Exercise

Please do the exercises on your own.

1. (8 pt.) Let $A$ be an array of length $n$ containing real numbers. A longest increasing subsequence (LIS) of $A$ is a sequence $0 \leq i_1 < i_2 < \ldots < i_d < n$ so that $A[i_1] < A[i_2] < \cdots < A[i_d]$, so that $d$ is as long as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then a LIS is $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 6$ corresponding to the subsequence $3, 5, 6, 8$. (Notice that a longest increasing subsequence doesn’t need to be unique).

In the following parts, we’ll walk through the recipe that we saw in class for coming up with DP algorithms to develop an $O(n^2)$-time algorithm for finding an LIS.

(a) (2 pt.) (Identify optimal sub-structure and a recursive relationship). We’ll come up with the sub-problems and recursive relationship for you, although you will have to justify it.

Let $D[i]$ be the length of the longest increasing subsequence of $[A[0], \ldots, A[i]]$ that ends on $A[i]$. Explain why

$$D[i] = \max \{\{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\} \cup \{1\}\}.$$  

[We are expecting: A short informal explanation (a paragraph or so). It might be good practice to write a formal proof, but this is not required for credit.]

(b) (3 pt.) (Develop a DP algorithm to find the value of the optimal solution) Use the relationship about to design a dynamic programming algorithm returns the length of the longest increasing subsequence. Your algorithm should run in time $O(n^2)$ and should fill in the array $D$ defined above.

[We are expecting: Pseudocode. No justification is required.]

(c) (3 pt.) (Adapt your DP algorithm to return the optimal solution) Adapt your algorithm above to return an actual LIS instead of its length. Your algorithm should run in time $O(n^2)$.

[We are expecting: Pseudocode AND a short English explanation of what your algorithm is doing. You do not need to justify that it is correct.]

Note: Actually, there is an $O(n \log(n))$-time algorithm to find an LIS, which is faster than the DP solution in this exercise!

SOLUTION:

(a) Soln. 1. Even though a formal proof is not required for credit, we include one below for completeness: We will first show that $D[i] \leq \max \{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\}$, for all $i \geq 1$.

Since $D[i]$ is the length of a LIS that ends on $A[i]$, let $i_1 < \ldots < i_{D[i]}$ be that subsequence; notice that $i_{D[i]} = i$, since the sequence ends on $A[i]$. Let $\hat{k} = i_{D[i] - 1}$, and consider the sequence $i_1 < i_2 < \ldots, i_{D[i] - 1}$ of length $D[i] - 1$ ending at $\hat{k}$. This is an increasing subsequence ending at $\hat{k}$. Thus, $D[i] - 1$ is at most $D[\hat{k}]$, since $D[\hat{k}]$ is the length of the longest such sequence, and this one had length $D[i] - 1$.

Since $\hat{k} = i_{D[i] - 1} < i_{D[i]} = i$ and $A[\hat{k}] < A[i]$ (since the original sequence was increasing) this only gets larger when we maximize over all such $k$, so Therefore

$$D[i] - 1 \leq D[\hat{k}] \leq \max \{D[k] : 0 \leq k < i, A[k] < A[i]\}.$$
Now we go the other direction and show that \( D[i] \geq \max\{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\} \). Suppose that \( k \) is the index which maximizes the right hand side, and suppose that

\[ i_1 < \ldots < i_D[k] \]

is the corresponding sequence. Then the sequence

\[ i_1 < \ldots < i_D[k] < i \]

is an increasing subsequence ending at \( i \), with length \( D[k] + 1 \). The length of the longest one is at least as long as this, so

\[ D[i] \geq D[k] + 1 = \max\{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\} \].

**Soln. 2.** The following solution would be an acceptable informal solution:

Suppose that \( D[i] = t \), for \( t > 1 \), so there is some subsequence indexed by \( i_1 < i_2 < \cdots < i_t = i \) that is an LIS ending in position \( i \). Now consider the subsequence \( i_1 < i_2 < \cdots < i_{t-1} \). This is the longest subsequence that ends at \( k = i_{t-1} \) — otherwise, if there were a longer subsequence ending at \( k \), we could make a longer solution ending at \( i \) just by adding \( A[i] \) to that subsequence. Thus, \( D[k] = t - 1 \) for some \( k \geq 0 \), so there is some \( k \) so that \( D[k] + 1 = t = D[i] \). Since \( D[i] \) is the value of the longest increasing subsequence ending at \( i \), we should take the maximum over all possibilities for \( k \), and this gives us

\[ D[i] = \max_{k<i} \{D[k] + 1\} \].

However, the above was only in the case that \( t > 1 \), so in order to account for the possibility that \( t = 1 \) (i.e., \( A[j] > A[i] \) for all \( j < i \)), we also add a 1 into the maximum, and this results in the expression on the problem statement.

(b) Our algorithm is as follows:

```python
def LIS(array A of length n):
    Initialize an array D of length n full of 1's
    for i = 1,...,n-1:
        for k in 0,...,i-1:
            if A[k] < A[i]:
                D[i] = max( D[i], D[k] + 1 )
    return max(D)
```

(c) Our algorithm is as follows. We modify our answer from part (b) to additionally store an array \( P \). If \( P[i] = k \), the interpretation is that the LIS ending at \( i \) had its second-to-last entry at \( k \). After we are done computing the length of the LIS, we can follow the pointers in \( P \) backwards to recover the LIS itself.

```python
def LIS(array A of length n):
    Initialize an array D of length n full of 1's
    Initialize an array P of length n full of None's
    for i = 1,...,n-1:
        for k in 0,...,i-1:
            if A[k] < A[i]:
                if D[i] < D[k] + 1:
                    D[i] = D[k] + 1
                    P[i] = k
    Find k so that D[k] is maximized
    ret = []
    current = k
    while current != None:
        ret.append(current)
        current = P[current]
    return ret
```
ret.append(A[current])
current = P[current]
reverse ret
return ret
2. (7 pt.) [MinElementSum.] Consider the following problem, MinElementSum.

MinElementSum(n, S): Let S be a set of positive integers, and let n be a non-negative integer. Find the minimal number of elements of S needed to write n as a sum of elements of S (possibly with repetitions). If there is no way to write n as a sum of elements of S, return None.

For example, if S = {1, 4, 7} and n = 10, then we can write n = 1 + 1 + 1 + 7 and that uses four elements of S. The solution to the problem would be “4.” On the other hand if S = {4, 7} and n = 10, then the solution to the problem would be “None,” because there is no way to make 10 out of 4 and 7.

Your friend has devised a divide-and-conquer algorithm to solve MinElementSum. Their pseudocode is below.

```python
def minElementSum(n, S):
    if n == 0:
        return 0
    if n < min(S):
        return None
    candidates = []
    for s in S:
        cand = minElementSum( n-s, S )
        if cand is not None:
            candidates.append( cand + 1 )
    if len(candidates) == 0:
        return None
    return min(candidates)
```

Your friend’s algorithm correctly solves MinElementSum. Before you start doing the problems on the next page, it would be a good idea to walk through the algorithm and to understand what this algorithm is doing and why it works.

[Questions on next page]
(a) (1 pt.) Argue that for $S = \{1, 2\}$, your friend’s algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$). You may use any statement that we have seen in class.

[HINT: Consider the example of the Fibonacci numbers that we saw in class.]

[We are expecting:
• A recurrence relation that the running time of your friend’s algorithm satisfies when $S = \{1, 2\}$.
• A convincing argument that the closed form for this expression is $2^{\Omega(n)}$. You do not need to write a formal proof.
]

(b) (3 pt.) Turn your friend’s algorithm into a top-down dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[HINT: Add an array to the pseudocode above to prevent it from solving the same sub-problem repeatedly.]

[We are expecting:
• Pseudocode AND a short English description of the idea of your algorithm.
• An informal justification of the running time.
]

(c) (3 pt.) Turn your friend’s algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[HINT: Fill in the array you used in part (b) iteratively, from the bottom up.]

[We are expecting:
• Pseudocode AND a short English description of the idea of your algorithm.
• An informal justification of the running time.
]

SOLUTION:

(a) When $S = \{1, 2\}$, then the algorithm running on $n$ makes two recursive calls, one to $n - 1$ and one to $n - 2$. The running time of this algorithm satisfies the recurrence

$$T(n) = T(n - 1) + T(n - 2) + O(1) \geq T(n - 1) + T(n - 2),$$

since the $O(1)$ term is non-negative. Thus, the running time of this algorithm grows at least as fast as the Fibonacci numbers. We saw in class that these grow exponentially quickly.

To see this from first principles, we can also write

$$T(n) \geq T(n - 1) + T(n - 2) + O(1) \geq 2T(n - 2),$$

using the fact that $T$ is increasing and the $O(1)$ term is positive. Let

$$C = 2^{-1/2} \cdot \min\{T(1), T(2)\},$$

so $C > 0$ is some constant. Then we use the substitution method with the inductive hypothesis that $T(n) \geq C \cdot 2^{n/4}$. For the base case, we have by our choice of $C$ that $T(1) \geq C \cdot 2^{1/2} \geq C \cdot 2^{1/4}$ and $T(2) \geq C \cdot 2^{1/2}$. Then for the inductive step, assume that $n > 2$ and that the inductive hypothesis holds for all positive integers less than $n$. Then we have

$$T(n) \geq 2T(n - 2) \geq 2 \cdot C \cdot 2^{(n-2)/4} \geq C \cdot 2^{n/4},$$

so this establishes the inductive hypothesis for $n$. We conclude that $T(n) \geq C \cdot 2^{n/4} = 2^{n/4 + \log_2(C)}$ for all sufficiently large $n$, which means that $T(n) = 2^{\Omega(n)}$, as desired. (Note, a formal proof by induction is not required for credit on this problem).
(b) To make a top-down DP algorithm, we add an array $D$ to keep track of previous solutions. We set $D[k]$ to be the minimal number of elements of $S$ needed to make $k$.

Initialize a global array $D$ of length $n+1$ to all -1’s.

$D[0] = 0$

Set $D[k] = \text{None}$ for all $0 < k < \min(S)$.

```python
def minimumElements(n, S):
    if n < 0:
        return None
    if D[n] != -1:
        return D[n]
    candidates = []
    for s in S:
        cand = minimumElements( n-s, S )
        if cand is not None:
            candidates.append( cand + 1 )
    if len(candidates) == 0:
        D[n] = None
    else:
        D[n] = min(candidates)
    return D[n]
```

The running time of this algorithm is $O(n|S|)$. Each $k \leq n$ gets run at most once, and for each one, we have work $O(|S|)$, to make the recursive calls and aggregate the results.

(c) To make a bottom-up DP algorithm, we’ll use the same interpretation of the array as before, but fill it in in order.

Initialize a global array $D$ of length $n$ to all -1’s.

$D[0] = 0$

Set $D[k] = \text{None}$ for all $0 < k < \min(S)$.

```python
for i = \min(S), \ldots, n:
    candidates = []
    for s in S:
        if i < s:
            continue
        cand = D[i-s]
        if cand is not None:
            candidates.append( cand + 1 )
    D[i] = min(candidates)
```

Again the running time is $O(n|S|)$, since the outer for loop has $n$ iterations, and the inner for loop has $|S|$ iterations. Inside the inner loop we do $O(1)$ work.
3. (6 pt.) [Rotten Tomatoes.] You are planting tomato plants in a garden, and the garden has \( n \) spots arranged in a line. Different spots in the garden will result in different quality tomatoes: suppose that the location \( i \) will result in tomatoes of deliciousness \( T[i] \), where \( T[i] \) is a positive integer. Further, you cannot plant two plants directly next to each other, because they will compete for resources and wilt. Your goal is to create the most deliciousness possible (summed up over all of the tomato plants).

**For example**, if the input was \( T = [21, 4, 6, 20, 2, 5] \), then you should plant tomatoes in the pattern

![Tomato planting pattern example](image)

and you would obtain deliciousness \( 21 + 20 + 5 = 46 \). You would **not** be allowed to plant tomatoes in the pattern

![Tomato planting pattern disallowed](image)

because there are two tomato plants next to each other.

In this question, you will design a dynamic programming algorithm which runs in time \( O(n) \) which takes as input the array \( T \) and returns the maximum deliciousness possible given \( T \). Do this by answering the two parts below.

(a) (3 pt.) What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between the sub-problems?

**We are expecting:**

- A clear description of your sub-problems.
- A recursive relationship that they satisfy, along with a base case.
- An informal justification that the recursive relationship is correct.

(b) (3 pt.) Write pseudocode for your algorithm. Your algorithm should take as input the array \( T \), and return a single number which is the maximum amount of deliciousness possible. Your algorithm does not need to output the optimal way to plant the tomatoes.

**We are expecting:** Pseudocode **AND** a clear English description. You do not need to justify that your algorithm is correct, but correctness should follow from your reasoning in part (a).

**SOLUTION:**

(a) Our sub-problems will be:

\[ A[i] = \text{the most deliciousness you can get from the first } i \text{ positions of the garden.} \]

The recursive relationship they satisfy is:

\[ A[i] = \max \{ A[i-2] + T[i], A[i-1] \}, \]

for any \( i \geq 2 \). The base case is

\[ A[0] = T[0] \quad A[1] = \max \{ T[0], T[1] \}. \]

The reason for this relationship is as follows. Suppose that we have an optimal solution for \( A[i] \). There are two cases: either there is a tomato planted at position \( i \), or there is not.
• In the case where there is a tomato planted at position $i$, we have


This is because if we just ignore the $i - 1, i$ positions of the garden, this gives us an optimal solution to $A[i - 2]$. (Otherwise, if there were a better solution, we could just take that solution and plant a tomato at position $i$, and this would give us a better solution for $A[i]$).

• In the case where there is no tomato planted at position $i$, we have


This follows from similar reasoning: our optimal solution to $A[i]$ is also an optimal solution to $A[i - 1]$; otherwise we’d have a better solution for $A[i]$.

Thus, since exactly one of the two cases occurs and $A[i]$ is the maximum deliciousness, we have

$$A[i] = \max\{A[i - 2] + T[i], A[i - 1]\}.$$

(b) The pseudocode is as follows:

```python
def plantTomatoes(T):
    Initialize an array $A = [T[0], \max\{T[0], T[1]\}, \text{None}, \text{None}, \ldots, \text{None}]$ of length $n$
    for $j = 2, \ldots, n-1$:
    return $A[n-1]$
```

That is, the algorithm walks through $A$ in order, applying the recursive relationship from part (a).
4. (6 pt.) [Fish fish eat eat fish.] Plucky the Pedantic Penguin enjoys fish, and he has discovered that on some days the fish supply is better in Lake A and some days the fish supply is better in Lake B. He has access to two tables $A$ and $B$, where $A[i]$ is the number of fish he can catch in Lake A on day $i$, and $B[i]$ is the number of fish he can catch in Lake B on day $i$, for $i = 0, \ldots, n - 1$.

If Plucky is at Lake A on day $i$ and wants to be at Lake B on day $i + 1$, he may pay $L$ fish to a polar bear who can take him from Lake A to Lake B overnight; the same is true if he wants to go from Lake B back to Lake A. The polar bear does not accept credit, so Plucky must pay before he travels. (And if he cannot pay, he cannot travel).

Assume that when day 0 begins, Plucky is at Lake A, and he has zero fish. Also assume that $A[i]$ and $B[i]$ are positive integers for $i = 0, 1, \ldots, n - 1$ and that $L$ is also a positive integer.

For example, suppose that $n = 3$, $L = 3$, and that $A$ and $B$ are given by


Then Plucky might do:

\begin{align*}
\text{Lake A} & \quad \text{Lake B} \\
\text{Day 0} & \quad \{ \\
0 \text{ fish} & \quad 2 \text{ fish} \\
+5 & \quad +7 \\
5 \text{ fish} & \quad 9 \text{ fish} \\
\} & \quad +4 \\
\downarrow & \quad \downarrow \\
-3 & \quad 13 \text{ fish} \\
\text{Day 1} & \quad \{ \\
\text{Day 2} & \quad \}
\end{align*}

So Plucky's total fish at the end of day $n - 1 = 2$ is 13.

In this question, you will design an $O(n)$-time dynamic programming algorithm that finds the maximum number of fish that Plucky can have at the end of day $n - 1$. Do this by answering the two parts below.

(a) (3 pt.) What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between the sub-problems?

[We are expecting:

- A clear description of your sub-problems.
- A recursive relationship that they satisfy, along with a base case.
- An informal justification that the recursive relationship is correct.
]

(b) (3 pt.) Design a dynamic programming algorithm that takes as input $A, B, L$ and $n$, and in time $O(n)$ returns the maximum number of fish that Plucky can have at the end of day $n - 1$.

[We are expecting: Pseudocode AND a short English description of what it does and why it works, and a justification of why it runs in time $O(n)$.]
SOLUTION:

(a) Our sub-problems will be indexed by a flag $\ell$ (either A or B), an integer $i \in \{0, \ldots, n-1\}$. Let $K[\ell, i]$ denote the number of fish that Plucky owns on day $i$, assuming he is in location $\ell$ on that day. The recursive relationship is:

$$K[\ell, i] = \begin{cases} \max\{K[A, i - 1] + A[i], K[B, i - 1] - L + A[i]\} & \text{if } K[B, i - 1] \geq L \\ K[A, i - 1] + A[i] & \text{else} \end{cases}$$

and

$$K[B, i] = \begin{cases} \max\{K[B, i - 1] + B[i], K[A, i - 1] - L + B[i]\} & \text{if } K[A, i - 1] \geq L \\ K[B, i - 1] + B[i] & \text{else} \end{cases}$$

The base case is that $K[A, 0] = A[0]$ and $K[B, 0] = -\infty$. To see that this is correct, consider just the first one. The maximum amount of fish that Plucky can eat if he is in location A on day $i$ is the maximum of two possibilities: either he was in location A on day $i - 1$, stayed in location A, and gained $A[i]$ fish; or else he was in location B on day $i - 1$, paid $L$ to the polar bear to move from B to A, and then ate $A[i]$ fish. Plucky only has the possibility of traveling if his current amount of fish, $K[B, i - 1]$, exceeds $L$. This explains the expressions in the first equation. The second equation is similar.

To see why the base case is correct, observe that $K[A, 0]$ should be the number of fish that Plucky has at the end of day 0, assuming he was in location A; this is just $A[0]$. $K[B, 0]$ should be the number of fish that Plucky has at the end of day 0 assuming he was in location B; Plucky is not allowed to be in location B on day 0, so we’ll set $K[B, 0]$ to negative infinity and we observe that this works with our recursive relationship because the maximum will never choose $-\infty$.

(b) `fishFishEatEatFish(A, B, L, n):
    initialize a 2-by-n array K
        // the first coordinate of K is indexed by {A,B}
        // the second coordinate of K is indexed by {0, ..., n-1}
    K[A,0] = A[0]
    K[B,0] = -Infinity
    for i = 1, ..., n-1:
        if K[B,i-1] >= L:
            K[A,i] = max{ K[A,i], K[B,i-1] - L + A[i] }
        K[B,i] = K[B,i-1] + B[i]
        if K[A,i -1] >= L:
            K[B,i] = max{ K[B,i], K[A,i-1] - L + B[i] }
    return max{ K[B,n-1], K[A,n-1] }
`

This works because it is implementing the recursive relationship from part (a). The base case is that on day 1, we have $K[A,0] = A[0]$ (Plucky can stay and fish) and $K[B,0] = -\infty$ (Plucky is not allowed to start Day 0 at Lake B).

The running time is $O(n)$, because the loop is over $i = 1, ..., n - 1$, and the amount of work inside each iteration is $O(1)$.
This problem set is long enough, but more practice with dynamic programming is always good. If you’d like another practice problem (or just miss Socrates the Scientific Squirrel), you can do the problem below for one bonus point. To get the bonus point, you should complete the whole problem.

5. (NOT REQUIRED, WORTH 1 BONUS pt.) [Nuts! (part 2)]

Socrates the Scientific Squirrel (from HW1) is back! Recall that Socrates lives in a very tall tree with \( n \) branches, and she wants to find out what is the lowest branch \( i \in \{1, \ldots, n\} \) so that an acorn will break open when dropped from branch \( i \). If an acorn breaks open when dropped from branch \( i \), then an acorn will also break open when dropped from branch \( j \) for any \( j \geq i \). (If no branch will break an acorn, Socrates should return \( n + 1 \)).

The catch is that, once an acorn is broken open, Socrates will eat it immediately and it can’t be dropped again.

In HW1, you designed a strategy for Socrates to use very few drops, given that she had \( k \) acorns. She was pretty pleased with that algorithm, but now she wants to compute exactly the number of drops she needs, in the worst case.

For \( n \geq 0 \) and \( k \geq 1 \), let \( D[n, k] \) be the optimal worst-case number of drops that Socrates needs to determine the correct branch out of \( n \) branches using \( k \) acorns. That is, \( D[n, k] \) is the number of drops that the best algorithm would use in the worst-case.

(a) For any \( 1 \leq j \leq k \), what is \( D[0, j] \)? What is \( D[1, j] \)? For any \( 1 \leq m \leq n \), what is \( D[m, 1] \)? (Note that if \( n = 0 \), then Socrates needs zero drops to identify the correct branch, since there are no branches).

[We are expecting: Your answer. No justification required.]

(b) Suppose the best algorithm drops the first acorn from branch \( x \in \{1, \ldots, n\} \). Write a formula for the optimal worst-case number of drops remaining in terms of \( D[x - 1, k - 1] \) and \( D[n - x, k] \).

[We are expecting: Your formula and an informal explanation of why this formula is correct.]

(c) Write a formula for \( D[n, k] \) in terms of values \( D[m, j] \) for \( j \leq k \) and \( m < n \).

[HINT: Use part (b).]

[We are expecting: Your formula and an informal explanation of why this formula is correct.]

(d) Design a dynamic programming algorithm which will compute \( D[n, k] \) in time \( O(n^2k) \).

[HINT: Use parts (a) and (c).]

[We are expecting: Pseudocode AND an English description of how it works, as well as an informal justification of the running time. You do not need to justify that it is correct.]

SOLUTION:

(a) \( D[0, i] = 0 \) because if there are zero branches then Socrates needs zero drops. \( D[1, i] = 1 \) because if there is only one branch then Socrates should drop an acorn from that branch to see if it breaks or not. \( D[m, 1] = m \) because with one acorn Socrates needs \( m \) drops in the worst case, since she must start from the bottom of the tree and work up one branch at a time.
(b) The formula is
\[ D[n, k] = 1 + \max \{ D[x - 1, k - 1], D[n - x, k] \}. \]

This is because if the acorn breaks from branch \( x \), then Socrates has \( k - 1 \) acorns left, and she knows that there are only \( x - 1 \) possible branches, so this is the same as trying to solve the original problem with \( k \leftarrow k - 1 \) and \( n \leftarrow x - 1 \), so the number of drops is \( D[k - 1, x - 1] \). On the other hand, if the acorn does not break, then Socrates still has \( k \) acorns left, and she knows that there are only \( n - x \) possible branches (the ones above branch \( x \)), so this is the same as trying to solve the original problem with \( k \leftarrow k \) and \( n \leftarrow n - x \). Thus the number of drops in this case is \( D[k, n - x] \). Since we are considering worst-case outcomes, the number of drops Socrates needs is the maximum of the two.

(c) The formula is
\[ D[n, k] = 1 + \min \{ \max \{ D[x - 1, k - 1], D[n - x, k] \}, x \in \{1, \ldots, n\} \}. \]

The reason is because the best algorithm must drop its first acorn from some branch \( x \). By part (b), if the best algorithm drops its first acorn from \( x \), then the number of drops remaining is \( \max \{ D[x - 1, k - 1], D[n - x, k] \} \); so then after we take the min we must have the number of drops of the best algorithm.

(d) The idea of our algorithm is to start by filling in the first two rows of the table \( D \), corresponding to \( n = 0 \) and \( n = 1 \). Then we will use the recurrence relation from part (c) to fill in the table, filling in the rows from bottom to top, and going across each row from left to right. The pseudocode is below. (You can assume that it is correct, although it’s a good exercise to prove it!)

---

**Algorithm 1: Find \( D[n, k] \)**

**Input:** \( n, k \)

- Initialize an \((n + 1) \times (k + 1)\) array \( D \) (zero-indexed).
- Set \( D[0, i] = 0 \) for all \( i = 0, 1, \ldots, k \)
- Set \( D[1, i] = 1 \) for all \( i = 0, 1, \ldots, k \)
- Set \( D[i, 0] = \infty \) for all \( i = 1, \ldots, n \)
- Set \( D[i, 1] = i \) for \( i = 0, \ldots, n \)

for \( m = 2, \ldots, n \) do

for \( j = 2, \ldots, k \) do

- \( D[m, j] = \infty \)

for \( x = 1, \ldots, m \) do

- \( D[m, j] = \min \{ D[m, j], \max \{ D[x - 1, j - 1], D[m - x, j] \} \} \)

/* Notice that \( D[m - x, j] \) and \( D[x - 1, j - 1] \) have already been filled out. */

- \( D[m, j] = D[m, j] + 1 \)

end do

end do

return \( D[n, k] \)

---

The running time is \( O(n^2k) \) because there are three for-loops, one over \( n \), one over \( k \) and one over \( m \leq n \), and the work done inside the three for-loops is \( O(1) \) (just looking up items in a table and taking mins/maxes).