CS 577: Introduction to Algorithms Fall 2022

Homework 3

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers dynamic programming. **Problem 3 must be submitted for grading by the start of class on 10/4, i.e., by 2:29pm.** Please refer to the homework guidelines on Canvas for detailed instructions. For this assignment, include a space complexity analysis in addition to the usual time complexity analysis.

Warm-up problems

1. Suppose you are given a string of letters representing text in some foreign language, but without any spaces or punctuation. You want to break this string into its individual constituent words. For example, you might be given the following passage from Cicero's famous oration in defense of Lucius Licinius Mureta in 62BCE, in the *scriptia continua* of classical Latin:

   PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
   ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
   ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE[1]

   A fluent Latin reader would parse this string (in modern orthography) as *Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae.*

   Some strings can be parsed in multiple ways, but you are not concerned with that. You want to know, given a string S of n characters, can it be segmented into words *at all*? Assume you have access to a subroutine $\textsc{IsWord}(i, j)$ that takes indices i, j with $i \leq j$ as input and indicates whether $S[i \ldots j]$ is a "word" in the foreign language, and that it takes constant time to run.

   Design an algorithm that solves this problem in $O(n^2)$ time.

2. Recall that a subsequence of an array is obtained by deleting any number of positions (possibly none, possibly all). A subarray is a sequence in which the positions that are not deleted form an interval (possibly empty). For example, consider the array $(-1, -2, -3, -4, -5)$. A valid subsequence is $(-1, -3, -5)$; it is not a subarray. A valid subarray is $(-2, -3)$.

   You are given an array $A[1 \ldots n]$ of integers and want to find the maximum sum of the elements of (a) any subsequence, and (b) any subarray. The sum of an empty subarray is 0. For the example above, the maximum-sum subarray and subsequence has length zero.

   Design an $O(n)$ algorithm for both problems.

Regular problems

3. [Graded] The library has n books that must be stored in alphabetical order on adjustable-height shelves. The i-th book has height $h[i]$ and thickness $t[i]$, $i \in [n]$. The width of the

[1] "First of all, dignity in such paltry knowledge is impossible; this is trivial stuff, mostly concerned with individual letters and the placement of points between words."

shelf is fixed at w, and the sum of the thicknesses of books on a single shelf cannot exceed w. The next shelf will be placed atop the tallest book on the shelf. You can assume the shelving takes no vertical space.

   Design an algorithm that minimizes the total height of shelves used to store all the books. You are given the list of books in alphabetical order. Your algorithm should run in time $O(n^2)$.

4. When you were little, every day on your way home from school you passed the house of your grandmother. If you stop by for a chat on day i, Grandma would give you a number $\ell[i]$ of lollipops but also tell you that she won't give you any more lollipops for the next $k[i]$ days. For example, if day 1 is a Monday and $k[1] = 3$, then if you visit her that day, you would have to wait patiently until Friday to get your next lollipop.

   Design an $O(n)$ algorithm that takes as input the arrays $\ell[1 \ldots n]$ and $k[1 \ldots n]$, and outputs the maximum number of lollipops you can get during those n days.

5. You want to go running and have n minutes to spare. You want to run as long a distance as possible, but your exhaustion level cannot exceed a given limit e. Initially your exhaustion level is zero. During each minute, you can choose to either run or rest for the whole minute. When you choose to run the i-th minute, you run exactly $d[i]$ feet during that minute, and your exhaustion level increases by one. When you choose to rest, you run zero feet during that minute, and your exhaustion level decreases by one (if your exhaustion level is already zero, it will stay zero). Moreover, when you choose to rest, you must continue to rest until your exhaustion level reaches zero; once it reaches zero, you can again choose to run or rest. Finally, your exhaustion level at the end of your run must be zero.

   For example, for $e = 2$ and $d[1 \ldots 5] = (500, 300, 400, 200, 1000)$, the best you can do is to run during minutes 1 and 3, and rest the other minutes, so the answer is $500 + 400 = 900$.

   Design an algorithm that takes a positive integer e and an array $d[1 \ldots n]$ of $n \geq 1$ positive integers as input, and ouputs the maximum distance you can run subject to the constraints above. Your algorithm should run in time $O(ne)$ and space $O(e)$.

Challenge problem

6. There is a famous joke-riddle for children:

   > Three turtles are crawling along a road. One turtle says, "There are two turtles ahead of me." Another turtle says, "There are two turtles behind me." The third turtle says, "There are two turtles ahead of me and two turtles behind me." How could this have happened? Of course, the third turtle is lying!

   In this problem you have n turtles crawling along a road. Some of them are crawling side-by-side, so there may be turtles that are neither ahead nor behind one another. Each turtle makes a statement of the form: "There are a_i turtles ahead of me, and b_i turtles behind me."

   Your task is to find the minimal number of turtles that must be lying. More formally, let x_i denote the position along the road of turtle i, $1 \leq i \leq n$. Some turtles may be at the same position. Turtle i tells the truth if and only if a_i is the number of turtles j such that $x_j > x_i$ and b_i is the number of turtles j such that $x_j < x_i$. Otherwise, turtle i is lying.

   Design an $O(n \log n)$ algorithm for this problem.

Programming problem

7. SPOJ problem Coins Game (problem code MCOINS).

CS 577: Introduction to Algorithms Fall 2022

Homework 1 Solutions to Warm-Up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroieski

Problem 1

You want to know, given a string S of n characters, can it be segmented into words? Assume you have access to a subroutine $\textsc{IsWord}(i, j)$ that takes indices i, j with $i \leq j$ as input and indicates whether $S[i, \ldots, j]$ is a "word" in the foreign language, and that it takes constant time to run.

Design an algorithm that solves this problem in $O(n^2)$ time.

Denote the input as an array $S[1, \ldots, n]$ of letters. Observe that, if S can be segmented, then the last letter in S is the last letter of some word, and the prefix of S without that word can also be segmented. In other words, there is some position j so that $\textsc{IsWord}$ indicates $S[j+1, \ldots, n]$ is a word, and a recursive computation on $S[1, \ldots, j]$ reveals that it can be segmented. We can, for all possible j, check directly whether the former condition holds, and check the latter condition by recursion. If any one of choice of j works, then S is segmentable; otherwise, it is not.

Over the course of such a computation, each recursive call to our procedure operates on a prefix of S. This suggests to define the values $\text{CanSeg}(i)$, $i = 0, \ldots, n$, where $\text{CanSeg}(i)$ is `True` or `False` according to whether $S[1, \ldots, i]$ can be segmented. Here, $S[1, \ldots, 0]$ denotes the empty string; we define it to be segmentable as this makes for a convenient base case below. The answer we seek is precisely $\text{CanSeg}(n)$. Following the above discussion, we can compute it using the following recurrence:

$$\text{CanSeg}(i) = \begin{cases} \texttt{True} & : i = 0 \\ \bigvee\limits_{0 \leq j < i} \text{CanSeg}(j) \wedge \textsc{IsWord}(j+1, i) & : i > 0 \end{cases}.$$

The $\vee$ represents a Boolean OR (`||` in Java), and $\wedge$ represents a Boolean AND (`&&` in Java).

This recurrence can be implemented via a recursive algorithm that is made efficient through the use of memoization.

It can also be made iterative. Computing $\text{CanSeg}(i)$ requires knowing $\text{CanSeg}(j)$ only for $j < i$. So starting from the base case of $i = 0$ and working up ensures that when we compute $\text{CanSeg}(i)$, all the required values of $\text{CanSeg}(j)$ have already been computed. Pseudocode for this iterative implementation is given in Algorithm 1.

Correctness Correctness essentially follows from the first paragraph above. More formally, we argue that for all inputs $S[1, \ldots, n]$ and indices $i = 0, \ldots, n$, the recurrence for $\text{CanSeg}(i)$ correctly computes the definition of $\text{CanSeg}(i)$. We do this by induction on i.

Base case: The base case is when $i = 0$. $\text{CanSeg}(0)$ is computed according to its definition.

Inductive step: In the inductive step, we have $i > 1$. $S[1, \ldots, i]$ can be segmented if and only if there is an index $0 \leq j < i$ so that $S[1, \ldots, j]$ is empty or can be segmented, and so that $S[j+1, \ldots, i]$ is a word. For each fixed j, the inductive hypothesis implies that $\text{CanSeg}(j) \wedge \textsc{IsWord}(j+1, i)$

Algorithm 1 Text Segmentation

```
Input: string $S[1,\dots,n]$, access to IsWord
Output: indicate whether $S$ can be segmented into words
1: procedure Segment($A$)
2:     CanSeg$[0,\dots,n] \leftarrow$ fresh array of Booleans
3:     CanSeg$[0] \leftarrow$ True
4:     for $i \leftarrow 1$ to $n$ do
5:         CanSeg$[i] \leftarrow \bigvee_{0\le j<i}$ CanSeg$[j] \wedge$ IsWord$(j+1,i)$
6:     return CanSeg$[n]$
```

correctly tests whether j satisfies that condition. Therefore, S can be segmented if and only if there is j so that the j-th term in the OR in the recurrence for CanSeg evaluates to `True`; that is, if and only if the OR evaluates to `True`. This establishes the inductive step.

That the recurrence for CanSeg(i) correctly computes its specification now follows by induction. This also proves that a recursive implementation with memoization is correct. Correctness of the iterative version, Algorithm 1, follows, because it fills in CanSeg$[i]$ = CanSeg(i) for $i = 0,\dots,n$ using the recurrence.

Time and space analysis There are $n+1 = O(n)$ values of CanSeg$(\cdot)$ to compute. Each requires examining $O(n)$ values of j, and for each value of j, the work is constant. The overall work done in the recursive implementation with memoization is therefore $O(n^2)$. The space is the number of calls to memoize, which is $O(n)$.

As for Algorithm 1, direct inspection reveals that it runs in $O(n^2)$ time and uses $O(n)$ space.

Problem 2

You are given an array $A[1,\dots,n]$ of integers and want to find the maximum sum of the elements of (a) any subsequence, and (b) any subarray. The sum of an empty subarray is 0. For the example above, the maximum-sum subarray and subsequence has length zero.

Design an $O(n)$ algorithm for both problems.

In the subsequence case, the maximum sum is exactly equal to the sum of all the positive elements in A. This can be computed easily in linear time with a single sweep through A.

As for the maximum-sum subarray, we take a dynamic programming approach. We find, for every position i in the array, the maximum sum of a subarray that is a suffix of $A[1,\dots,i]$ (including possibly the empty array). Since every subarray of A is the suffix of $A[1,\dots,i]$ for some i, taking the maximum of all those values gives the maximum sum among all subarrays of A.

Fix a position i. We partition the suffixes of $A[1,\dots,i]$ into two cases: each suffix is either the empty suffix (in which case its sum is 0), or $A[k,\dots,i]$ for some $k \le i$. In the latter case, $A[k,\dots,i]$ is a suffix of $A[1,\dots,i-1]$ plus $A[i]$; the maximum sum of such a subarray is $A[i]$ plus the maximum sum of a suffix of $A[1,\dots,i-1]$. The maximum of the two cases gives us the best value for subarrays ending at position i.

This reasoning implies that, given the maximum sum for subarrays that are suffixes of $A[1,\dots,i-1]$, we can compute the maximum sum of subarrays that are suffixes of $A[1,\dots,i]$ with a constant number of operations. We define the quantity OPT(i) for $1 \le i \le n$ as the maximum sum of a subarray of A that ends at index i, including possibly the empty array. OPT satisfies the following recurrence:

$$\text{OPT}(i) = \begin{cases} \max(0, A[1]) & : i = 1 \\ \max(0, A[i] + \text{OPT}(i-1)) & : i > 1 \end{cases}$$

The final output is then $\max_{1\le i\le n} \text{OPT}(i)$.

The recurrence can be computed via a recursive algorithm that is made efficient through memoization. The final output can be computed by a wrapper procedure, where the memoization table is re-used from one computation of OPT to the next.

OPT can also be computed iteratively. Computation of OPT(i) depends only on OPT$(i-1)$, so starting from the base case of $i = 1$ and working up ensures that, when we compute OPT(i), OPT$(i-1)$ has already been computed. The iterative algorithm moreover needs only to remember the most recently-computed value of OPT at a time, and it can compute $\max_{1\le i\le n} \text{OPT}(i)$ on the fly. This allows for a more economical use of space. Pseudocode for this iterative implementation is given in Algorithm 2.

Correctness Correctness essentially follows from the above discussion. Formally, we argue that, for all inputs $A[1,\dots,n]$ and indices $i = 1,\dots,n$, the recurrence for OPT(i) correctly computes the definition of OPT(i). We do this by induction on i.

Base case: The base case is when $i = 1$. There are only two subarrays that are suffixes of $A[1,\dots,1]$: their sums are 0 and $A[1]$. Thus OPT(1) is the maximum of these two.

Inductive step: For the inductive step, $i > 1$. Every subarray that is a suffix of $A[1,\dots,i]$ is either empty with sum zero or, for some k, the concatenation of a subarray $A[k,\dots,i-1]$ with $A[i]$. In the latter case, the sum of the subarray is maximized by maximizing the sum of the

Algorithm 2 Maximum Sum Subarray

```
Input: array $A[1...n]$ of integers
Output: maximum sum of a subarray of $A$
1: procedure MaximumSumSubarray($A$)
2:     OPT $\leftarrow \max(0, A[1])$
3:     $M \leftarrow$ OPT
4:     for $i \leftarrow 2$ to $n$ do
5:         OPT $\leftarrow \max(0, A[i] +$ OPT$)$
6:         $M \leftarrow \max(M,$ OPT$)$
7:     return $M$
```

subarray that is a suffix of $A[1,\dots,i-1]$. By the inductive hypothesis, the maximum sum of such a subarray is OPT$(i-1)$. It follows that OPT(i) is the larger of 0 and OPT$(i-1)+A[i]$.

That OPT(i) correctly computes its definition now follows by induction. This also proves that a recursive implementation with memoization is correct. As for correctness of Algorithm 2, it can be verified that before the for loop, OPT stores OPT(1) and M stores OPT(1), and that after iteration i, OPT stores OPT(i) and M stores the maximum among OPT$(1),\dots,$OPT(i). It follows that the returned value of M coincides with the correct output.

Time and space analysis There are n entries of OPT to compute. Each requires only a constant number of operations. So a recursive implementation with memoization would require overall linear time. It uses linear space for the memoization table.

As for Algorithm 2, direct inspection reveals it runs in linear time and uses only constant space.

Homework 1 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroieski

Problem 3

The library has n books that must be stored in alphabetical order on adjustable-height shelves. The i-th book has height $h[i]$ and thickness $t[i]$, $i \in [n]$. The width of the shelf is fixed at w, and the sum of the thicknesses of books on a single shelf cannot exceed w. The next shelf will be placed atop the tallest book on the shelf. You can assume the shelving takes no vertical space.

Design an algorithm that minimizes the total height of shelves used to store all the books. You are given the list of books in alphabetical order. Your algorithm should run in time $O(n^2)$.

The input consist of n books $b_1,\dots,b_n$, where b_i has height $h[i]$ and thickness $t[i]$. Consider the possibilities for the last level of books. Since we have to put the books on the shelf in order, the last level must consist of a suffix of the books, i.e., $b_i,\dots,b_n$ for some $i \in [n]$. The books have to fit on the shelf, so the choice of i has to satisfy $\sum_{i\le j\le n} t[j] \le w$. Any choice of i satisfying that constraint is a valid possibility.

As for which choice of i is the best, first note that if all the books fit on one shelf (i.e., the case $i = 1$ above), then doing that is the optimal way to shelve the books. This is because the total height has to be at least the height of the tallest book, and this limit is attained with all the books on the same shelf.

When not all the books fit on one shelf, the best choice of i is less clear. We can observe that for a fixed choice of i, the height of the last level is fixed also: it is $\max_{i\le j\le n} h[j]$. So among all shelvings of books that place $b_i,\dots,b_n$ on the last shelf, the best one minimizes the height of shelving for books $b_1,\dots,b_{i-1}$, and then places the last shelf atop that. Minimizing the height of shelving for $b_1,\dots,b_{i-1}$ is a smaller instance of the same problem, so we can find a solution recursively. In this way, we can find for each i the minimum-height shelving among those that place books b_i through b_n on the last level. The overall best shelving for $b_1,\dots,b_n$ can be found by choosing for i the best among the valid possibilities

Subproblems and recurrence This intuition gives us a recursive solution. The subproblems handled by recursive calls are parametrized by an index ℓ, $1 \le \ell \le n$, where the ℓ-th subproblem is to minimize the total height of shelving for $b_1,\dots,b_\ell$. Let $OPT(\ell)$ denote this minimum total height; we want to know $OPT(n)$. Per the above discussion, we can compute $OPT(n)$ using the following recurrence:

$$OPT(\ell) = \begin{cases} \max\limits_{1\le j\le \ell} h[j] & : \text{books 1 through } \ell \text{ fit on one shelf} \\ \min\limits_{i\in S_\ell}\left(OPT(i-1) + \max\limits_{i\le j\le \ell} h[j]\right) & : \text{otherwise} \end{cases}$$

where S_ℓ denotes the set of indices $i \le \ell$ such that books $b_i,\dots,b_\ell$ can fit on one shelf.

This recurrence can be computed by a recursive algorithm. As the number of possibilities for ℓ is small (only n), memoization will make this recursive procedure efficient. A naïve evaluation of the recurrence performs $\Theta(n^2)$ local work giving $\Theta(n^3)$ running time overall; however, with some care (see the paragraph after the next), the work local to a recursive call can be done in $O(n)$ time. This leads to an overall $O(n^2)$ running time.

We can also compute the recurrence iteratively. Computation of $OPT(\ell)$ depends only on knowledge of $OPT(i)$ for $i < \ell$, so starting from $\ell = 1$ and working up ensures that, when we compute $OPT(\ell)$, all requisite $OPT(i)$ have already been computed.

It remains to say how to compute $OPT(\ell)$ in $O(n)$ time given $OPT(i)$ for $i < \ell$. $OPT(\ell)$ is the minimum of $OPT(i-1) + \max_{i \le j \le \ell} h[j]$ as i ranges through those where $\sum_{i \le j \le \ell} t[j] \le w$. We iterate through the choices of i, starting with $i = \ell$ and working downward. Along the way, we maintain the values $h = \max_{i \le j \le \ell} h[j]$ and $t = \sum_{i \le j \le \ell} t[j]$. As long as $t \le w$, i is in S_ℓ. $OPT(\ell)$ is the minimum value of $OPT(i-1) + h$ during the iteration. We can compute the initial values for h and t in constant time, since when $i = \ell$, we have $h = h[\ell]$ and $t = t[\ell]$. We can also update h and t from one value of i to the next in constant time by using the update rules $h \leftarrow \max(h[i], h)$ and $t \leftarrow t[i] + t$. As there are at most n values of i to try, there are at most n updates, so the total work done is $O(n)$, as desired.

For clarity, pseudocode, including the $O(n)$-time way to compute $OPT(\ell)$, is given in Algorithm 1.

Correctness Correctness essentially follows from the above discussion. More formally, we prove for all ℓ that the recurrence for $OPT(\ell)$ correctly computes its definition. We do this by induction on ℓ.

Base case: The base cases are when books $b_1, \ldots, b_\ell$ fit onto one shelf. As discussed above, the minimum-height shelving is to put all the books on one shelf, in which case the height is the height of the tallest book.

Inductive step: The inductive step needs only address when books $b_1, \ldots, b_\ell$ do not fit onto one shelf. Given this, for every feasible shelving, there is some index $i \in S_\ell$ so that books $b_i, \ldots, b_\ell$ go onto the top shelf, and books $b_1, \ldots, b_{i-1}$ are shelved optimally beneath them. The height of the shelf for $b_i, \ldots, b_\ell$ is the largest height of those books. Also, the minimum-height of a shelving for $b_1, \ldots, b_{i-1}$ is $OPT(i-1)$, by the inductive hypothesis. So for each $i \in S_\ell$, the optimal shelving placing books $b_i, \ldots, b_\ell$ on the top shelf has height $OPT(i-1) + \max_{i \le j \le \ell} h[j]$. As this accounts for all feasible shelvings, choosing $i \in S_\ell$ to minimize this quantity correctly minimizes the height of shelving for books $b_1, \ldots, b_\ell$. This proves the inductive step.

That the recurrence for $OPT(\cdot)$ correctly computes the definition of $OPT(\cdot)$ now follows by induction. This also proves that a recursive implementation with memoization is correct.

Correctness of the iterative version, Algorithm 1, follows, because it fills in $OPT[\ell] = OPT(\ell)$ in an order such that the values $OPT[i]$ needed to evaluate the recurrence for $OPT(\ell)$ have already been computed before they are needed.

Time and space analysis There are n values of OPT to compute. The local work required to compute some $OPT(\ell)$ is $O(n)$. So the overall work done by a recursive implementation with memoization is $O(n^2)$. It uses $O(n)$ space for the memoization table.

Algorithm 1

Input: arrays $h[1 \ldots n]$ and $t[1 \ldots n]$ of positive integers denoting the height and thickness of each book; a positive integer w denoting the width of the shelf

Output: the minimum height of a shelving for the books

```
procedure ComputeMinimumHeight($h[1 \ldots n]$, $t[1 \ldots n]$, $w$)
    $OPT[1 \ldots n] \leftarrow$ fresh array
    Here we fill in the base cases
    $\ell \leftarrow 1$
    $t \leftarrow t[1]$                         ▷ $t$ tracks the thickness of the current shelf
    $h \leftarrow h[1]$                         ▷ $h$ tracks the maximum height of the current shelf
    $OPT[1] \leftarrow h$
    while $\ell + 1 \le n$ and $t + t[\ell + 1] \le w$ do
        $\ell \leftarrow \ell + 1$
        $t \leftarrow t + t[\ell]$
        $h \leftarrow \max(h, h[\ell])$
        $OPT[\ell] \leftarrow h$
    Here we compute the recursive cases
    while $\ell + 1 \le n$ do
        $\ell \leftarrow \ell + 1$
        $i \leftarrow \ell$                     ▷ First consider $i = \ell \ldots$
        $t \leftarrow t_i$                      ▷ $t = \sum_{i \le j \le \ell} t[j]$
        $h \leftarrow h[i]$                     ▷ $h = \max_{i \le j \le \ell} h[j]$
        $OPT[\ell] \leftarrow OPT[i-1] + h$
        while $t + t[i-1] \le w$ do             ▷ ...then consider smaller $i$ until books don't fit
            $i \leftarrow i - 1$
            $t \leftarrow t[i] + t$
            $h \leftarrow \max(h[i], h)$
            $OPT[\ell] \leftarrow \min(OPT[\ell], OPT[i-1] + h)$
    Here we return the final answer
    return $OPT[n]$
```

As for the iterative implementation in Algorithm 1, inspection reveals that each while loop makes at most n iterations. They are nested two deep, so the running time is at most $O(n^2)$. Space usage is dominated by the array OPT, and it takes $O(n)$ space.

Alternate Solution

A natural way to approach the problem is to try keeping track of the remaining space in the last shelf in addition to a prefix of the books that we want to organize. Note, however, that since the remaining space is a real number, we can't directly keep track of it in the indices of the subproblem specification. To deal with this, we have the subproblem specification keep track of the first book (and thus of all books) that are in the last shelf, as this gives us enough information to compute the remaining space. This idea leads to a two-dimensional subproblem specification, that we define next.

Subproblems and recurrence For $1 \le i \le j \le n$, $OPT(i, j)$ is the minimum height required to shelve books $1, \ldots, j$ given that the last shelf contains books $i, \ldots, j$. If this is not possible, then $OPT(i, j) = \infty$. Given this subproblem specification, the final answer that the algorithm needs to compute is $\min_{1 \le i \le n}(\text{OPT}(i, n))$.

Now, how do we compute $OPT(i, j)$? Let us start with the base case. If $j = 1$, then it must be the case that $i = 1$ and we have $OPT(1, 1) = h[1]$ by shelving the single book. For $i < j$, we can use the value of $OPT(i, j-1)$ to help: We check whether book j would fit together with the others on the last shelf. If not, then the answer is ∞, otherwise, the answer will be $OPT(i, j-1)$ plus whatever extra height we get from book j (which may be 0 if it is shorter than the others). To write out the recurrence, we introduce some notation that not only simplifies the expression but ends up being necessary to obtain an efficient solution (we get into more detail about this later). Let $W[\ell, k]$ be the sum of the widths of books $\ell, \ldots, k$ for $1 \le \ell \le k \le n$ and $H[\ell, k]$ be the maximum height among books $\ell, \ldots, k$, again for $1 \le \ell \le k \le n$. Then, our recurrence for $i < j$ is

$$OPT(i, j) = \begin{cases} OPT(i, j-1) + (H[i, j] - H[i, j-1]) & \text{if } W[i, j] \le w, \\ \infty & \text{otherwise.} \end{cases}$$

What about $OPT(i, i)$? In this case, we know that we get a contribution of $h[i]$ from the i-th book, but then we need to shelve the remaining $i - 1$ books optimally as well, and to do this we need to try all possibilities for the first book on the last shelf. This leads to the following recurrence for $2 \le i \le n$

$$OPT(i, i) = h[i] + \min_{1 \le \ell \le i-1} (OPT(\ell, i-1))$$

Notice that, by setting $OPT(i, j) = \infty$ when books $i, \ldots, j$ don't fit on the same shelf, we don't need to worry about only considering values of ℓ such that the books $\ell, \ldots, i-1$ fit in a single shelf. Correctness essentially follows from the discussion above, and we leave a formal proof of correctness as an exercise.

Time and space analysis Analyzing the running time for this recurrence is a little trickier since it has two different formats depending on the values of i and j. Notice, however, that we have $O(n^2)$ subproblems, so to get a running time of $O(n^2)$ we need to be able to compute individual entries in H and W in constant time, which seems hard to do. However, we can use another layer of dynamic programming to precompute all of those values in time $O(n^2)$. For table W, the base cases are $W[i, i] = w[i]$ for all i. For $i < j$, the recurrence is then

$$W[i, j] = W[i, j-1] + w[j],$$

which leads to $O(1)$ time per entry and a total of $O(n^2)$ for computing every entry. For H, we do something similar. $H[i, i] = h[i]$ for every i and

$$H[i, j] = \max(H[i, j-1], h[j])$$

for $i < j$. This again leads to $O(1)$ time per entry and a total of $O(n^2)$ for computing every entry.

Finally, computing $OPT[i, j]$ for $i < j$ takes time $O(1)$ per entry, while computing $OPT[i, i]$ takes time $O(n)$ per entry. Thankfully, there are only $O(n)$ entries of the second type, so the final running time is still $O(n^2)$.

As for the space complexity, a recursive solution with memoization requires space $O(n^2)$ for storing the OPT table (as well as tables H and W). Notice, however, that to compute column j of OPT we only need to know the values of OPT at column $j - 1$. Therefore, an iterative implementation that only keeps two columns in memory improves the space complexity of computing OPT to $O(n)$, but we would still require space $O(n^2)$ to store H and W. To deal with this, we use a similar strategy: To compute OPT for a column j, we just need to know the values of H and W at column j, and these only depend on column $j - 1$ of the respective table. Therefore, we can just keep two columns of H and W in memory and update these along the way, together with OPT. This leads to an implementation with space complexity $O(n)$.

Observation Note that, at its core, this solution ends up being almost the same as the previous one: All of the actual decisions are made when computing the entries $OPT[i, i]$, which essentially amount to computing $OPT[i-1] + h[i]$ (where here OPT refers to the previous solution).

Problem 4

When you were little, every day on your way home from school you passed the house of your grandmother. If you stop by for a chat on day i, Grandma would give you a number $\ell[i]$ of lollipops but also tell you that she won't give you any more lollipops for the next $k[i]$ days. For example, if day 1 is a Monday and $k[1] = 3$, then if you visit her that day, you would have to wait patiently until Friday to get your next lollipop.

Design an $O(n)$ algorithm that takes as input the arrays $\ell[1 \dots n]$ and $k[1 \dots n]$, and outputs the maximum number of lollipops you can get during those n days.

We can reduce an instance of this problem to an easier instance of the same problem by considering the first decision we need to make: Do we get lollipops on the first day or not? If we do, then we get $\ell[1]$ lollipops the first day, and we additionally need to find the maximum number of lollipops we can get during days $1 + k[1] + 1$ through n. Otherwise, we do not get any lollipops the first day, and need to find the maximum number of lollipops we can get during days 2 through n. Overall, the maximum number of lollipops we can get during days 1 through n is the maximum of the two possibilities.

Applying this idea recursively leads to subproblems of the following form: For $1 \leq i \leq n$, $\mathrm{OPT}(i)$ denotes the maximum number of lollipops we can get during days i through n. The above discussion yields the following recurrence:

$$\mathrm{OPT}(i) = \max \left\{ \ell[i] + \mathrm{OPT}(i + k[i] + 1), \mathrm{OPT}(i + 1) \right\},$$

where $\mathrm{OPT}(i) = 0$ for $i > n$ are convenient base cases. We use the recurrence to compute $\mathrm{OPT}(i)$ for $i = n, n-1, \dots, 1$, and return $\mathrm{OPT}(1)$.

Correctness To prove correctness of the recursive case of the recurrence for OPT, one divides the possible lollipop-acquisition strategies into those that take the lollipops on day i, and those that do not. The cases correspond to the two terms in the max above. We leave the remaining details of a formal proof by induction on i as an exercise.

Time and space analysis There are $O(n)$ subproblems, and each update takes constant time and space. Therefore the total running time is $O(n)$, and the total space is $O(n)$.

Alternate solution As an alternate solution, we can efficiently reduce this problem to weighted interval scheduling. There is one interval for every day i, $1 \leq i \leq n$. The interval corresponding to day i is $[i, i + k[i]]$ and has weight $\ell[i]$. With this setup, a valid selection of days on which we get lollipops corresponds to a valid interval schedule, and vice versa. Moreover, the total number of lollipops we get equals the weight of the intervals scheduled.

Note that, apart from the initial sorting phase and the construction of the table p of predecessors, the weighted interval scheduling algorithm from class takes time $O(n)$. Also, while we sorted the intervals by nondecreasing end time and went over them from back to front, we could as well sort them by nondecreasing start time and go over them from the front to the back. (This is like reverting the direction of the time axis.) Since we are given the intervals sorted by their start time, we do

the latter as it obviates the need for the initial sorting. Moreover, we have that $p(i) = i + k[i] + 1$, so the table p can be computed in time $O(n)$. With these provisos, the alternate solution also runs in $O(n)$ time.

Problem 5

You want to go running and have n minutes to spare. You want to run as long a distance as possible, but your exhaustion level cannot exceed a given limit e. Initially your exhaustion level is zero. During each minute, you can choose to either run or rest for the whole minute. When you choose to run the i-th minute, you run exactly $d[i]$ feet during that minute, and your exhaustion level increases by one. When you choose to rest, you run zero feet during that minute, and your exhaustion level decreases by one (if your exhaustion level is already zero, it will stay zero). Moreover, when you choose to rest, you must continue to rest until your exhaustion level reaches zero; once it reaches zero, you can again choose to run or rest. Finally, your exhaustion level at the end of your run must be zero.

Develop an algorithm that takes a positive integer e and an array $d[1, \dots, n]$ of $n \geq 1$ positive integers as input, and ouputs the maximum distance you can run subject to the constraints above. Your algorithm should run in time $O(ne)$ and space $O(e)$.

The problem statement stipulates that whenever we start running, we can do so for at most e minutes, after which we need to have an equal period of rest. The first decision we need to make is (i) whether to run or rest during minute 1, and if run, then (ii) for how many consecutive minutes $m \leq e$ to run before starting to rest. In the rest case, it remains to solve the same problem for the period starting from minute 2. In the run case, it remains to solve the same problem for the period starting from minute $2m + 1$ for each $m \in [e]$. We pick the choice that leads to the longest total distance run.

Subproblems and recurrence Recursive application leads to subproblems that correspond to suffixes of the given array $d[1, \dots, n]$. We define $\mathrm{OPT}(i)$ for $i \in [n]$ to be the solution to the given problem for exhaustion limit e and array $d[i, \dots, n]$, and define $\mathrm{OPT}(n+1) \doteq 0$ as a convenient base case. By the above reasoning, we have the recurrence

$$\mathrm{OPT}(i) = \max \left(\mathrm{OPT}(i+1), \max_{m \in [e], i+2m \leq n+1} \left(\mathrm{OPT}(i + 2m) + \sum_{j=i}^{i+m-1} d[j] \right) \right)$$

for $i = 1, \dots, n$ with $\mathrm{OPT}(n+1) \doteq 0$ as the base case.

Correctness We formalize the case-work from the first paragraph to prove, by induction on i (starting from $i = n + 1$ and working down), that the stated recurrence for $\mathrm{OPT}(i)$ computes the definition of $\mathrm{OPT}(i)$.

Base case: The base case is $i = n + 1$. $\mathrm{OPT}(n + 1)$ is defined to be zero and is computed as such.

Inductive step: For the inductive step, $i \leq n$. Every solution for the suffix $d[i, \dots, n]$ starts by deciding to run or rest at minute i.

Among solutions that start by resting, the best we can do is given by $\mathrm{OPT}(i+1)$: when $i < n$ this follows from the definition of $\mathrm{OPT}(i + 1)$; when $i = n$, resting at minute i concludes the run after running $0 \doteq \mathrm{OPT}(i + 1)$ distance.

Among solutions that start by running, each starts by running for some m minutes (minutes $i, \dots, i + m - 1$), then rests for m minutes (minutes $i + m, \dots, i + 2m - 1$), and then either

concludes (if $i + 2m = n + 1$) or else is a solution to the problem with suffix $d[i + 2m, \dots, n]$. Here m is always at most e, and moreover must satisfy $i + 2m \leq n + 1$ in order to finish the run with zero energy. For fixed m, the optimal distance covered is $\sum_{j=i}^{i+m-1} d[j]$ for the first stretch plus $\mathrm{OPT}(i + 2m)$ for the rest.

The maximum among all the possibilities (rest or run, and, if running, for how long) is therefore the optimal solution. This maximum is precisely what is computed by the recurrence.

Implementation and complexity We start with the entry $\mathrm{OPT}(n + 1) \doteq 0$, and apply the recurrence for $i = n$ down to $i = 1$, which gives us the final answer $\mathrm{OPT}(1)$. We evaluate the term for the run case by keeping track of a running sum s as in Algorithm 2. This way, the amount of work per cell is $O(e)$. As there are $O(n)$ cells, this gives us a time complexity of $O(ne)$.

Algorithm 2

```
$\mathrm{OPT}(i) \leftarrow \mathrm{OPT}(i+1)$
$s \leftarrow 0$
for $m \leftarrow 1, \dots, e$ do
    if $i + 2m \leq n + 1$ then
        $s \leftarrow s + d[i + m - 1]$
        $\mathrm{OPT}(i) \leftarrow \max(\mathrm{OPT}(i), \mathrm{OPT}(i + 2m) + s)$
```

Note that in order to apply the recurrence for $\mathrm{OPT}(i)$, we only need $\mathrm{OPT}(i+1)$ and $\mathrm{OPT}(i+2m)$ for $m \leq e$. Thus, it suffices to remember the last $2e$ cells computed. This gives a space complexity of $O(e)$.

Alternate solutions There are a number of natural alternate solutions:

- One may work with prefixes instead of suffixes of the array $d[1, \dots, n]$.
- One may consider the above binary decision of run-or-walk only instead of also deciding on the length m of the stretch to run. This idea can be developed by keeping track of two additional pieces of information, namely the exhaustion level at the start, and if the level is positive, whether we ended the previous minute by running or by resting. This results in two two-dimensional tables with dimensions $n \times e$, or one three-dimensional table with dimensions $n \times e \times 2$, and constant work per table entry. A similar approach works for prefixes instead of suffixes.
- Another way of developing this idea is to only keep track of the exhaustion level as additional information, but in the case where we decide to rest during minute i with exhaustion level j, to rest for j minutes and continue the process with minute $i+j$. This obviates the need for the above $n \times e$ table corresponding to ending the previous minute by resting, while maintaining constant work per table entry. A similar approach works for prefixes instead of suffixes.