v] ← True
        end
        else
            S[j][v] ← S[j − 1][v]
        end
    end
end
return S[n][V]
```

One of the most famous variants is the 0-1 *knapsack* problem. In this problem, we are given as input W, an Array of n natural numbers, and V, an Array of n positive real numbers. These arrays encode the weights and values of n items. We are also given a weight budget $B \in \mathbb{N}$. The goal is to return the maximum possible total value $\sum_{i \in T} V[i]$ of a subset $T \subseteq [n]$ of items, subject to the total weight $\sum_{i \in T} W[i]$ staying within the budget B. That is, we want to determine

$$\max_{\substack{T \subseteq [n] \\ \sum_{i \in T} W[i] \leq B}} \sum_{i \in T} V[i]. \tag{7}$$

Intuitively, the 0-1 knapsack problem models budget optimization problems, where taking items incurs a cost against a budget limit B. Perhaps the least natural part of the problem definition is the requirement that the budget B and all the weights $W[i]$ are integers. Again, this is believed to be inherent to efficient algorithms for 0-1 knapsack, to be discussed later in Part VIII.

There are three main differences between 0-1 knapsack and subset sum. First, rather than seeking a subset whose total weight exactly *equals* a value, 0-1 knapsack considers subsets whose total weight is *at most* a value. Second, in addition to having a weight $W[i]$, every item in 0-1 knapsack also has a value $V[i]$. Finally, rather than merely seeking existence of such a subset (i.e., a **True**-**False** question), 0-1 knapsack asks for the value of the optimal subset.

Nonetheless, we will be able to solve 0-1 knapsack in $O(nB)$ time using a slight modification of our DP strategy for subset sum. We define nB different subproblems $S[j][b]$, indexed by $j \in [n]$ and $b \in [B]$. We let $S[j][b]$ store the maximum possible value of a subset of $L[:j]$, subject to staying within a weight budget of b. That is, our goal is to compute all

$$S[j][b] := \max_{\substack{T \subseteq [j] \\ \sum_{i \in T} W[i] \leq b}} \sum_{i \in T} V[i].$$

Comparing to (7), we see that $S[n][B]$ solves our original problem. Further, very similarly to (6),

$$S[j][b] = \max\left(S[j-1][b - W[j]] + V[j], S[j-1][b]\right),$$

where again we only include the first term if $W[j] \leq b$, and we treat $S[j][0] = 0$ for all $j \in [n]$. This is because a subset of $L[:j]$ either takes the j^{th} item or it does not. If we take the j^{th} item, we have a new budget constraint of $b - W[j]$, but we have gained $V[j]$ in value. Otherwise, we keep our budget constraint of b but are restricted to the first $j-1$ items. Finally, we can solve these subproblems by iterating over $j \in [n]$ one row at a time and applying the above formula, as in Algorithm 7. This takes $O(1)$ time per subproblem via memoization, or $O(nB)$ time total.

Unbounded knapsack. The *unbounded knapsack* problem is a variant of 0-1 knapsack where we can take items multiple times. In other words, rather than restricting ourselves to subsets $T \subseteq [n]$ (which contain either 0 or 1 copies of each item), we modify (7) to read

$$\max_{\substack{\mathbf{c} \in \mathbb{Z}^n_{\geq 0} \\ \sum_{i \in [n]} \mathbf{c}_i W[i] \leq B}} \sum_{i \in [n]} \mathbf{c}_i V[i].$$

To explain this expression, $\mathbf{c}_i \in \mathbb{Z}_{\geq 0}$ represents the count of each item i. We still wish to stay within the budget constraint, so $\sum_{i \in [n]} \mathbf{c}_i W[i] \leq B$, and we are rewarded based on the total value of taken items, $\sum_{i \in [n]} \mathbf{c}_i V[i]$. We refer to the sets of items we are allowed to take as multisets.

It no longer makes sense to define subproblems based on prefixes (i.e., indexing subproblems by $j \in [n]$), because the unbounded knapsack problem lacks the recursive structure of subset selection problems: once we take item j, we could still take it again. However, there is a simple fix: we index our DP subproblems on the budget constraint $b \in [B]$ only. Namely, for all $b \in [B]$, let

$$S[b] := \max_{\substack{\mathbf{c} \in \mathbb{Z}^n_{\geq 0} \\ \sum_{i \in [n]} \mathbf{c}_i W[i] \leq b}} \sum_{i \in [n]} \mathbf{c}_i V[i].$$

The observation driving the recurrence for unbounded knapsack subproblems is: a multiset either contains no elements, or it contains at least one copy of item $i \in [n]$, for some i with $W[i] \leq b$. Therefore, defining $S[0] = 0$,

$$S[b] = \max \left(0, \max_{\substack{i \in [n] \\ W[i] \leq b}} S[b - W[i]] + V[i] \right).$$

By solving the subproblems $S[b]$ for $b \in [B]$ in incremental order, each computation of $S[b]$ takes $O(n)$ time using the above formula via memoization. Thus, the total runtime is again $O(nB)$.

4 Strings

In this section, we give DP-based solutions to two common problems on strings. Strings are arrays of n characters from a universe Ω. For example, the universe Ω could be the English alphabet $\{\text{'a'}, \text{'b'}, \text{'c'}, \ldots, \text{'z'}\}$, or the set of characters commonly found on computer keyboards. Unlike the problems in Section 2, which focused on real numbers or restricted character types, string problems deal with potentially arbitrary sets of characters. Nonetheless, several techniques we have developed so far (in combination with new tricks) will prove to be very useful.

Throughout the rest of this section, we fix an arbitrary universe Ω, and a length-n *string* S is an Array of n entries from Ω. We refer to the i^{th} entry of S as $S[i]$ for all $i \in [n]$. A *subsequence* of a string S is any string T that can be formed by deleting entries from S, and concatenating the remaining entries preserving the order. A *substring* of a string S is any subsequence T whose characters originally appeared in consecutive order in S. For example, if $S = \{1, 2, 3, 4, 5\}$, then $T = \{1, 3, 5\}$ is a subsequence of S, $T = \{1, 2, 3\}$ is a substring of S, and $T = \{1, 4, 3\}$ is neither.

4.1 Longest common subsequence

One of the most fundamental problems on strings is the *longest common subsequence* (LCS) problem. In this problem, we are given as input two strings X and Y, with lengths m and n respectively. The goal is to determine the length of the LCS of X and Y. That is, we want to find the longest a string Z could possibly be, where Z is a subsequence of both X and Y.

This problem has various applications. For example, in biology, X and Y could be two DNA sequences, and we are interested in finding long common subsequences as evidence of proximity of X and Y in the evolution tree. More generally, any time a problem involves strings, the length of the LCS is a reasonable metric (i.e., notion of similarity) for comparing pairs of strings.

Once again, the naïve solution to LCS (enumerate all pairs of substrings and compare them) runs in exponential time. A natural choice of "special subproblems" for LCS is the set of LCS solutions on pairs of a prefix of X and a prefix of Y. Let us define $S[i][j]$ to be the length of the LCS of $X[:i]$ and $Y[:j]$, i.e., the first i characters in X and first j in Y, for all $i \in [m]$, $j \in [n]$. We claim

$$S[i][j] = \max\left(S[i][j-1], S[i-1][j], S[i-1][j-1] + \mathbb{I}_{X[i]=Y[j]}\right) \tag{8}$$

where $S[i][j] := 0$ if $i = 0$ or $j = 0$. Here, $\mathbb{I}_{\mathcal{E}}$ is the 0-1 indicator corresponding to an event $\mathcal{E}$, so

$$\mathbb{I}_{X[i]=Y[j]} = \begin{cases} 1 & X[i] = Y[j] \\ 0 & X[i] \neq Y[j] \end{cases}.$$

In other words, the option $S[i-1][j-1]+1$ is only included in (8) if $X[i] = Y[j]$. Otherwise, we can limit (8) to the first two options, as any substring of $X[:i-1]$ and $Y[:j-1]$ is a substring of $X[:i]$ and $Y[:j-1]$, so $S[i-1][j-1] \leq S[i][j-1]$ and thus the third option in (8) can be excluded.

Let us now prove the formula (8) is correct. Suppose first that $X[i] \neq Y[j]$. Then, at least one of $X[i]$ or $Y[j]$ does not belong to the LCS of $X[:i]$ and $Y[:j]$, since they would both be the last element in the subsequence, but they are unequal. Thus, the LCS is also the LCS of the pair $(X[:i-1], Y[:j])$, if we do not include $X[i]$, or the LCS of $(X[:i], Y[:j-1])$, if we do not include $Y[j]$. This implies $S[i][j] = \max(S[i][j-1], S[i-1][j])$ as claimed in (8).

Otherwise, suppose that $X[i] = Y[j]$. There is now a third possibility: the LCS includes both $X[i]$ and $Y[j]$. In this case, the LCS is just the LCS of $X[:i-1]$ and $Y[:j-1]$, with $X[i] = Y[j]$ appended to the end. This gives the third option in (8) as claimed.

Finally, we need to specify an order in which to evaluate the DP subproblem values (8). We will have an outer for loop that increments through $i \in [m]$, and an inner for loop that increments through $j \in [n]$, solving $S[i][j]$ iteratively. When $i = 1$ or $j = 1$, the formula (8) can directly be evaluated in $O(1)$ time under our convention handling $i = 0$ or $j = 0$. Otherwise, it is straightforward to check that all of the subproblem solutions with index pairs $(i, j-1)$, $(i-1, j)$, and $(i-1, j-1)$ have been memoized by the time we reach $S[i][j]$. Thus, evaluating (8) takes $O(1)$ time per index pair (i, j) with $i \in [m]$, $j \in [n]$, and the algorithm overall takes $O(mn)$ time.

Multiple strings. It is possible to generalize the LCS problem to more than two strings. For example, to determine the LCS of three strings W, X, Y, with lengths ℓ, m, n respectively, we can simply define subproblems $S[i][j][k]$ to be the length of the LCS of $W[:i]$, $X[:j]$, and $Y[:k]$ for all triplets $i \in [\ell]$, $j \in [m]$, $k \in [n]$. These subproblems satisfy the recursion

$$S[i][j][k] = \max\left(S[i][j][k-1], S[i][j-1][k], S[i-1][j][k], S[i-1][k-1][j-1] + \mathbb{I}_{W[i]=X[j]=Y[k]}\right),$$

whose validity is proven analogously to (8). This formula can be evaluated in $O(1)$ time if we memoize subproblems in three nested for loops over $i \in [\ell]$, $j \in [m]$, and $k \in [n]$, so the overall algorithm takes $O(\ell mn)$ time. The generalization to even more strings is straightforward.

Edit distance. The LCS is a close variant of computing arguably the most popular metric on strings: the *edit distance*. The edit distance between strings X and Y has a very intuitive definition: it is the fewest number of operations needed to transform X into Y, where the set of allowable operations is: delete a character, insert a character, or substitute a character for any other. For example, the edit distance between "kitten" and "sitting" is 3, witnessed by the substitutions 'k' ← 's', 'e' ← 'i', and the insertion of a 'g' at the end of the string.

It turns out that if we are limited to only deletions and insertions, the way to transform X to Y via the fewest operations has a very simple description. Let Z be the LCS of X and Y. First, transform X into Z via deletions. Second, transform Z into Y via insertions.

To see that this is optimal, note that any deleted character either came from X or was a new insertion, and inserting a character only to delete it later creates a suboptimal sequence of operations. Thus, we may assume all deletions came from X. We can then rearrange the operations

so all deletions come first without loss of generality, which does not affect the outcome. We have now reduced to the case where our operations first delete characters from X to form Z and then add characters to form Y, so Z must be a substring of both. This sequence of operations is clearly shortest if Z is as long as possible, i.e., the LCS of X and Y, as claimed. Thus, knowledge of the LCS lets us compute the edit distance, in this variant that does not allow substitutions.

Fortunately, the edit distance is not much harder to compute than the LCS. Let X and Y have lengths m and n respectively, and for all $i \in [m]$, $j \in [n]$, let $S[i][j]$ be the edit distance between $X[:i]$ and $Y[:j]$. We claim that

$$S[i][j] = \min\left(S[i][j-1]+1, S[i-1][j]+1, S[i-1][j-1]+\mathbb{I}_{X[i]\neq Y[j]}\right), \tag{9}$$

where the base cases are $S[0][j] = j$ for all $j \in [n]$ (the best strategy is to perform j insertions), and $S[i][0] = i$ for all $i \in [m]$ (the best strategy is to perform i deletions). Repeatedly evaluating (9) via memoization over two nested for loops yields an $O(mn)$ time algorithm for edit distance.

We now explain why (9) is true. We claim that without loss of generality, all operations are one of the following: delete a character from X, insert a character of Y, or substitute a character of Y for a character of X. All other deletions are due to an earlier insertion, i.e., operations are wasted; similarly, all other insertions and substitutions are wasteful.

Now, suppose $X[i] \neq Y[j]$, and consider what must happen to $X[i]$ and $Y[j]$. Because they are unequal, a small extension of our earlier claim shows we must either delete $X[i]$, insert $Y[j]$, or substitute $Y[j]$ for $X[i]$. Indeed, all other strategies (such as substituting $Y[j]$ for an earlier character, only to later have to remove $X[i]$) can be shown to be wasteful.

These three cases are captured by (9). When $X[i] = Y[j]$, we have an extra option of not touching either entry as they are already matched, i.e., $S[i][j] = S[i-1][j-1]$.

4.2 Longest palindromic substring

The longest palindromic substring problem is a classical algorithms problem on strings, and allows us to highlight a particularly creative DP solution due to Manacher [Man75]. Say that a string P is a *palindrome* if it is the same if the order of all characters is reversed, e.g., "dad" is a palindrome but "dab" is not. In the longest palindromic substring problem, we are given as input a string X of length n, and the goal is to determine the longest possible length of Y, a palindromic substring of X. For example, the longest palindromic substring of $X =$ "banana" is $Y =$ "anana".

We first introduce some helpful simplifications. We will assume without loss of generality that we are looking for the longest *odd-length* palindromic substring. It turns out that there is a clean reduction to this case which also handles even-length palindromes, which we will go over at the end of this subsection. Therefore, every palindrome $P := X[\ell : r]$ of interest has a *starting index* ℓ, an *ending index* r, and a *center index* c satisfying $c = \frac{\ell+r}{2}$. For any index $i \in [\ell, r]$, we also have a mirrored index $\text{mirr}_c(i) := 2c - i$, so that $\text{mirr}_c(c) = c$ and $\text{mirr}_c(\ell) = r$. Then, the condition that $X[\ell : r]$ is a palindrome can be concisely summarized as $X[i] = X[\text{mirr}_c(i)]$ for all $i \in [\ell, r]$.

There is a very natural, straightforward, $O(n^2)$ algorithm for the longest odd-length palindromic substring problem. Let $S[j]$ be the longest possible odd length of a palindromic substring with center index j, for all $j \in [n]$. For example, $S[j] \geq 1$ for all $j \in [n]$, because each individual character $X[j]$ is an odd-length palindromic substring with center index j. We claim we can compute $S[j]$ in $O(n)$ time for all $j \in [n]$. The strategy is to maintain a pointer r to the ending index of a current palindrome P, initialized to $r \leftarrow j$ and $P \leftarrow X[j]$. We will repeatedly increment r by one, checking whether $P[r] = P[\text{mirr}_j(r)]$ for the new value of r. If it is, we can grow the palindrome by one character. We continue doing this until we cannot, and the result is the longest palindromic substring centered at $X[j]$. This algorithm takes $O(1)$ time to perform all the steps each time we increment the pointer r, so $O(n)$ time total, to compute $S[j]$. Running this computation for all $j \in [n]$ takes $O(n^2)$ time as claimed. Note that this solution did not really use any memoization.

Perhaps the most natural first attempt to use DP is to define a subproblem $S[i][j]$ for each index pair $1 \leq i \leq j \leq n$, such that $L[i : j]$ is odd-length, and

$$S[i][j] = \begin{cases} \textbf{True} & X[i:j] \text{ is a palindrome} \\ \textbf{False} & X[i:j] \text{ is not a palindrome} \end{cases}.$$

Then it is simple to check that $S[i][j]$ satisfies the following recursive definition:

$$S[i][j] = S[i+1][j-1] \textbf{ and } X[i] == X[j],$$

where we handle the base cases of $S[j][j] =$ **True** for all $j \in [n]$. In fact, we can compute all $S[i][j]$ in $O(1)$ time, by proceeding in the same order in which we would compute them in the straightforward algorithm, i.e., picking each center and growing the palindrome out. Unfortunately, there are $\Theta(n^2)$ subproblems, and this results in another $O(n^2)$ time algorithm.

Manacher's algorithm. We present an elegant solution by [Man75] that runs in linear time. In Manacher's algorithm, as in the straightforward algorithm, we again define $S[j]$ to be the longest possible odd length of a palindromic substring with center index j. Our goal is to compute all of the $S[j]$ in $O(n)$ time. Some of the $S[j]$ will take us longer than others to compute, but overall, they take $O(n)$ time combined. The entire algorithm is based on the following observation.

Lemma 1. *Let $P_c = X[\ell : r]$ be an odd-length palindrome with center index $c = \frac{\ell+r}{2}$. Let $j \in [c, r-1]$ be an index in the right half of P_c, and let $\text{mirr}_c(j) := 2c-j$ be its mirrored index in the left half. Finally, let P_{mirr} be the longest odd-length palindromic substring with center index $\text{mirr}_c(j)$, and suppose P_{mirr} has starting index $> \ell$ (so it is a substring of P). Then, $S[j] = S[\text{mirr}_c(j)]$.*

Let us unpack Lemma 1. It applies in the following situation: we have already found a (hopefully long) palindromic substring P_c, centered at c, and are currently exploring its right half at index j. Suppose we have already memoized the special subproblems, i.e., $S[i]$, defined earlier, for all the previous indexes $i \in [j-1]$. Then, the claim is that if P_{mirr}, the longest palindrome centered at $\text{mirr}_c(j)$, is fairly short, i.e., it fits within P, then we can compute $S[j]$ in $O(1)$ time: it is the same as $S[\text{mirr}_c(j)]$. This fast computation will be the heart of our savings. We now prove this lemma.

Proof of Lemma 1. We first prove $S[j] \geq S[\text{mirr}_c(j)]$. By our definition of $S[j]$, this is saying there is some odd-length palindromic substring centered at $X[j]$ with length $\geq S[\text{mirr}_c(j)]$. We construct this palindromic substring as follows. Let P_{mirr} be as defined in the lemma statement, which is a substring of P by assumption. Reflect every character of P_{mirr} over the center index c. This results in P_j, some substring centered at j. In fact, P_j is a palindromic substring, since all the characters are the same as in P_{mirr}, because all of the reflection happens within the larger palindrome P. Thus, P_j is a length-$S[\text{mirr}_c(j)]$ palindromic substring centered at j as claimed.

Next, we prove $S[j] = S[\text{mirr}_c(j)]$. Suppose otherwise for the sake of contradiction, i.e., $S[j] > S[\text{mirr}_c(j)]$. This means that P_j, the length-$S[\text{mirr}_c(j)]$ substring constructed in the first half of the proof, can be extended to a longer palindrome of length $S[j]$ centered at j. However, this also implies that P_{mirr} can be extended by at least one character on both sides to another palindrome. This is since we assumed P_{mirr} had a starting index $> \ell$, so this extension happens within P. This contradicts the definition of P_{mirr} as the longest palindrome centered at $\text{mirr}_c(j)$. □

We can now state Manacher's algorithm in Algorithm 8. The key trick to speed up its implementation is that the algorithm always maintains a current palindromic substring $P_c = X[\ell : r]$, centered at an index $c = \frac{\ell+r}{2}$. We maintain two invariants about P_c. The first is that P_c is the longest palindromic substring centered at c, so its length is $S[c]$. Note that given the length of P_c, we can compute its endpoints as $c - \frac{S[c]-1}{2}$ and $c + \frac{S[c]-1}{2}$. The second is that r is the rightmost ending index of any palindrome the algorithm has found. We use this current palindromic substring P_c to speed up subproblem computations for as long as we can, via Lemma 1.

We now explain Algorithm 8's components and analyze its correctness. It proceeds in n iterations, always maintaining the two required invariants about P_c at the end of every iteration. To see why, suppose the invariants hold at the start of iteration j. There are two cases, one handled by Lines 5 to 7, and one handled by Lines 8 to 13. In the first case, Lemma 1 applies, so we know that $S[j] \leftarrow S[\text{mirr}_c(j)]$. This also means that the ending index of P_j, the length $S[j]$ substring centered at j, is $j + \frac{S[j]-1}{2} < r$, since it is the mirror index of $\text{mirr}_c(j) - \frac{S[j]-1}{2} > \ell$. Thus, P_j does not achieve the rightmost ending index of any palindrome, so we do not update r or P_c.

In the second case, we know for sure that we should update P_c. This is because Line 5 failed, which could mean one of two things. Either the longest palindrome centered at $\text{mirr}_c(j)$ extends beyond the current P_c, so there is a palindrome centered at j whose ending index is at least r, or j itself is already larger than r. In either case, we initialize a current palindrome centered at j and

Algorithm 8: LongestPalindromicSubstring(X)

```
1  Input: X, an Array instance containing n := |X| characters in Ω
2  S ← Array.Init(n)
3  (S[1], ℓ, c, r) ← (1, 1, 1, 1)
                    // Initialize maintained palindrome P_c ← X[1 : 1], the longest palindrome centered at c = 1.
4  for 2 ≤ j ≤ n do
5      if j < r and mirr_c(j) − (S[mirr_c(j)] − 1)/2 > ℓ then
                                                                  // Lemma 1 applies.
6          S[j] ← S[mirr_c(j)]
7      end
8      else
                                  // Lemma 1 does not apply, so we should grow the palindrome and update r.
9          (S[j], ℓ, c, r) ← (1 + 2 max(0, r − j), j − max(0, r − j), j, j + max(0, r − j))
                        // We update P_c ← X[j − max(0, r − j) : j + max(0, r − j) = max(j, r)] where c ← j.
10         while r + 1 ≤ n and X[ℓ − 1] == X[r + 1] do
11             (S[j], ℓ, r) ← (S[j] + 2, ℓ − 1, r + 1)
12         end
13     end
14 end
15 return max_{j∈[n]} S[j]
```

ending at $\max(j, r)$ in Line 9. We then repeatedly grow it until we cannot anymore in Lines 10 to 12. This preserves the invariant that the new P_c has the rightmost ending index.

The correctness proof follows by combining the two cases presented above. We now discuss the runtime of Algorithm 8. The total time complexity of Lines 5 to 7 is $O(n)$, since they run at most n times each requiring constant time. Similarly, Line 9 requires at most $O(n)$ times total.

This leaves bounding the runtime of Lines 10 to 12. It is perhaps least clear that these lines also require only $O(n)$ time total. In some iterations, we could require substantially growing the length of the current palindrome, e.g., if we are in an iteration where the longest discovered palindromic substring changes. However, notice that every time Lines 10 to 12 run, r increments by 1. Moreover, r never decreases throughout the entire algorithm; indeed, the only other line where it could change is Line 9, where it is set to $r \leftarrow \max(j, r)$. Thus, Lines 10 to 12 run at most n times total, because $r \leq n$ throughout the algorithm. This completes the runtime analysis.

Manacher's algorithm is a clever application of pointer tricks. It highlights a powerful tool: using pointer movement to bound runtimes. We showed that the complexity of all parts of Algorithm 8 was constant time per iteration, other than a loop that involved incrementing the pointer r each time. Since r can only increment n times, we achieved our claimed bound.

Even-length palindromes. To generalize to even-length palindromes, it suffices to first insert a specially chosen character, say '#' for the sake of this discussion, between every pair of characters in the initial string X (as well as the start and end), and then run Algorithm 8 on the modified string X'. For example, if $X =$ "banaana", then $X' =$ "#b#a#n#a#a#n#a#". The claim is that the longest odd-length palindromic substring of X' (e.g., "#a#n#a#a#n#a#") yields the longest overall palindromic substring of X (e.g., "anaana"), allowing for odd or even length, after removing all the inserted '#' characters. This is true because both parities of palindromes P in X yield corresponding odd-length palindromes in X', centered either at the original center of (odd-length) P or the '#' character in between the two middle characters of (even-length) P. Thus, solving longest odd palindromic substring on X' suffices to solve longest palindromic substring on X.

5 Graphs

In this final section of these notes, we cover graph-based applications of dynamic programming. We follow notation and conventions from Section 4, Part I.

5.1 Game theory

Consider a two-player game with a winner and a loser. We can model the set of states the gameplay can be in as a directed graph, where each game state is a vertex of the graph. For example, if the game is tic-tac-toe (where the 'X' player goes first), each game state corresponds to a possible state of the board in the middle of gameplay. In particular, a vertex in the tic-tac-toe graph corresponds to a game state where there are i copies of 'X' and j copies of 'O' placed in the nine grid cells, such that $i + j \leq 9$, $i = j$ or $i = j + 1$, and there is at most one three-in-a-row that has occurred. These are the only possible game states because the 'X' player has either played the same number of moves as the 'O' player (so it is the 'X' player's turn), or one more (so it is the 'O' player's turn). Moreover, the game ends once a single three-in-a-row is achieved.

We now explain the graph representation of the two-player game in more detail. Each game state s has a directed edge pointing to all possible game states t that can appear, after a single move from s. For example, the starting game state of tic-tac-toe, i.e., an empty board, has nine outgoing edges. Each outgoing edge points to a game state with a single 'X' somewhere on the board.

In the remainder of this section, we call the first player Alice and the second player Bob. We call a game state terminal if no further moves can be taken (i.e., the game has ended, and the state has no outgoing edges). For simplicity, in the following discussion suppose that we are playing a game with no ties, so every terminal state either ends in Alice or Bob winning. Our goal is to determine whether Alice has a winning strategy. That is, is it true that for any moves that Bob responds with, Alice can always respond with a move that ensures she wins the game eventually?

It turns out that there is a natural dynamic programming-based way to answer this question for many common games. Consider for example a situation where the tic-tac-toe game state looks like

X		X
	O	
O		X

(10)

and it is Bob's turn to play. We claim this is a losing position for Bob. It is intuitive why this is the case: whatever Bob does, Alice can respond with the top-center square or the center-right square. For example, if Bob takes the top-center square, the resulting game state is:

X	O	X
	O	
O		X

(11)

which is a forced win for Alice, as now she can take the center-right square. Thus, in the strange variant of tic-tac-toe where we initialize the game state to (10) and ask Bob to play next, Alice has a winning strategy. We arrived at this conclusion by reverse-engineering from terminal states: (11) is winning for Alice since there is an edge she can take to a winning terminal state, and (10) is winning for Alice because no matter what Bob does, Alice can move to a winning state.

We can generalize this intuition to determine whether the starting position is winning. Our goal is to recursively label each vertex v, corresponding to a game state in the game graph, as either **True** or **False**, stored in a DP subproblem solution table S. At the end of the algorithm, a vertex v will have $S[v] =$ **True** if there is a gameplay strategy for Alice that ensures she wins no matter what Bob does, from the game state at v, and will have $S[v] =$ **False** otherwise. By solving the subproblem at the starting vertex, we determine whether Alice has an overall winning strategy.

The way to recursively solve a subproblem is to use the following formula:

$$S[v] \leftarrow \left(\text{Alice plays next } \textbf{and} \bigvee_{(v,u) \text{ is an edge}} S[u]\right) \textbf{ or } \left(\text{Bob plays next } \textbf{and} \bigwedge_{(v,u) \text{ is an edge}} S[u]\right), \tag{12}$$

where any terminal state in the graph is first labeled as **True** or **False** depending on who has won, and the $\bigwedge$ and $\bigvee$ symbols denote taking the **and** or **or** over a set of Booleans.

Let us unpack (12). If $S[v]$ is not a terminal state, that means either Alice or Bob plays next, so only one of the two cases in (12) can evaluate to **True**. In the first case, Alice is up to play, and she has a winning strategy if there is some edge she can take (corresponding to a move) leading to another winning state for her. This is expressible as an **or** over all of the subproblems $S[u]$ she can move to. For example, (11) is a winning state because Alice can move to a winning terminal state. In the second case, Bob is up to play, which means that Alice can only guarantee a win if *all of Bob's moves* lead to winning states for Alice. This is similarly expressible as an **and**.

We next move to a subtle issue: in what order do we recursively evaluate (12)? It turns out that there exists an order that works iff the game graph is a *directed acyclic graph* (DAG), which we define and study in Section 5.2. To get a sense of what could go wrong, suppose that there is a directed cycle among the game states (a, b, c), where edges (a, b), (b, c), and (c, a) all exist. To use (12) to compute $S[a]$, we need to know the value of $S[b]$, but $S[c]$ is needed for $S[b]$ and $S[a]$ is needed for $S[c]$. This leads us to a cycle where we are unable to compute the truth value of any of these game states successfully. This situation is essentially the logical fallacy of circular reasoning in algorithmic form, just as a successful recursive algorithm is an example of strong induction.

For games like tic-tac-toe, where there is an obvious measure of progress (e.g., the number of symbols on the board), we can show that no cycles can exist in the game graph, so it is a DAG. To see why, notice that every time an edge is taken (i.e., a move is made), the number of symbols grows. Thus, the existence of a cycle would yield a contradiction, since we cannot keep adding symbols and eventually result in the same number of symbols that we started with. However, for games like chess where pieces can move back to where they came from, undoing any progress measure, cycles can easily occur. This motivates the inclusion of rules preventing repetitions; indeed, chess tournaments are played with such a "no threefold repetition" rule.

Another complication is the size of the game graph: any algorithm that fills out the DP table must at least visit all of the vertices to compute their subproblem values, so if the number of vertices is large, this can be very cumbersome. For example, the number of game states in chess is $\gg 10^{40}$, making the game graph infeasible to explore in full. In practice, heuristics are often used to simplify the formula (12), such as proceeding top-down and only evaluating (12) recursively to a certain depth rather than all the way to the terminal states.

Ties. The formula (12) straightforwardly generalizes if terminal game states can result in a tie. The only difference is that at the beginning, any tied terminal state is labeled as **False** (in addition to any terminal state where Bob wins). This is due to the observation that Bob can prevent an Alice win as long as he can move to either a tied terminal state, or a terminal state where he wins.

Zero-sum games. This DP-based solution further extends to a family of two-player games called zero-sum games, which generalize those in which there is always a winner and a loser. In zero-sum games, Alice and Bob each have a score at the end of the game, such that the sum of their scores in any terminal state is zero. For example, win-loss games are zero-sum by setting the score of the winner to 1, and the score of the loser to -1. More generally, suppose a game involves subdividing a set of objects via taking them in turn, such that each object has a value, and each player's score is the total value they have taken. We can convert this into a zero-sum game by observing that the sum of the two scores at the end of the game is always the total value, say T. We can then define a new game where Bob starts with a value of $-T$, so that if Alice ends with k total points, Bob ends with $-T + (T - k) = -k$ points. This new game is thus zero-sum.

The reason why zero-sum games are important is that we can succinctly express Alice and Bob's goals: Alice is trying to maximize her score, and Bob is trying to minimize Alice's score. This is because in a zero-sum game, the smaller Alice's score is, the higher Bob's score is, so minimizing Alice's score is aligned with Bob's goal. In these situations, we can generalize the formula (12) to

$$S[v] = \begin{cases} \max_{(v,u) \text{ is an edge}} S[u] + \text{Alice's score change from taking edge } (v,u) & \text{Alice plays next} \\ \min_{(v,u) \text{ is an edge}} S[u] + \text{Alice's score change from taking edge } (v,u) & \text{Bob plays next} \end{cases},$$

where $S[v]$ denotes the maximum value Alice can guarantee starting at game state v, and we fill in terminal states first with a score of 0. This way, once we have solved all DP subproblems, the initial game state subproblem solution contains the final score after taking the optimal path to a terminal node. The formula above can compute the optimal strategy for Alice in exactly the same settings as (12) can for win-loss games, i.e., whenever the game graph is acyclic.

5.2 Directed acyclic graphs

A directed graph $G = (V, E, \mathbf{w})$ with vertices V, edges $E \subseteq V \times V$, and edge weights $\mathbf{w} \in \mathbb{R}^E_{\geq 0}$ is said to be a *directed acyclic graph* (DAG) if it contains no directed cycles. That is, there should be no set of distinct vertices $\{v_1, v_2, \ldots, v_i\} \subseteq V$ in the graph, such that all of the directed edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{i-1}, v_i), (v_i, v_1)$ belong to the edge set E (which would cause a cycle).

The most important property of DAGs is that their vertices can be relabeled with the numbers in $[n]$ where $n := |V|$, such that there are no "backwards edges." That is, after this relabeling of the vertices, all edges are of the form (i, j) where $i < j$. Such a vertex ordering is called a *topological ordering*.[3] It turns out that a directed graph's vertices admit a topological ordering iff it is a DAG. We will later prove the harder direction, i.e., that all DAGs admit a topological ordering on their vertices, in Part V of the lecture notes. In fact, such an ordering can be computed in $O(|V| + |E|)$ time. Here, we give a short proof of the simpler direction, stated as follows.

Lemma 2. *Let $G = (V, E, \mathbf{w})$ be a directed graph, with vertices V identified with the set $[n]$ in a topological ordering, i.e., all edges in E are of the form (i, j) for $i < j$. Then G is a DAG.*

Proof. Suppose for the sake of contradiction that G is not a DAG, so there are vertices $\{i_1, i_2, \ldots, i_k\}$ forming a cycle, i.e., such that $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, i_1) \in E$. Because V is in a topological ordering, the existence of the first $k - 1$ edges in the cycle implies $i_1 < i_2 < i_3 < \ldots < i_k$. Therefore, $i_1 < i_k$. However, existence of the last edge implies $i_k < i_1$, a contradiction. □

Note that the proof technique in Lemma 2 is very similar to our argument in Section 5.1 about why the tic-tac-toe game graph is acyclic. In particular, any sequence of edges taken in the tic-tac-toe game graph grows the number of symbols on the game board by one each, so it cannot return to the same starting point. Similarly, in Lemma 2 we used the indices $i_1, i_2, \ldots, i_k$ as our measure of progress, as the index always grows after taking an edge in a topological ordering.

We hinted in the previous section that the presence of directed cycles can cause significant issues for DP-based solutions. We now formalize this statement. Consider an arbitrary DP algorithm, involving a set of subproblems $S[v]$. We can associate a *dependency graph* with the DP algorithm, where there is a directed (unweighted) edge (u, v) for every pair of subproblems u, v, where the recursive formula for $S[v]$ depends on $S[u]$. For example, in FibBasic (Algorithm 2), the subproblems $S[i]$ correspond to computing the i^{th} Fibonacci number F_i for each $i \in [n]$. There are $\leq 2n$ edges in the FibBasic dependency graph, as each vertex $i \in [n-2]$ has two outgoing edges $(i, i+1)$ and $(i, i+2)$, and we also have an edge $(n-1, n)$. This is because other than boundary issues, each $S[i]$ is used in the computation of $S[i+2] = S[i+1] + S[i]$, as well as in the computation of $S[i+1] = S[i] + S[i-1]$. As another example, the dependency graph for playing two-player win-lose games is just the game graph with all edges reversed, since the formula (12) at v depends on all $S[u]$ for outgoing edges (v, u) in the game graph, so in the dependency graph, we add (u, v).

The dependency graph of a DP algorithm has a very simple interpretation. It says that before computing the value of a subproblem $S[v]$, we have to make sure we have already memoized all of the $S[u]$ where u points to v (i.e., (u, v) is an edge). Otherwise, we cannot evaluate the formula at $S[v]$. A convenient fact is that there exists a consistent ordering for evaluating DP subproblems, i.e., an ordering of the vertices such that each subproblem $S[v]$ is evaluated after all of the $S[u]$ it depends on, iff the dependency graph is a DAG. To show one direction, if the dependency graph is not a DAG, then there exists a sequence of dependencies $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, i_1)$, where to solve subproblem $S[i_2]$, we need to have memoized $S[i_1]$, etc. If, for the sake of contradiction, there was a way to evaluate these subproblem solutions consistently, then there must be some vertex on the cycle whose subproblem we solve first. However, we cannot solve this subproblem, as it depends on a subproblem that we have not yet solved, a contradiction.

On the other hand, suppose that the dependency graph is a DAG, and without loss of generality suppose that we have relabeled the vertices in a topological order. We claim that evaluating the subproblems in sequence one by one, starting from vertex 1 and ending at vertex $n := |V|$, is a consistent evaluation order. We prove this by strong induction, where the hypothesis is that all of the subproblems required in computing $S[j]$ have been memoized when we reach vertex j in the

[3] This definition appears to have nothing to do with topology from mathematics, but is instead related to the concept of a "network topology," which is jargon from the study of communication networks.

evaluation order, so we can successfully evaluate $S[j]$. First, when we evaluate $S[1]$, we do not require any other subproblems, since there can be no edges $(i, 1)$ in a topological ordering, proving the base case. Next, for the strong inductive step, suppose that all of the subproblems $S[1], \ldots, S[j]$ have been successfully memoized. When evaluating $S[j+1]$ in a topological ordering, the formula can only depend on vertices $S[i]$ for $i < j+1$, i.e., $i \in [j]$. These have all been memoized by the strong induction hypothesis, so we can compute $S[j+1]$, completing the induction.

It is a good exercise to check that all of the DP algorithms covered in these notes actually induce dependency graphs that are DAGs. However, the reader can find comfort: as long as you are convinced that the DP evaluation orders we declared earlier are indeed consistent, the dependency graph must be a DAG, as we just proved these statements are equivalent.

Single-source shortest paths on DAGs. Finally, for obvious reasons, algorithmic problems on DAGs are ripe for DP-based solutions: they already come with an evaluation order! We give a simple example here: the single-source shortest paths problem (SSSP). In this problem, we are given as input a DAG $G = (V, E, \mathbf{w})$, where the vertices V are in a topological ordering. Our goal is to compute the minimum possible weight of a path from vertex 1 to vertex j for all $j \in [n]$, where we let $n := |V|$ and $m := |E|$. That is, we are interested in computing

$$\min_{\substack{P \text{ is a path from } 1 \to j \\ P \subseteq E}} \sum_{e \in P} \mathbf{w}_e. \tag{13}$$

The formula (13) has a natural interpretation as the minimum length of a path from 1 to j, where we view the weight $\mathbf{w}_e$ as the length of edge e. If it is impossible to reach vertex j from vertex 1 by following a path, we let (13) evaluate to ∞ by default.

We claim that it suffices to define subproblems $S[j]$ for all $j \in [n]$, where $S[j]$ is the value of (13) for a given value of j, initializing $S[1] = 0$ as the base case and $S[j] = \infty$ for all $j \neq 1$. We evaluate these subproblems one by one in incremental order, using the formula

$$S[j] = \min_{(i,j) \in E} S[i] + \mathbf{w}_{(i,j)}. \tag{14}$$

If there are no such $(i, j) \in E$, we let (14) evaluate to ∞ by default.

To see why (14) is true, any path P to vertex j must include at least one edge (i, j), where potentially $i = 1$. If no such edge (i, j) exists, then j is unreachable (from any vertex, not just from 1), so (13) is ∞. Otherwise, we must take some edge $\mathbf{w}_{(i,j)}$, and now our goal is to get from vertex 1 to vertex i as cheaply as possible, yielding the formula (14). As V is topologically ordered, $i < j$, so we have memoized $S[i]$ by the time we need to evaluate $S[j]$ in (14). Notice that this argument works just fine with negative-weight edges $e \in E$ having $\mathbf{w}_e < 0$; the cheapest way to get from 1 to j if you are forced to take the edge (i, j) is still to get from 1 to i as cheaply as possible first. The runtime of our SSSP algorithm on DAGs is $O(n + m)$, since we must evaluate all of the n formulas (14), and each of the m edges $(i, j) \in E$ is used in a formula exactly once.

5.3 All-pairs shortest paths

We are now ready to tackle our last DP example: the all-pairs shortest paths (APSP) problem. In this problem, we are given as input a directed graph $G = (V, E, \mathbf{w})$. Here, G is not necessarily a DAG. For simplicity, we let $n := |V|$ and $m := |E|$, and we again identify the vertices V with the set $[n]$. Our goal is to compute, for all of the $n(n-1)$ pairs $(i, j) \in V \times V$ with $i \neq j$, the value of

$$\min_{\substack{P \text{ is a path from } i \to j \\ P \subseteq E}} \sum_{e \in P} \mathbf{w}_e. \tag{15}$$

That is, for all distinct $(i, j) \in V \times V$, we want to compute the shortest path from i to j. Remarkably, in the course of tackling this problem, we will combine many of the techniques we have seen thus far, including multidimensional DP, prefix-based DP, DAGs, and even divide-and-conquer from Part II of the lecture notes. It is thus a very natural ending topic for the DP unit.

This problem has a straightforward solution on DAGs: run our SSSP algorithm from Section 5.2 n times, once with each starting vertex. Indeed, for any $i \in [n]$, taking the induced subgraph on the vertices $\{i, i+1, \ldots, n\}$ results in another DAG, and the SSSP algorithm from Section 5.2 on this

induced subgraph (with shifted indices so that i is the first vertex) gives all of the shortest path values (i, j) for $j > i$. Therefore, running the SSSP algorithm with each $i \in [n]$ solves APSP on DAGs in time $O(n^2 + mn)$, as each SSSP computation on an induced DAG takes time $O(n + m)$. Since $m \leq n^2$, in the worst case, this gives an $O(n^3)$ time algorithm for APSP on DAGs.

The main result of this section is that we can solve APSP in $O(n^3)$ time, for *arbitrary directed graphs*, with one caveat. It turns out that the APSP problem is not well-defined if there exist any negative-weight cycles, i.e., directed cycles $C \subseteq E$ such that $\sum_{e \in C} \mathbf{w}_e < 0$. This is because for any pair (i, j) such that there exists a $i \to j$ path passing through an edge of C, we can loop around the cycle C as long as we want to decrease the weight of the path arbitrarily. Therefore, for the remainder of the section, we explicitly assume that the graph G contains no negative-weight cycles. This lets us use the following helpful lemma, bounding the size of any shortest path.

Lemma 3. *Let $G = (V, E, \mathbf{w})$ be a directed graph with no negative-weight cycles. Then for every pair of vertices $(i, j) \in V \times V$ with $i \neq j$, there exists a shortest path from i to j (i.e., a path achieving the value (15)), with at most $n - 1$ edges, where $n := |V|$.*

Proof. Let P be the shortest path from i to j using the fewest edges. We claim P has $\leq n - 1$ edges. Suppose for contradiction that there are $\geq n$ edges in P, so there are $\geq n + 1$ vertices involved. By the pigeonhole principle, some $i \in [n]$ appears in P at least twice. Thus, there must be a cycle from i to i in P. This cycle has nonnegative weight, so it must in fact have zero weight, as otherwise removing it creates a shorter path, contradicting the definition of P. Now, we can replace P with another shortest $i \to j$ path with fewer edges, also contradicting P's definition. □

Why is Lemma 3 useful? Consider first what goes wrong if we try to apply the SSSP algorithm from Section 5.2 to a non-DAG G. The dependency graph corresponding to the DP subproblems (14) is just G itself, because the solution at every vertex depends on all incoming edges. If G has a directed cycle, repeatedly trying to compute (13) necessarily results in the implementation issue described in Section 5.2. This is because unlike in the case of DAGs, for general directed graphs, there is no notion of "progress" or natural ordering to solve DP subproblems in.

The key idea to break out of this chicken-and-egg problem is to make our own progress measure by adding an extra dimension (similarly to Section 3.3, where we augmented with the value). In particular, we will augment with an extra parameter ℓ that corresponds to the maximum number of edges allowable in a path. By Lemma 3 we can always bound $\ell \leq n - 1$. We will solve the DP subproblems for each value of ℓ, one at a time incrementally, creating a natural ordering.

We now describe our first DP-based solution to APSP more formally. We define a set of DP subproblems $S[i][j][\ell]$, for all pairs $(i, j) \in [n] \times [n]$ with $i \neq j$, and all possible cardinalities $0 \leq \ell \leq n - 1$ of a shortest path. Our goal is to compute all of the $S[i][j][\ell]$, where

$$S[i][j][\ell] := \min_{\substack{P \text{ is a path from } i \to j \\ P \subseteq E \\ |P| \leq \ell}} \sum_{e \in P} \mathbf{w}_e. \tag{16}$$

That is, compared to (15) we add the additional stipulation that P contains at most ℓ edges. By Lemma 3, the slice of the 3-d array S corresponding to $\ell = n - 1$ contains all of the shortest path lengths, as every shortest path uses at most $n - 1$ edges, so this restriction changes nothing.

We can now reuse the idea from our DAG SSSP algorithm, since a nonempty $i \to j$ path of length $\leq \ell$ is the combination of its last edge, say (k, j) for some k, and the shortest possible path from i to k of length $\leq \ell - 1$. Moreover, all of the base cases $S[i][j][1]$ are straightforward to evaluate: they are either $\mathbf{w}_{(i,j)}$ if $(i, j) \in E$, or ∞. Thus, we can use the formula

$$S[i][j][\ell] = \min_{(k,j) \in E} S[i][k][\ell - 1] + \mathbf{w}_{(k,j)}, \tag{17}$$

derived similarly to (14) (but making sure to adjust the ℓ parameter), in the ordering that solves all subproblems for each value of ℓ before incrementing ℓ. This is correct because computing $S[i][j][\ell]$ using (17) only depends on values of $S[\cdot][\cdot][\ell - 1]$, i.e., the previous set of subproblems. Each of the $O(n^3)$ subproblems $S[i][j][\ell]$ takes $O(n)$ time to evaluate via (17) and memoization of previous subproblems. Thus, this algorithm for the APSP problem runs in $O(n^4)$ time.

Let us apply our interpretation from Section 5.2 to demystify this solution: how did we break the issue of cycles by using an extra dimension? It turns out that all shortest paths in G are equivalent to shortest paths in a different *layered graph* L, which is importantly a DAG.[4]

The vertex set of L consists of n copies of V, where the ℓ^{th} copy for $\ell \in [n]$ is denoted $V^{(\ell)}$ and viewed as a layer in the graph. Similarly, for all vertices $i \in [n]$, we denote the ℓ^{th} copy of i by $i^{(\ell)}$. It remains to specify the edges of L. We add $n-1$ zero-weight "shortcut" edges $\{(i^{(\ell)}, i^{(\ell+1)})\}_{\ell \in [n-1]}$ for all $i \in [n]$, so there is no cost to moving to a copy of the same vertex on any future layer. We also add copies of all edges E between each pair of adjacent layers $V^{(\ell)} \times V^{(\ell+1)}$, such that each original edge $(i,j) \in E$ results in $n-1$ new edges $\{(i^{(\ell)}, j^{(\ell+1)})\}_{\ell \in [n-1]}$. L is clearly a DAG, because every edge moves to a future layer, so there is no way of cycling back to an older layer.

We claim that for all $(i,j) \in V \times V$, the shortest path distance (15) in G is the shortest path distance between $i^{(1)}$ and $j^{(n)}$ in L. To see this, given a shortest $i \to j$ path in G of length ℓ, we can take the same edges in L to reach $j^{(\ell+1)}$, and then take shortcut edges to reach $j^{(n)}$. Similarly, given a shortest $i^{(1)} \to j^{(n)}$ path in L, we can follow the same steps in G, where we make sure to not move whenever a shortcut is used. Thus, the two problems are indeed equivalent.

Divide-and-conquer. While our APSP algorithm thus far runs in polynomial time, it still falls short of the $O(n^3)$ time algorithm promised at the beginning of the section. Here we give a simple improvement to our algorithm that brings us within a logarithmic factor. The idea is to divide-and-conquer. Instead of defining the subproblems $S[i][j][\ell]$ in (16) to correspond to length-$\le \ell$ paths, we instead let them correspond to length-$\le 2^\ell$ paths:

$$S[i][j][\ell] := \min_{\substack{P \text{ is a path from } i \to j \\ P \subseteq E \\ |P| \le 2^\ell}} \sum_{e \in P} \mathbf{w}_e.$$

By Lemma 3, it is enough to solve these subproblems for $1 \le \ell \le \lceil \log_2(n) \rceil$. Why can we evaluate these values quickly? The key observation is that instead of treating a length-$\le 2^\ell$ path as the concatenation of a length-1 path and a length-$\le 2^\ell - 1$ path, we can instead treat it as concatenating two length-$\le 2^{\ell-1}$ paths, from $i \to k$ and $k \to j$. This yields the recursive formula

$$S[i][j][\ell] = \min_{k \in V} S[i][k][\ell-1] + S[k][j][\ell-1]. \tag{18}$$

This still takes $O(n)$ time to compute, but now there are only $O(n^2 \log(n))$ subproblems in total due to our smaller range of ℓ, giving an overall $O(n^3 \log(n))$-time algorithm for APSP. Interestingly, this algorithm is very reminiscent of our "repeated squaring" approach to matrix multiplication in Section 5.3, Part II. For the interested reader, this point is expanded upon in Chapter 9.7 of [Eri24].

Floyd-Warshall algorithm. It turns out it is even possible to remove this extra $\log(n)$ factor. We briefly summarize a famous result known as the Floyd-Warshall algorithm, originally attributed to [Flo62, War62] but actually published earlier in [Roy59]. The algorithm is similar to our previous algorithms, but overcomes the key issue of each subproblem taking $O(n)$ time to evaluate, as was the case in (17) and (18). To obtain this improvement, we revisit an old friend: prefix-based DP.

The point of prefix-based DP is to significantly reduce on computation in subproblems, since to reduce a prefix of a list $L[:j]$ to a smaller prefix $L[:j-1]$, we only need to make a decision about the j^{th} entry, and there are typically only 2 options. For example, this idea was used in all of the formulas (2), (3), (4), (6), and (8), to speed them up to run in $O(1)$ time.

It takes some ingenuity to define the right prefix-based subproblems for APSP, but the Floyd-Warshall algorithm does exactly this. Specifically, for all pairs $(i,j) \in [n] \times [n]$ and all possible prefix lengths $k \in [n]$, we define a subproblem

$$S[i][j][k] := \min_{\substack{P \text{ is a path from } i \to j \\ P \subseteq E \\ P \text{ only uses intermediate vertices from } [k]}} \sum_{e \in P} \mathbf{w}_e.$$

Here, we say an $i \to j$ path P only uses intermediate vertices from $[k]$ if it consists of the edges $(i_0, i_1), (i_1, i_2), \ldots, (i_{\ell-1}, i_\ell)$, where $i_0 = i$ and $i_\ell = j$, and $i_a \in [k]$ for all $a \in [\ell - 1]$. That is, all of

[4] We remark that this layered graph is slightly different than the dependency graph of our DP-based APSP algorithm, because it does not consider each subproblem (i,j) as a vertex. However, it is similarly motivated.

the vertices $i_1, \ldots, i_{\ell-1}$ along the path other than possibly i and j are among the first k vertices. Because of our restriction to prefixes, these subproblems admit the recursive definition

$$S[i][j][k] = \min\left(S[i][j][k-1], S[i][k][k-1] + S[k][j][k-1]\right),$$

which only requires selecting among two options. This is because a shortest path P with intermediate vertices from $[k]$ either does not use k (the first term above), or we can assume without loss of generality that it does so exactly once by the argument in Lemma 3. In this latter case, the two subpaths formed by splitting P at k both are restricted to intermediate vertices $[k-1]$ (the second term above). Each of the $O(n^3)$ Floyd-Warshall subproblems can be solved in $O(1)$ time via memoization, by proceeding incrementally over k, giving our claimed $O(n^3)$ runtime.

Further reading

These notes contain a collection of famous DP problems, some of which do not appear in previous standard texts. Here, we summarize additional resources for some of the more common DP problems that we covered. For general overviews of DP and more examples, see Chapter 14, [CLRS22], or Chapter 3, [Eri24], or Chapter 6, [KT05], or Chapters 16-18, [Rou22].

For more on Section 3.1, see Chapter 6.1, [KT05].

For more on Section 3.2, see Chapter 3.6, [Eri24].

For more on Section 3.3, see Chapter 3.8, [Eri24] or Chapter 6.4, [KT05] or Chapter 16.5, [Rou22].

For more on Section 4.1, see Chapter 14.4, [CLRS22] or Chapter 3.7, [Eri24].

For more on Section 5.2, see Chapters 6.3 to 6.4, [Eri24] or Chapter 3.6, [KT05].

For more on Section 5.3, see Chapter 9, [Eri24] or Chapter 18, [Rou22].

References

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[Eri24] Jeff Erickson. *Algorithms*. 2024.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[KT05] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 2005.

[Man75] Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975.

[Rou22] Tim Roughgarden. *Algorithms Illuminated*. Soundlikeyourself Publishing, 2022.

[Roy59] Bernard Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[Sch61] Craige Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[Wik24a] Wikipedia. Longest increasing subsequence. https://en.wikipedia.org/wiki/Longest_increasing_subsequenc 2024. Accessed: 2024-09-08.

[Wik24b] Wikipedia. Maximum subarray problem. https://en.wikipedia.org/wiki/Maximum_subarray_problem, 2024. Accessed: 2024-08-24.