CS 577: Introduction to Algorithms Fall 2020

Homework 3 Solutions

Instructor: Dieter van Melkebeek TA: Andrew Morgan

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:

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

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 size of the array used for memoization, which is $O(n)$.

As for Algorithm 1, it is simply some loops and basic statements (no recursion, subroutines, etc). 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 $\mathrm{OPT}(i)$ 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:

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

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

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 $\mathrm{OPT}(i)$ depends only on $\mathrm{OPT}(i-1)$, so starting from the base case of $i = 1$ and working up ensures that, when we compute $\mathrm{OPT}(i)$, $\mathrm{OPT}(i-1)$ 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 $\max_{1\le i\le n} \mathrm{OPT}(i)$ 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 $\mathrm{OPT}(i)$ correctly computes the definition of $\mathrm{OPT}(i)$. 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 $\mathrm{OPT}(1)$ 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 $\leftarrow \max(0, A[i] + \text{OPT})$
6:         $M \leftarrow \max(M, \text{OPT})$
7:     return $M$
```

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

That $\text{OPT}(i)$ 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, first observe that before the for loop, OPT stores $\text{OPT}(1)$ and M stores $\text{OPT}(1)$. With each iteration of the loop, OPT is updated using its recurrence and M is updated to incorporate the new value in a running maximum. By correctness of the recurrence, it follows that after each iteration $i = 2, \ldots, n$, OPT stores $\text{OPT}(i)$ and M stores the maximum among $\text{OPT}(1), \ldots, \text{OPT}(i)$. Therefore, when the algorithm returns M after the iteration for $i = n$, 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, it consists of a loop and basic statements (no recursion or subroutines). Direct inspection reveals it runs in linear time and uses only constant space.

Problem 3

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 $d[1, \ldots, n]$ 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 $O(e)$.

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 $d[1, \ldots, n]$. We define $\text{OPT}(i)$ for $i \in [n]$ to be the maximum distance one can run in minutes $i, \ldots, n$ starting minute i with exhaustion level 0, never exceeding exhaustion limit e, and ending minute n with exhaustion level 0. We define $\text{OPT}(n+1) \doteq 0$ as a convenient base case. By the above reasoning, we have the recurrence

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

for $i = 1, \ldots, n$ with $\text{OPT}(n+1) \doteq 0$ 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 $\text{OPT}(i)$ computes the definition of $\text{OPT}(i)$.

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

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

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

Among solutions that start by running, each starts by running for some m minutes (minutes $i, \ldots, i+m-1$), then rests for m minutes (minutes $i+m, \ldots, i+2m-1$), and then either concludes (if $i + 2m = n + 1$) or else follows some solution to the problem for minutes $i+2m, \ldots, n$. 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 3. 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 3

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

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 $d[1, \ldots, n]$.
- One may consider the above binary decision of run-or-rest 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.

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 ℓ_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 patiently wait until Friday to get your next lollipop.

Design an $O(n)$ algorithm that takes as input the numbers (ℓ_i, k_i) for $i \in \{1, 2, \dots, n\}$, 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 ℓ_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 ℓ_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

The library has n books that must be stored in alphabetical order on adjustable-height shelves. Each book has a height, and a thickness. 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 $b_1, \ldots, b_n$, 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., $b_i, \ldots, b_n$ for some $1 \leq i \leq n$. The books have to fit on the shelf, so the choice of i has to satisfy $\sum_{i \leq j \leq n} t_j \leq w$. 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 $\max_{i \leq j \leq n} h_j$. So among all shelvings of books that place $b_i, \ldots, b_n$ on the last shelf, the best one minimizes the height of shelving for books $b_1, \ldots, b_{i-1}$, and then places the last shelf atop that. Minimizing the height of shelving for $b_1, \ldots, b_{i-1}$ 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 $b_1, \ldots, b_n$ can be found by choosing for i the best among the valid possibilities

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

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

where S_ℓ denotes the set of indices $i \leq \ell$ such that books $b_i, \ldots, b_\ell$ 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 $H(\ell)$ depends only on knowledge of $H(i)$ for $i < \ell$, so starting from $\ell = 1$ and working up ensures that, when we compute $H(\ell)$, all requisite $H(i)$ have already been computed.

It remains to say how to compute $H(\ell)$ in $O(n)$ time given $H(i)$ for $i < \ell$. $H(\ell)$ is the minimum of $H(i-1) + \max_{i \leq j \leq \ell} h_j$ as i ranges through those where $\sum_{i \leq j \leq \ell} t_j \leq w$. 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. $H(\ell)$ is the minimum value of $H(i-1) + h$ 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 $H(\ell)$, is given in Algorithm 4.

Algorithm 4

Input: positive integers $h_1, \ldots, h_n$, $t_1, \ldots, t_n$ 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

```
 1: procedure ComputeMinimumHeight($h_1, \ldots, h_n$, $t_1, \ldots, t_n$, $w$)
 2:     $H[1, \ldots, n] \leftarrow$ fresh array
      Here we fill in the base cases
 3:     $\ell \leftarrow 1$
 4:     $t \leftarrow t_1$                               ▷ $t$ tracks the thickness of the current shelf
 5:     $h \leftarrow h_1$                               ▷ $h$ tracks the maximum height of the current shelf
 6:     $H[1] \leftarrow h$
 7:     while $\ell + 1 \le n$ and $t + t_{\ell+1} \le w$ do
 8:         $\ell \leftarrow \ell + 1$
 9:         $t \leftarrow t + t_\ell$
10:         $h \leftarrow \max(h, h_\ell)$
11:         $H[\ell] \leftarrow h$
      Here we compute the recursive cases
12:     while $\ell + 1 \le n$ do
13:         $\ell \leftarrow \ell + 1$
14:         $i \leftarrow \ell$                    ▷ First consider $i = \ell$ ...
15:         $t \leftarrow t_i$                     ▷ $t = \sum_{i \le j \le \ell} t_j$
16:         $h \leftarrow h_i$                     ▷ $h = \max_{i \le j \le \ell} h_j$
17:         $H[\ell] \leftarrow H[i-1] + h$
18:         while $t + t_{i-1} \le w$ do          ▷ ... then consider smaller $i$ until books don't fit
19:             $i \leftarrow i - 1$
20:             $t \leftarrow t_i + t$
21:             $h \leftarrow \max(h_i, h)$
22:             $H[\ell] \leftarrow \min(H[\ell], H[i-1] + h)$
      Here we return the final answer
23:     return $H[n]$
```

Correctness Correctness essentially follows from the above discussion. More formally, we prove for all ℓ that the recurrence for $H(\ell)$ 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 $H(i-1)$, 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 $H(i-1) + \max_{i \le j \le \ell} h_j$. 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 $H(\cdot)$ correctly computes the definition of $H(\cdot)$ now follows by induction. This also proves that a recursive implementation with memoization is correct.

Correctness of the iterative version, Algorithm 4, follows, because it fills in $H[\ell] = H(\ell)$ in an order such that the values $H[i]$ needed to evaluate the recurrence for $H(\ell)$ have already been computed before they are needed.

Time and space analysis There are n values of H to compute. The local work required to compute some $H(\ell)$ 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.

As for the iterative implementation in Algorithm 4, 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 H, and it takes $O(n)$ space.