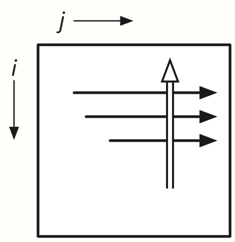
1. (a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome.

Solution (recurrence+cartoon): Let $LPS(i,j)$ denote the length of the longest palindrome subsequence of $A[i \mathinner{..} j]$. This function obeys the following recurrence:

$$LPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max\left\{ \begin{array}{c} LPS(i+1,j) \\ LPS(i,j-1) \end{array} \right\} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ \max\left\{ \begin{array}{c} 2 + LPS(i+1,j-1) \\ LPS(i+1,j) \\ LPS(i,j-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We need to compute $LPS(1,n)$.



The resulting dynamic programming algorithm runs in ***O*(*n*²) *time***. ■

Solution (pseudocode): In the following algorithm, $LPS[i,j]$ is the length of the longest palindrome subsequence of $A[i \mathinner{..} j]$.

```
LPS(A[1..n]):
  for i ← n down to 1
      LPS[i, i−1] ← 0
      LPS[i, i] ← 1
      for j ← i+1 to n
          LPS[i, j] ← max{LPS[i+1, j], LPS[i, j−1]}
          if A[i] = A[j]
              LPS[i, j] ← max{LPS[i, j], 2 + LPS[i+1, j−1]}
  return LPS[1, n]
```

The algorithm runs in ***O*(*n*²) *time***. ■

Solution (with greedy optimization): Let $LPS(i,j)$ denote the length of the longest palindrome subsequence of $A[i \mathinner{..} j]$. Before stating a recurrence for this function, we make the following useful observation.[a]

Claim 1. *If $i < j$ and $A[i] = A[j]$, then* $\text{LPS}(i,j) = 2 + \text{LPS}(i+1,j-1)$.

Proof: Suppose $i < j$ and $A[i] = A[j]$. Fix an arbitrary longest palindrome subsequence S of $A[i \mathinner{..} j]$. There are four cases to consider.

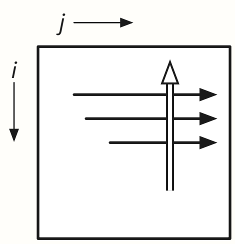
- If S uses neither $A[i]$ nor $A[j]$, then $A[i] \bullet S \bullet A[j]$ is a palindrome subsequence of $A[i\,..\,j]$ that is longer than S, which is impossible.
- Suppose S uses $A[i]$ but not $A[j]$. Let $A[k]$ be the last element of S. If $k = i$, then $A[i] \bullet A[j]$ is a palindrome subsequence of $A[i\,..\,j]$ that is longer than S, which is impossible. Otherwise, replacing $A[k]$ with $A[j]$ gives us a palindrome subsequence of $A[i\,..\,j]$ with the same length as S that uses both $A[i]$ and $A[j]$.
- Suppose S uses $A[j]$ but not $A[i]$. Let $A[h]$ be the first element of S. If $h = j$, then $A[i] \bullet A[j]$ is a palindrome subsequence of $A[i\,..\,j]$ that is longer than S, which is impossible. Otherwise, replacing $A[h]$ with $A[i]$ gives us a palindrome subsequence of $A[i\,..\,j]$ with the same length as S that uses both $A[i]$ and $A[j]$.
- Finally, S might include both $A[i]$ and $A[j]$.

In all cases, we find either a contradiction or a longest palindrome subsequence of $A[i\,..\,j]$ that uses both $A[i]$ and $A[j]$. □

Claim 1 implies that the function LPS satisfies the following recurrence:

$$LPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max\left\{LPS(i+1,j),\ LPS(i,j-1)\right\} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ 2 + LPS(i+1,j-1) & \text{otherwise} \end{cases}$$

We need to compute $LPS(1,n)$.



The resulting dynamic programming algorithm runs in ***$O(n^2)$ time***.

(Well, that was a whole lot of work for nothing.) ■

[a]And yes, optimizations like this ***always*** require a proof of correctness, both in homework and on exams. Premature optimization is the root of all evil.

Rubric: 5 points: standard dynamic programming rubric (scaled). This is neither the only correct recurrence nor the only correct evaluation order for this recurrence.

(b) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a repeater.

Solution (recurrence+iterative details): We actually solve a more general problem. A ***longest common subsequence*** of two strings A and B is a string of maximum length that is both a subsequence of A and a subsequence of B. For example, ONA is a longest common subsequence of IRONMAN and TONYSTARK. Every repeater subsequence of $A[1..n]$ is a string of the form ww, where w is a common subsequence of some prefix $A[1..j-1]$ and the corresponding suffix $A[j..n]$.

For any indices $i < j \leq k$, let $LCS(i, j, k)$ denote the length of the longest common subsequence of the prefix $A[1..i]$ and the substring $A[j..k]$. The length of the longest repeater subsequence of A is exactly $2 \cdot \max_{1 \leq j \leq n} LCS(j-1, j, n)$. The LCS function obeys the following recurrence:

$$LCS(i,j,k) = \begin{cases} 0 & \text{if } k < j \\ 0 & \text{if } i < 1 \\ \max\left\{ \begin{array}{c} LCS(i,j,k-1) \\ LCS(i-1,j,k) \end{array} \right\} & \text{if } A[i] \neq A[k] \\ \max\left\{ \begin{array}{c} LCS(i,j,k-1) \\ LCS(i-1,j,k) \\ 1 + LCS(i-1,j,k-1) \end{array} \right\} & \text{if } A[i] = A[k] \end{cases}$$

We can memoize this function into a three-dimensional array $LCS[0..n, 0..n, 0..m]$, which we can fill using three nested for-loops, considering j in arbitrary order in the outer loop, increasing i in the middle loop, and increasing k in the inner loop.

The resulting algorithm runs in ***$O(n^3)$ time***. ■

Solution (pseudocode): In the following algorithm, $LCS[i, j, k]$ is the length of the longest common subsequence of the prefix $A[1..i]$ and the substring $A[j..k]$.

```
MaxRepeater(A[1..n]):
  maxLCS ← 0
  for j ← 1 to n
      for k ← j − 1 to n
          LCS[0, j, k] ← 0
      for i ← 1 to j − 1
          LCS[i, j, j − 1] ← 0
          for k ← j to n
              LCS[i, j, k] ← max{LCS[i, j, k − 1], LCS[i − 1, j, k]}
              if A[i] = A[k]
                  LCS[i, j, k] ← max{LCS[i, j, k], 1 + LCS[i − 1, j, k − 1]}
      maxLCS ← max{maxLCS, 2 · LCS[j − 1, j, n]}
  return maxLCS
```

The algorithm runs in ***$O(n^3)$ time***. ■

Rubric: 5 points: standard dynamic programming rubric (scaled). This is neither the only correct recurrence nor the only correct evaluation order for this recurrence. We can simplify this algorithm *very* slightly by using a two-dimensional memoization array, indexed only by i and k.

We can also simplify the last case of the recurrence to $LCS(i, j, k) = 1 + LCS(i-1, j, k-1)$ if $A[i] = A[k]$, but this greedy optimization requires a proof.

2. Describe and analyze an efficient algorithm to solve the following one-dimensional clustering problem. Given an unsorted array $Data[1..n]$ of real numbers, we want to decompose this data into k clusters, each represented by an interval of indices and a real value, so that the maximum error between any data point and its cluster value is minimized. *See the homework handout for a detailed problem description.*

Solution: Let $MinErr(j,m)$ denote the error of the optimal covering of the prefix $Data[1..j]$ by m intervals. We need to compute $MinError(n,k)$. We also define two helper functions:

- $Min(i,j)$ is the minimum element of $Data[i..j]$
- $Max(i,j)$ is the maximum element of $Data[i..j]$

The best value for an output interval covering $Data[i..j]$ is $\frac{1}{2}(Max(i,j)+Min(i,j))$; this value has maximum error $\frac{1}{2}(Max(i,j)-Min(i,j))$.

Our three functions satisfy the following recurrences:

$$Min(i,j) = \begin{cases} Data[i] & \text{if } i=j \\ \min\left\{Data[j],\ Min(i,j-1)\right\} & \text{otherwise} \end{cases}$$

$$Max(i,j) = \begin{cases} Data[i] & \text{if } i=j \\ \max\left\{Data[j],\ Max(i,j-1)\right\} & \text{otherwise} \end{cases}$$

$$MinErr(j,m) = \begin{cases} 0 & \text{if } j=0 \text{ and } m=0 \\ \infty & \text{if } j=0 \text{ xor } m=0 \\ \min\limits_{1\le i\le j} \max\left\{\begin{array}{l} \frac{1}{2}(Max(i,j)-Min(i,j)) \\ MinErr(i-1,m-1) \end{array}\right\} & \text{otherwise} \end{cases}$$

The last case of the $MinErr$ recurrence looks for the first index i covered by the mth interval in the optimal clustering.

Our dynamic programming algorithm has two phases:

- First we memoize the Min and Max functions into $n\times n$ arrays, which we can fill in standard row-major order using two nested for-loops, increasing i in the outer loop and increasing j in the inner loop, in $O(n^2)$ time.
- Then we memoize the $MinErr$ function into an $n\times k$ array, which we fill in row-major order using *three* nested for-loops, increasing j in the outer loop, increasing m in the middle loop, and increasing i in the inner loop.

The overall algorithm runs in $\boldsymbol{O(n^2k)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. It is not necessary to describe how to compute the actual breakpoints and values.

3. You've been hired to store a sequence of n books on shelves in a library using as little *vertical* space as possible. Each shelf must store a contiguous interval of the given sequence of books. Each shelf has length equal to L, and each book has width at most L. You can adjust the height of each shelf to match the tallest book on that shelf.

   (a) Show that the natural greedy algorithm (pack as many books as possible on the first shelf and recurse) does *not* yield an optimal solution if the books can have different heights.

   Solution: Suppose $H = [1,2,2]$ and $W = [1,1,1]$ and $L = 2$. The greedy algorithm puts the first two books on one shelf and the third book on another shelf; this placement requires total height 4. But putting book 1 on one shelf and books 2 and 3 on another shelf uses total height 3. ■

   Rubric: 2 points = 1 for valid counterexample + 1 for argument that greedy is not optimal. This is not the only correct solution.

   (b) Describe and analyze an efficient algorithm to assign books to shelves to minimize the total height of the shelves.

   Solution (dynamic programming): For any index i, let $\mathit{MinTotalH}(i)$ denote the minimum total height required to shelve books i through n. We also define two helper functions for all indices $i \le j$:

   - $\mathit{MaxH}(i,j)$ denotes the maximum height of books i through j.
   - $\mathit{TotalW}(i,j)$ denotes the total width of books i through j.

   These functions satisfy the following recurrences:

$$\mathit{MaxH}(i,j) = \begin{cases} 0 & \text{if } i > j \\ \max\left\{\mathit{MaxH}(i,j-1),\ H[j]\right\} & \text{otherwise} \end{cases}$$

$$\mathit{TotalW}(i,j) = \begin{cases} 0 & \text{if } i > j \\ \mathit{TotalW}(i,j-1) + W[j] & \text{otherwise} \end{cases}$$

$$\mathit{MinTotalH}(i) = \begin{cases} 0 & \text{if } i > n \\ \min\left\{ \begin{array}{c} \mathit{MaxH}(i,j) + \\ \mathit{MinTotalH}(j+1) \end{array} \;\middle|\; \begin{array}{c} i \le j \le n \text{ and} \\ \mathit{TotalW}(i,j) \le L \end{array} \right\} & \text{otherwise} \end{cases}$$

   The following dynamic programming algorithm evaluates these recurrences and computes $\mathit{MinTotalH}(1)$ in $\boldsymbol{O(n^2)}$ ***time***:

```
SHELVE(H[1..n], W[1..n], L):
  ⟨⟨Precompute all MaxH and TotalW values⟩⟩
  for i ← n down to 1
      MaxH[i, i−1] ← 0
      TotalW[i, i−1] ← 0
      for j ← 1 to i
          MaxH[i, j] ← max{MaxH[i, j−i], H[j]}
          TotalW[i, j] ← TotalW[i, j−1] + W[j]

  ⟨⟨Main algorithm⟩⟩
  MinTotalH[n+1] ← 0
  for i ← n down to 1
      MinTotalH[i] ← ∞
      for j ← i to n
          if TotalW[i, j] ≤ L
              MinTotalH[i] ← min { MinTotalH[i], MaxH[i, j] + MinTotalH[j+1] }
  return MinTotalH[1]
```

With a bit more care, we can eliminate both two-dimensional arrays by computing the necessary values of $MaxH(i, j)$ and $TotalW(i, j)$ on the fly in the main body of the algorithm:

```
SHELVE(H[1..n], W[1..n], L):
  MinTotalH[n+1] ← 0
  for i ← n down to 1
      MinTotalH[i] ← ∞
      totalW ← 0
      maxH ← 0
      for j ← i to n
          totalW ← totalW + W[j]
          maxH ← max{maxH, H[j]}
          if totalW ≤ L
              MinTotalH[i] ← min {MinTotalH[i], maxH + MinTotalH[j+1]}
  return MinTotalH[1]
```

If every book width $W[i]$ and the shelf length L are positive integers, we can further improve the running time to $O(\min\{n^2, nL\})$ by breaking out of the inner loop as soon as $totalW > L$. On the other hand, the algorithms described above are correct even if book widths are real numbers, and arguably even if some books have negative width! ■

Solution (shortest path in a dag): We construct an edge-weighted directed acyclic graph $G = (V, E)$ as follows.

- There are $n + 1$ vertices, identified by the integers 0 through n. Vertex 0 is an artificial source; for every positive integer i, vertex i corresponds to the ith book.
- G contains the edge $i \to j$ for every pair of indices i and j such that books $i + 1$ through j fit onto a shelf. G contains at most $O(n^2)$ edges. Each edge goes from a lower-numbered vertex to a higher-numbered vertex, so G is acyclic, as claimed.

- The weight of edge $i \to j$ is equal to the maximum height of books $i+1$ through j.

Every path in G from vertex 0 to vertex n corresponds to a legal shelving of the books; specifically, each edge $i \to j$ in the path indicates that some shelf holds books $i+1$ through j. The length of the path is equal to the total height of the shelves. In particular, the *optimal* shelf assignment corresponds to the *shortest* path in G from 0 to n.

We can construct G in $O(n^2)$ time using the following algorithm:

```
BuildLibraryGraph(H[1..n], W[1..n], L):
  V ← {0, 1, 2, ..., n}
  E ← ∅
  for i ← 0 to n − 1
      TotalW ← 0
      MaxH ← 0
      for j ← i + 1 to n:
          TotalW ← TotalW + W[j]
          if TotalW > L
              break out of the inner for loop
          MaxH ← max{MaxH, H[j]}
          add edge i→j to E
          w(i→j) ← MaxH
  return V, E, w
```

To compute the optimal assignment of books to shelves, it remains only to find the shortest path in G from vertex 0 to vertex n. We can find this path in $O(V+E) = O(n^2)$ time by depth-first search, as described in the textbook. The total running time of our algorithm is $\boldsymbol{O(n^2)}$. ■

Rubric: 8 points.

- Dynamic programming: standard dynamic programming rubric (scaled). The final optimizations (in gray) are not necessary for full credit.
- Shortest path in DAG: 2 for correct graph definition + 2 for graph construction + 2 for correct problem (shortest path from 0 to $n+1$) + 2 for correct algorithm (DFS) + 2 for running time as a function of n.

These are not the fastest algorithms for this problem; these are not the only correct $O(n^2)$-time algorithms.

Max 6 points for $O(n^3)$-time algorithm (for example, recomputing *MaxH*(i,j) and/or *TotalW*(i,j) from scratch in each iteration of the inner loop); scale partial credit.

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  − 1 for naming the function "OPT" or "DP" or any single letter.
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns, not (here) an explanation of *how* that value is computed.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  − 2 for greedy optimizations without proof, even if they are correct.
  - **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 3 points for iterative details
  + 1 for describing (or sketching) an appropriate memoization data structure
  + 1 for describing (or sketching) a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order.
  + 1 for correct time analysis. (It is not necessary to state a space bound.)
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- ***Iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. However, you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10.***
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details.
  - If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points.
  - If the described function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points (= 70% of 3, rounded).
  - If the described function cannot lead to a polynomial-time algorithm, it could be worth 1 or even 0 points.