CS 577: Introduction to Algorithms Fall 2019

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 IsWord(i, j) that takes indices i, 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.

We use dynamic programming. 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. This is to say there is some position j so that 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 one of them works, then S is segmentable.

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 CanSeg(i), $i = 0, \ldots, n$, where 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 CanSeg(n). Following the above discussion, we can compute it using the following recurrence:

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

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

Algorithm 1 Text Segmentation

Input: string $S[1, \ldots, n]$, access to IsWord
Output: indicate whether S can be segmented into words

```
1: procedure Segment(A)
2:     CanSeg[0, ..., n] ← fresh array of Booleans
3:     CanSeg[0] ← True
4:     for i ← 1 to n do
5:         CanSeg[i] ← ⋁_{0≤j<i} CanSeg[j] ∧ IsWord(j + 1, i)
6:     return CanSeg[n]
```

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 CanSeg(i) correctly computes the definition of CanSeg(i). We do this by induction on i.

Base case: The base case is when $i = 0$. CanSeg(0) is computed correctly by definition.

Inductive step: For the inductive step, $i > 1$. $S[1, \ldots, i]$ 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 CanSeg(j) $\wedge$ IsWord($j+1, i$) correctly tests whether j satisfies the requisite 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 CanSeg(i) correctly computes its definition 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 $i = 0, \ldots, n$.

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 $A[1, \ldots, n]$ 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.

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 in the array the maximum sum of a subarray that ends at that position, and then take the maximum of all those values. Consider position i. There are two possibilities for a best subarray (that is, one of maximal sum) that ends at position i: either it is empty (in which case the maximum sum is 0), or it is $A[k, \ldots, i]$ for some $k \leq i$. If the latter, then $A[k, \ldots, i-1]$ must be a best subarray that ends at $i-1$ since otherwise, replacing $A[k, \ldots, i-1]$ with a best subarray would increase the total sum of elements in $A[k, \ldots, i]$. The maximum of the two possibilities gives us the best value for subarrays ending at position i.

This suggests that given the best value for subarrays that end at $i-1$, we can compute the best value for a subarray that ends at i with a constant number of operations. We define the quantity $\mathrm{OPT}(i)$ for $1 \leq i \leq n$ as the maximum sum of a subarray of A that ends at index i. 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 $\max_{1 \leq i \leq n} \mathrm{OPT}(i)$.

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 $\max_{1 \leq i \leq n} \mathrm{OPT}(i)$ 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 $A[1, \ldots, n]$ and indices $i = 1, \ldots, n$, 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 $i = 1$, where the only subarrays ending at 1 are the empty subarray and $A[1]$, so the base case is correct.

Inductive step: For the inductive step, $i > 1$. Every subarray ending at i is either empty or the concatenation of a subarray $A[j, \ldots, i-1]$ for some j with $A[i]$. In the latter case, the sum of the subarray is maximized by maximizing the sum of the subarray ending at $i-1$. By 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 ← max(0, A[1])
3:     M ← OPT
4:     for i ← 2 to n do
5:         OPT ← max(0, A[i] + OPT)
6:         M ← max(M, OPT)
7:     return M
```

inductive hypothesis, the maximum sum of such a subarray is $\text{OPT}(i-1)$. It follows that the formula in the recursive case computes $\text{OPT}(i)$ correctly. This proves the inductive step.

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, it can be verified that before the for loop, OPT stores $\text{OPT}(1)$ and M stores $\text{OPT}(1)$, and that after iteration i, OPT stores $\text{OPT}(i)$ and M stores the maximum among $\text{OPT}(1), \ldots, \text{OPT}(i)$. 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.

Problem 3

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 $O(n^2)$.

We use dynamic programming. 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. With some care (see pseudocode below), the work local to a recursive call can be done in $O(n)$ time (a naïve implementation would take $\Theta(n^2)$ 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 $i = \ell$, $h = h_\ell$ 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 3.

Algorithm 3

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

procedure ComputeMinimumHeight($h_1, \ldots, h_n$, $t_1, \ldots, t_n$, w)
    $H[1, \ldots, n] \leftarrow$ 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
    $H[1] \leftarrow h$
    while $\ell + 1 \le n$ **and** $t + t_{\ell+1} \le w$ **do**
        $\ell \leftarrow \ell + 1$
        $t \leftarrow t + t_\ell$
        $h \leftarrow \max(h, h_\ell)$
        $H[\ell] \leftarrow h$
    Here we compute the recursive cases
    while $\ell + 1 \le n$ **do**
        $\ell \leftarrow \ell + 1$
        $i \leftarrow \ell$ ▷ First consider $i = \ell \ldots$
        $t \leftarrow t_i$ ▷ $t = \sum_{i \le j \le \ell} t_j$
        $h \leftarrow h_i$ ▷ $h = \max_{i \le j \le \ell} h_j$
        $H[\ell] \leftarrow H[i-1] + h$
        while $t + t_{i-1} \le w$ **do** ▷ . . . then consider smaller i until books don't fit
            $i \leftarrow i - 1$
            $t \leftarrow t_i + t$
            $h \leftarrow \max(h_i, h)$
            $H[\ell] \leftarrow \min(H[\ell], H[i-1] + h)$
    Here we return the final answer
    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 need 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 3, follows, because it fills in $H[i] = H(i)$ in a suitable order.

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 3, 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.

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 $i \in \{1, 2, \ldots, n\}$, 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 lollies on the first day or not? If we do, then we get ℓ_1 lollies the first day, and we additionally need to find the maximum number of lollies we can get during days $1 + k_1 + 1$ through n. Otherwise, we do not get any lollies the first day, and need to find the maximum number of lollies we can get during days 2 through n. Overall, the maximum number of lollies 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 lollies we can get during days i through n. The above discussion yields the following recurrence:

$$\mathrm{OPT}(i) = \max\Big\{\ell_i + \mathrm{OPT}(i + k_i + 1), \mathrm{OPT}(i + 1)\Big\},$$

where $\mathrm{OPT}(i) = 0$ for $i > n$. We use the recurrence to compute $\mathrm{OPT}(i)$ for $i = n, n-1, \ldots, 1$, 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 lollies corresponds to a valid interval schedule, and vice versa. Moreover, the total number of lollies 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

Your local amusement park released a hip new slide. It has three lanes, and riders can change lanes as they ride. At several heights along the slide, one of the lanes will light up if a rider is in that lane at that height. You and your friends decide to make a game out of it: a rider gets one point for each light-up spot they activate, but *loses* one point for each lane change. Scores can go negative. Highest score wins.

Design an algorithm that takes as input a description of the slide and a starting lane, and outputs the maximum points attainable when starting from that lane. The slide is described by an array $L[1, \ldots, n]$, where for each $i = 1, \ldots, n$, $L[i] \in \{1, 2, 3\}$ indicates which lane contains the i-th light-up spot along the slide when they are indexed top-to-bottom. The algorithm should run in $O(n)$ time.

We start by observing that any score-maximizing trip down the slide involves making no redundant lane changes. For example, it is pointless to switch from lane 1 to lane 2 and then back to lane 1 between an adjacent pair of light-up spots. Given that, we can model a trip down the slide as a sequence of choices: for each $i = 1, \ldots, n$, decide in which lane to be riding while at the height of the i-th light-up spot. The point value of such a trip is the number of light-up spots entered, minus a point for each lane change necessary to realize the plan.

From here, we use dynamic programming. Let $L[1, \ldots, n]$ be a description of the slide as in the problem statement, and let ℓ_0 be the starting lane. Every route rides through some lane ℓ while at the height of the first light-up spot. If that lane contains the light-up spot ($\ell = L[1]$), then we get a point; otherwise we do not. If that lane agrees with ℓ_0 ($\ell = \ell_0$), we lose no points; otherwise we lose a point. Once all that is accounted for, we are in lane ℓ riding toward the height of the second light-up spot. For each fixed choice of ℓ, maximizing one's score through light-up spots 2 through n is a smaller instance of the same problem we are trying to solve, so we can solve it recursively. We can use as a base case when there are no light-up spots left ($n = 0$), in which case the maximum possible score is zero.

This intuition gives us a recursive solution to the problem. The subproblems handled by recursive calls are parametrized by an index m, $0 \le m \le n$, and a lane ℓ, where the (m, ℓ)-th subproblem is to maximize the score obtained for riding through or by the light-up spots $m + 1, \ldots, n$ when starting from lane ℓ. Let $\mathrm{OPT}(m, \ell)$ denote this maximum attainable score; we want to know $\mathrm{OPT}(0, \ell_0)$. Following the above intuition, we can compute it using the following recurrence:

$$\mathrm{OPT}(m, \ell) = \begin{cases} 0 & : m = n \\ \max \left\{ \begin{array}{l} \delta(1, L[m+1]) - (1 - \delta(1, \ell)) + \mathrm{OPT}(m+1, 1) \\ \delta(2, L[m+1]) - (1 - \delta(2, \ell)) + \mathrm{OPT}(m+1, 2) \\ \delta(3, L[m+1]) - (1 - \delta(3, \ell)) + \mathrm{OPT}(m+1, 3) \end{array} \right\} & : m < n \end{cases}$$

where, $\delta(\ell_1, \ell_2)$ is 1 if $\ell_1 = \ell_2$, and 0 otherwise. Under this definition of δ, the term "$\delta(1, L[m])$" adds a point if and only if the m-th light-up spot is in lane 1, and the term "$-(1-\delta(1, \ell))$" subtracts a point if and only if the starting lane (ℓ) differs from lane 1. Collectively, these two terms account for the net change in points when switching into (or staying within) lane 1 when passing by or going through the $(m + 1)$-th light-up spot.

This recurrence can be computed by a recursive algorithm. As the number of possibilities for (m, ℓ) is small (only $O(n)$), memoization will make this recursive procedure efficient.

We can also compute the recurrence iteratively. Computation of $\mathrm{OPT}(m, \ell)$ depends only on the values of $\mathrm{OPT}(m+1, \ell')$ for various ℓ'. So as long as we start with $m = n$ and work downward, we can compute $\mathrm{OPT}(m, \ell)$ and know that all the requisite $\mathrm{OPT}(m+1, \ell')$ values have been computed. Also, we need only remember the values of $\mathrm{OPT}(m+1, \ell')$, not those of any larger m. This allows for a more economical use of space. Pseudocode for this implementation is given in Algorithm 4.

Algorithm 4 Slide Game

Input: array $L[1...n]$ of lanes, initial lane ℓ_0

Output: maximum score attainable from the slide game played on the slide described by L when starting in lane ℓ_0

1: **procedure** SLIDEGAME(L, ℓ_0)
2: $\quad \mathrm{OPT}_1, \mathrm{OPT}_2, \mathrm{OPT}_3 \leftarrow 0, 0, 0$
3: $\quad$ **for** $m \leftarrow n-1$ **down to** 0 **do**
4: $\quad\quad$ **for** $\ell \leftarrow 1, 2, 3$ **do**
5: $\quad\quad\quad \mathrm{NextOPT}_\ell \leftarrow \max \left\{ \begin{array}{l} \delta(1, L[m]) - (1 - \delta(1, \ell)) + \mathrm{OPT}_1 \\ \delta(2, L[m]) - (1 - \delta(2, \ell)) + \mathrm{OPT}_2 \\ \delta(3, L[m]) - (1 - \delta(3, \ell)) + \mathrm{OPT}_3 \end{array} \right\}$
6: $\quad\quad \mathrm{OPT}_1 \leftarrow \mathrm{NextOPT}_1$
7: $\quad\quad \mathrm{OPT}_2 \leftarrow \mathrm{NextOPT}_2$
8: $\quad\quad \mathrm{OPT}_3 \leftarrow \mathrm{NextOPT}_3$
9: $\quad$ **return** OPT_{ℓ_0}

Correctness Correctness essentially follows from the discussion above. More formally, we prove for all m, ℓ that the recurrence for $\mathrm{OPT}(m, \ell)$ correctly computes its definition. We do this by induction on $n - m$ (i.e., induction on m with $m = n$ as the base case, and proceeding to $m = n-1, n-2, \ldots$).

Base case: In the base case, $m = n$. For every ℓ, $\mathrm{OPT}(m, \ell)$ is computed correctly by definition.

Inductive step: For the inductive step, $m < n$. Let ℓ be arbitrary. As discussed above, the optimal route through light-up spots $m+1, \ldots, n$ starting at ℓ is in some lane ℓ' at the height of light-up spot $m+1$. If $\ell' = L[m+1]$, a point is awarded; otherwise, not. If $\ell' \neq \ell$, a point is deducted; otherwise, not. From there, it is an optimal route through light-up spots $m+2, \ldots, n$ starting from lane ℓ'. The optimal score of such a route is $\mathrm{OPT}(m+1, \ell')$ by the inductive hypothesis. Thus, for each fixed ℓ', the ℓ'-th line in the max in the recurrence computes the maximum attainable score through light-up spots $m+1, \ldots, n$ starting from lane ℓ, provided one goes through lane ℓ' for light-up spot $m+1$. The overall optimum is the best of these, as computed by the max. This establishes the inductive step.

That the recurrence of OPT computes according to the definition of OPT now follows by induction. This also establishes correctness of a recursive implementation of the recurrence with memoization.

Correctness of the iterative implementation follows now from the fact that, after line 2, OPT_ℓ stores $\mathrm{OPT}(n, \ell)$ for each ℓ, and after iteration m, OPT_ℓ stores $\mathrm{OPT}(m, \ell)$ for each ℓ.

Time and space analysis A recursive implementation with memoization requires storage for each choice of m, ℓ. There are $O(n)$ such choices. The local work required to compute $\mathrm{OPT}(m, \ell)$ is constant. Thus the overall running time of a recursive implementation is $O(n)$. The space usage is also $O(n)$.

As for Algorithm 4, direct inspection reveals it runs in $O(n)$ time and uses $O(1)$ space.