1. Describe and analyze an efficient algorithm to compute the longest perfect ruler subsequence of a given array $A[1..n]$ of integers.

Solution: Let $A[1..n]$ be the input array. Define the *rank* of a perfect ruler sequence to be r if its length is $2^r - 1$. Every perfect ruler sequence of rank r is composed of a perfect ruler sequence of rank $r-1$, followed by the maximum element of the sequence, followed by a perfect ruler sequence of rank $r-1$.

For any indices i, k, and p, let $LPRS(i,k,p)$ denote the ***rank*** of the longest perfect ruler subsequence of $A[i..k]$ in which every entry is smaller than $A[p]$. (The letter p is mnemonic for "previous" or "parent".) The empty sequence is a perfect ruler sequence with rank 0, so we have $LPRS(i,k,p) \geq 0$ for all i, k, and p. The $LPRS$ function obeys the following recurrence (with the implicit base case $\max \varnothing = 0$):

$$LPRS(i,k,p) = \begin{cases} 0 & \text{if } i > j \\ \max\left\{ 1 + \min\left\{ \begin{array}{l} LPRS(i,j-1,j) \\ LPRS(j+1,k,j) \end{array} \right\} \;\middle|\; \begin{array}{c} i \leq j \leq k \text{ and} \\ A[j] < A[p] \end{array} \right\} & \text{otherwise} \end{cases}$$

The recurrence considers all possible indices j for the maximum of the perfect ruler subsequence of $A[i..k]$. For each index j, we recursively compute the largest perfect ruler subsequences on either side of $A[j]$ with all values less than $A[j]$, trim the larger of those two subsequences to match the rank of the smaller, and then assemble the results into a perfect ruler sequence.

If we add an artificial sentinel value $A[0] = \infty$, the longest perfect ruler subsequence of $A[1..n]$ has rank $LPRS(1,n,0)$, and therefore has length $2^{LPRS(1,n,0)} - 1$.

We can memoize $LPRS$ into a three-dimensional array $LPRS[1..n, 1..n, 0..n]$. Each entry $LPRS[i,k,p]$ depends on entries $LPRS[i',k',p']$ where either $i < i'$ and $k' = k$, or $i = i'$ and $k > k'$. Thus, we can fill the array using three nested loops, *decreasing* i in the outer loop, *increasing* k in the middle loop, and considering p in arbitrary order in the inner loop.

Each entry $LPRS[i,k,p]$ depends on $2(k-i+1) = O(n)$ other entries, so the overall algorithm runs in $\boldsymbol{O(n^4)}$ ***time***. ■

Rubric: 10 points: standard DP rubric. Max 8 points for $O(n^4 \log n)$ time (for example, defining a boolean recursive function with height as a fourth parameter); max 6 points for slower polynomial time algorithm; scale partial credit. This is not the only correct evaluation order for this recurrence. This is not the fastest possible algorithm for this problem; see below.

Solution (+3 extra credit): We can improve the previous solution by observing that in every call to $LPRS(i,k,p)$, we have either $p = j+1$ or $p = i-1$. So we only have to evaluate those $O(n^2)$ entries in the memoization table. Since evaluating each entry takes $O(n)$ time, the improved algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

Equivalently, we can define two functions, for any indices i and k.

- $LPRS^+(i,k)$ denotes the ***rank*** of the longest perfect ruler subsequence of $A[i\,..\,k]$ in which every entry is smaller than $A[k+1]$.
- $LPRS^-(i,k)$ denotes the ***rank*** of the longest perfect ruler subsequence of $A[i\,..\,k]$ in which every entry is smaller than $A[i-1]$.

After adding the sentinel value $A[0] = \infty$, we need to compute $LPRS^-(1,n)$ (or if we really want length instead of rank, $2^{LPRS^-(1,n)} - 1$). These two functions obey the following mutual recurrences:

$$LPRS^+(i,k) = \begin{cases} 0 & \text{if } i > j \\ \max\left\{ 1 + \min\left\{ \begin{array}{l} LPRS^+(i,j-1) \\ LPRS^-(j+1,k) \end{array} \right\} \;\middle|\; \begin{array}{l} i \le j \le k \text{ and} \\ A[j] < A[\boldsymbol{k+1}] \end{array} \right\} & \text{otherwise} \end{cases}$$

$$LPRS^-(i,k) = \begin{cases} 0 & \text{if } i > j \\ \max\left\{ 1 + \min\left\{ \begin{array}{l} LPRS^+(i,j-1) \\ LPRS^-(j+1,k) \end{array} \right\} \;\middle|\; \begin{array}{l} i \le j \le k \text{ and} \\ A[j] < A[\boldsymbol{i-1}] \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize these two functions into two 2d arrays $LPRS^+[1\,..\,n, 1\,..\,n]$ and $LPRS^-[1\,..\,n, 1\,..\,n]$. Each entry $LPRS^\pm[i,k]$ depends only on entries $LPRS^\pm[i',k']$ where either $i < i'$ and $k' = k$, or $i = i'$ and $k > k'$. Thus, we can fill the arrays using two nested loops, *decreasing* i in the outer loop, *increasing* k in the inner loop, and evaluating both $LPRS^+[i,k]$ and $LPRS^-[i,k]$ in each iteration of the inner loop.

Each entry $LPRS^\pm[i,k]$ depends on $2(k-i+1) = O(n)$ other entries, so the overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Rubric: This is not the only correct evaluation order for this recurrence. This is not the fastest algorithm for this problem; see below.

Solution (+10 extra credit): Let $A[1..n]$ be the input array. Like previous algorithms, the following algorithm finds the maximum *rank* of a perfect ruler subsequence of $A[1..n]$. If this maximum rank is r, then the maximum *length* of a perfect ruler subsequence is $2^r - 1$.

We implicitly consider a perfect ruler subsequence to be a perfectly balanced binary tree, with the maximum element at the root.

For any index j and any rank r, we define two functions that encode the *narrowest* perfect ruler subsequence of the entire array $A[1..n]$ whose root index is j (meaning its largest element is $A[j]$) and whose rank is r.

- $MaxL(j, r)$ is the largest first (leftmost) index of a perfect ruler subsequence of $A[1..n]$ with rank r and root index j, or $-\infty$ if there is no such subsequence.
- $MinR(j, r)$ is the smallest last (rightmost) index of a perfect ruler subsequence of $A[1..n]$ with rank r and root index j, or ∞ if there is no such subsequence.

We need to compute the largest value of r, such that $MinR(j, r) - MaxL(j, r)$ is finite for some index j. Because any subsequence of rank r has length $2^r - 1 \le n$, the largest rank r we ever need to consider is $\ell = \lfloor \log_2(n+1) \rfloor$.

These two functions obey the following mutual recurrence:

$$MaxL(j,r) = \begin{cases} j & \text{if } r = 1 \\ \max\left\{ MaxL(i, r-1) \;\middle|\; \begin{array}{c} 1 \le i < j \text{ and} \\ A[i] < A[j] \text{ and} \\ MinR(i, r-1) < j \end{array} \right\} & \text{otherwise} \end{cases}$$

$$MinR(j,r) = \begin{cases} j & \text{if } r = 1 \\ \min\left\{ MinR(k, r-1) \;\middle|\; \begin{array}{c} j < k \le n \text{ and} \\ A[k] < A[j] \text{ and} \\ MaxL(k, r-1) > j \end{array} \right\} & \text{otherwise} \end{cases}$$

The recurrence for $MaxL(j, r)$ considers all possible indices $i < j$ that could be the left child of the root j, and the recurrence for $MinR(j, r)$ considers all possible right children $k > j$ of the root j.

We can memoize these two functions into two two-dimensional arrays $MaxL[1..n, 1..\ell]$ and $MinR[1..n, 1..\ell]$. We can fill both of these arrays by decreasing r in the outer loop, considering j in any order in the inner loop, and evaluating both $MaxL[j, r]$ and $MinR[j, r]$ in each inner iteration.

Evaluating a single entry $MaxL[j, r]$ or $MinR[j, r]$ requires $O(n)$ time. Thus, the entire algorithm runs in $O(\ell \cdot n \cdot n) = \boldsymbol{O(n^2 \log n)}$ ***time***. ■

Rubric: +10 extra credit; partial credit at Jeff's discretion. I'll be *very* surprised if anyone seriously attempts this approach. This is not the fastest algorithm for this problem; see below!

Solution (What would Timothy do?): Let's look more closely at the recurrences for *MaxL* and *MinR*.

First, notice that the comparison $i < j$ in the *MaxL* recurrence is redundant, because $i \le MinR(i, r-1)$, and the recurrence already requires $MinR(i, r-1) < j$. Similarly, the comparison $j < k$ in the *MinR* recurrence is redundant.

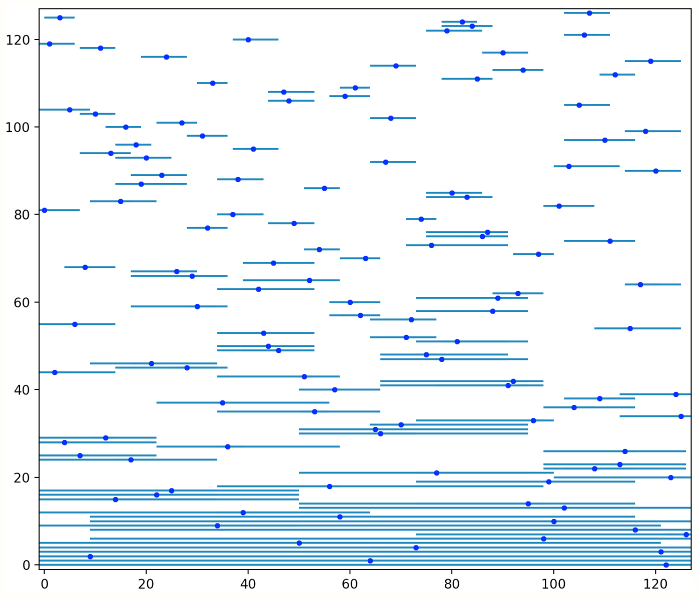
$$MaxL(j,r) = \begin{cases} j & \text{if } r = 1 \\ \max\left\{ MaxL(i, r-1) \;\middle|\; \begin{array}{c} A[i] < A[j] \text{ and} \\ MinR(i, r-1) < j \end{array} \right\} & \text{otherwise} \end{cases}$$

$$MinR(j,r) = \begin{cases} j & \text{if } r = 1 \\ \min\left\{ MinR(k, r-1) \;\middle|\; \begin{array}{c} A[k] < A[j] \text{ and} \\ MaxL(k, r-1) > j \end{array} \right\} & \text{otherwise} \end{cases}$$

Now fix a rank r. For any index i, let s_i denote the horizontal line segment with left endpoint $(MaxL[i, r-1], i)$ and right endpoint $(MinR[i, r-1], i)$, and let p_i denote the point $(A[i], i)$. We can rephrase the recurrences for *MaxL* and *MinR* as follows:

- $MaxL[j, r]$ is the rightmost left endpoint among all segments s_i that lie entirely in the quadrant below and left of p_j.
- $MinR[j, r]$ is the leftmost right endpoint among all segments s_i that lie entirely in the quadrant below and right of p_j.

Here is an example of the segments s_i and points p_j for one iteration ($r = 3$) of our algorithm, for an input array of size 127. (Thank you, matplotlib.) Each segment s_i passes through the corresponding point p_i because $MaxL[i, r-1] \le i \le MinR[i, r-1]$ (with equality if and only if $r = 2$).



We compute $MaxL[j, r]$ for all j using a *sweep line*. Imagine sweeping a vertical line from left to right across the segments s_i and points p_j. We maintain a data structure that stores all the *left* endpoints of segments to the left of the sweep line, and that answers queries of the form "What is the rightmost point below this horizontal line?" When the sweep line passes the *right* endpoint of any segment s_i, we insert the corresponding *left* endpoint into our data structure. When the sweep line reaches any point p_j, we perform a query with the horizontal line $y = j$. To process these insertions and queries in the correct order, we sort all n right endpoints and all n query points by their x-coordinates (breaking ties by ordering query points before right endpoints) in $O(n \log n)$ time, and then run a simple for loop over these $2n$ sorted points.

Our sweep-line data structure stores $(key, value)$ pairs — where the keys are the y-coordinates of segments, and the values are the x-coordinates of left endpoints — and supports queries of the following form: Given a number y, what is the maximum *value* among all $(key, value)$ pairs such that $key < y$?

We can build a suitable data structure by modifying a CS-225-standard balanced binary search tree (AVL tree, red-black tree, splay tree, or whatever) over the *keys*; at each node v in the tree, we store the maximum *value* among all nodes in the subtree rooted at v.

- Answering a query takes $O(\log n)$ time. We preform a standard binary search for the predecessor of the query value y. The answer to the query is the largest of the maximum values stored at left children of nodes on the search path.
- Inserting a new point also takes $O(\log n)$ time. We follow the standard algorithm for inserting a new key into a binary search tree, then update the maximum values along all nodes in the insertion path, and finally perform any necessary rotations to keep the tree balanced. Maintaining correct maximum values at the nodes during a rotation is straightforward.

Altogether, sorting the various x-coordinates, and then performing n insertions and n queries, takes $O(n \log n)$ time. Thus, for each rank r, we can compute $MaxL[j, r]$ for all j in $O(n \log n)$ time; a symmetric algorithm compute $MinR[j, r]$ for all j.

There are only $O(\log n)$ relevant values of r, so our overall algorithm runs in ***$O(n \log^2 n)$ time***.[a] ■

[a] Actually, Timothy would use a dynamic van Emde Boas tree, which can handle both insertions and queries in $O(\log \log n)$ time, for the sweep-line data structure, along with a $O(n)$-time counting/radix sort for the initial preprocessing. Both of these changes exploit the fact that all keys and values and so on are integers between 1 and n. This modification reduces the overall running time of the algorithm to **$O(n \log n \log \log n)$**. If you want to know what a "dynamic van Emde Boas tree" is, you'll have to take an advanced data structures course from one of us. Neither Timothy nor I have any reason to believe that this algorithm is as fast as possible.

Rubric: Have you thought about graduate school?

2. Describe and analyze an algorithm to load the ferry as lightly as possible. *[See the homework handout for a detailed problem statement]*

Solution: Let L be the given ferry length, and let $len[1..n]$ be the given length array.

For any integers a, b, c, and m, let $Ferry(a, b, c, m)$ denote the minimum number of cars that can be loaded onto the ferry if

- lane 1 has a meters available,
- lane 2 has b meters available,
- lane 3 has c meters available, and
- there are $n - m + 1$ cars in the queue with lengths $len[m..n]$.

We need to compute $Ferry(L, L, L, 1)$. This function satisfies the following recurrence:

$$Ferry(a,b,c,m) = \begin{cases} 0 & \text{if } m > n \\ 0 & \text{if } len[m] > \max\{a,b,c\} \\ \infty & \text{if } a < 0 \text{ or } b < 0 \text{ or } c < 0 \\ 1 + \min \begin{Bmatrix} Ferry(a - len[m],\ b,\ c,\ m+1) \\ Ferry(a,\ b - len[m],\ c,\ m+1) \\ Ferry(a,\ b,\ c - len[m],\ m+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function into an $L \times L \times L \times n$ array. We can fill this array in constant time per entry by *decreasing* the fourth index in the outermost loop, and considering the other three indices in any order in the inner loops. The resulting algorithm runs in at most ***$O(L^3 n)$ time***. ■

Rubric: 10 points: standard dynamic programming rubric. This is not the only correct evaluation order for this recurrence.

Solution (+2 extra credit): Because the length of each vehicle is a positive integer, at most $3L$ vehicles can fit on the ferry. So we can assume without loss of generality that $n \le 3L$. This assumption improves the running time of the previous algorithm to $\boldsymbol{O(L^4)}$. ■

Solution (+3 extra credit): We can speed up the previous algorithm by observing that in every recursive invocation of the *Ferry* function, the arguments satisfy the following identity:

$$3L - a - b - c = \sum_{i=1}^{m-1} len[i].$$

Both sides of this equation equal the total length of the first $m - 1$ vehicles. Thus, one of the parameters of our recurrence is redundant. For any integers a, b, and m, let $Ferry2(a, b, m)$ denote the minimum number of cars that can be loaded onto the ferry under the following conditions:

- lane 1 has a meters available,
- lane 2 has b meters available,
- lane 3 has $3L - a - b - \sum_{i=1}^{m-1} len[i]$ meters available, and
- there are $n - m + 1$ cars in the queue with lengths $len[m\,..\,n]$.

We need to compute $Ferry2(L, L, 1)$. As a helper function, for any integer $0 \le i \le n$, let $TotalLen(i)$ be the total length of the first i vehicles; this function obeys the following simple recurrence:

$$TotalLen(i) = \begin{cases} 0 & \text{if } i = 0 \\ TotalLen(i-1) + len[i] & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $TotalLen[0\,..\,n]$; we can easily fill this array in $O(n)$ time from left to right.

Now the *Ferry2* function obeys the following recurrence:

$$Ferry2(a, b, m) = \begin{cases} 0 & \text{if } m > n \\ 0 & \text{if } len[m] > \max\{a, b, c\} \\ \infty & \text{if } a < 0 \text{ or } b < 0 \text{ or } c < 0 \\ 1 + \min \begin{Bmatrix} Ferry2(a - len[m],\ b,\ m+1) \\ Ferry2(a,\ b - len[m],\ m+1) \\ Ferry2(a,\ b,\ m+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

$$\text{where } c = 3L - a - b - TotalLen(m-1)$$

We can memoize *Ferry2* into an $L \times L \times n$ array. Once the *TotalLen* array is filled, we can fill the *Ferry2* array in constant time per entry by *decreasing* the third index in the outermost loop, and considering the other two indices in any order in the inner loops. The resulting improved algorithm runs in ***$O(L^2 n)$ time***. ■

Solution (+5 extra credit): Because the length of each vehicle is a positive integer, at most $3L$ vehicles can fit on the ferry. So we can assume without loss of generality that $n \le 3L$. With this assumption in place, our improved algorithm actually runs in ***$O(L^3)$ time***. ■

Non-solution: Neither of the natural greedy strategies for filling the ferry lead to optimal assignments:

- **Put each vehicle into the lane with the *least* available space:** Consider the input $L = 6$ and $len = [1, 1, 1, 6, 6]$.
  - The greedy strategy fits all five vehicles onto the ferry, without loss of generality in lanes $1, 1, 1, 2, 3$.
  - The optimal assignment loads the first three cars into three different lanes, after which the fourth vehicle doesn't fit anywhere.
- **Put each vehicle into the lane with the *most* available space:** Consider the input $L = 12$ and $len = [1, 2, 3, 4, 5, 6, 7]$.
  - The greedy algorithm loads all seven vehicles onto the ferry, without loss of generality in lanes $1, 2, 3, 1, 2, 3, 1$.
  - The optimal assignment loads the first six cars into lanes $1, 2, 3, 3, 2, 1$, after which the seventh vehicle doesn't fit anywhere.

♣

3. Describe and analyze an algorithm to decide whether a given permutation of the integers 1 through n is an inorder traversal of a garbled binary search tree.

Solution (check root rank): Let $A[1..n]$ be the input permutation. For any indices i and k, define $\mathit{Garbled}(i,k) = \textsc{True}$ if $A[i..k]$ is a garbled inorder traversal, and define $\mathit{Garbled}(i,k) = \textsc{False}$ otherwise. We need to compute $\mathit{Garbled}(1,n)$.

As a helper function, let $\mathit{GoodSplit}(i,j,k) = \textsc{True}$ if and only if either of the following conditions are satisfied:

- $A[j] = \max A[i..j]$ and $A[j] = \min A[j..k]$
- $A[j] = \min A[i..j]$ and $A[j] = \max A[j..k]$

These are necessary conditions for $A[j]$ to be the root of a garbled binary search tree containing keys $A[i..k]$. The first condition is consistent with $A[j]$'s child pointers being correct; the second case is consistent with $A[j]$'s child pointers being swapped. Given any indices i, j, k, we can evaluate the function $\mathit{GoodSplit}(i,j,k)$ in $O(n)$ time by brute force.

The *Garbled* function satisfies the following recurrence:

$$\mathit{Garbled}(i,k) = \begin{cases} \textsc{True} & \text{if } i \geq k \\ \displaystyle\bigvee_{j=i}^{k} \begin{pmatrix} \mathit{GoodSplit}(i,j,k) \\ \wedge\ \mathit{Garbled}(i,j-1) \\ \wedge\ \mathit{Garbled}(j+1,k) \end{pmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\mathit{Garbled}[1..n, 1..n]$. Each entry $\mathit{Garbled}[i,k]$ depends only on entries $\mathit{Garbled}[i',k']$ with $i' > i$ and $k' = k$, or $i' = i$ and $k' < k$. Thus, we can fill the array by decreasing i in the outer loop, and increasing k in the inner loop.

Evaluating a single entry $\mathit{Garbled}[i,k]$ requires $O(n^2)$ time, which is dominated by $O(n)$ calls to the helper function *GoodSplit*. Thus, our overall algorithm runs in ***$O(n^4)$ time***. ■

Rubric: 10 points: standard dynamic programming rubric. Max 7 points for a slower polynomial-time algorithm; scale partial credit. This is not the only correct evaluation order for this recurrence. This is not the only correct algorithm that runs in $O(n^4)$ time. This is also not the fastest algorithm for this problem; see the next several pages!

Solution (+5 extra credit): We can speed up the previous algorithm by computing the *GoodSplit* function more efficiently.

For any fixed indices i and k, we can evaluate $GoodSplit(i, j, k)$ for ***all*** j in $O(n)$ time using the FindGoodSplits subroutine shown below. (This is actually two dynamic programming algorithms, one for normal children and one for swapped children.) For any indices i and k, after calling FindGoodSplits(i, k), we can evaluate $Garbled[i, k]$ in only $O(n)$ time. As a result, our overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

```
⟨⟨Compute an array $SplitHere[i..k]$ where $GoodSplit[j] = Splits(i, j, k)$ for all $j$⟩⟩
FindGoodSplits($i, k$):
  ⟨⟨Find all $j$ where either $A[j] = \max A[i..j]$ or $A[j] = \min A[i..j]$⟩⟩
  $max \gets -\infty$;  $min \gets +\infty$
  for $j \gets i$ to $k$
      if $A[j] > max$
          $max \gets A[j]$;  $Lmax[j] \gets$ True
      else
          $Lmax[j] \gets$ False
      if $A[j] < min$
          $min \gets A[j]$;  $Lmin[j] \gets$ True
      else
          $Lmin[j] \gets$ False
  ⟨⟨Find all $j$ where either $A[j] = \max A[j..k]$ or $A[j] = \min A[j..k]$⟩⟩
  $max \gets -\infty$;  $min \gets +\infty$
  for $j \gets k$ down to 1
      if $A[j] > max$
          $max \gets A[j]$;  $Rmax[j] \gets$ True
      else
          $Rmax[j] \gets$ False
      if $A[j] < min$
          $min \gets A[j]$:  $Rmin[j] \gets$ True
      else
          $Rmin[j] \gets$ False
  ⟨⟨Avengers, assemble!⟩⟩
  for $j \gets i$ to $k$
      $SplitHere[j] \gets (Lmax[j] \land Rmin[j]) \lor (Lmin[j] \land Rmax[j])$
  return $SplitHere[i..k]$
```

■

Solution (check root position, +5 extra credit): Let $A[1..n]$ be the input permutation. For any indices i and k define $Garbled(i, k) =$ True if and only if $A[i..k]$ is a garbled inorder traversal of a subtree $k - i + 1$ consecutive integers, and $Garbled(i, k) =$ False otherwise.

As a helper function, for any indices $i \le j \le k$, we define $GoodRoot(i, j, k) =$ True if and only if the value of $A[j]$ is consistent with j being the root of a garbled binary search tree for the $k - i + 1$ consecutive keys in $A[i..k]$. There are three conditions to consider:

- If $A[i\,..\,k]$ contains a permutation of $k-i+1$ consecutive integers, then $\max A[i\,..\,k] - \min A[i\,..\,k] = k-i$.
- If the children of the root have not been swapped, then $A[j]$ is both the $(j-i+1)$th smallest key and the $(k-j+1)$th largest key in the interval $A[i\,..\,k]$, and therefore

$$A[j] = \min A[i\,..\,k] + j - i = \max A[i\,..\,k] + j - k$$

- On the other hand, if the children of the root have been swapped, then $A[j]$ is both the $(j-i+1)$th *largest* key and the $(k-j+1)$th *smallest* key in the interval $A[i\,..\,k]$, and therefore

$$A[j] = \min A[i\,..\,k] - j + k = \max A[i\,..\,k] - j + i$$

For any indices i and k, we can compute $\min A[i\,..\,k]$ and $\max A[i\,..\,k]$ in $O(n)$ time by brute force, after which we can compute $\mathit{GoodRoot}(i,j,k)$ for any index j in $O(1)$ time.

The *Garbled* function satisfies the following recurrence:

$$\mathit{Garbled}(i,k) = \begin{cases} \textsc{True} & \text{if } i \geq k \\ \textsc{False} & \text{if } \max A[i\,..\,k] - \min A[i\,..\,k] > k - i \\ \displaystyle\bigvee_{j=i}^{k} \begin{pmatrix} \mathit{GoodRoot}(i,j,k) \\ \wedge\ \mathit{Garbled}(i,j-1) \\ \wedge\ \mathit{Garbled}(j+1,k) \end{pmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\mathit{Garbled}[1\,..\,n, 1\,..\,n]$. Each entry $\mathit{Garbled}[i,k]$ depends only on entries $\mathit{Garbled}[i',k']$ with $i' > i$ and $k' = k$, or $i' = i$ and $k' < k$, so we can fill the array by decreasing i in the outer loop, and increasing k in the inner loop.

We can evaluate any single entry $\mathit{Garbled}[i,k]$ in $O(n)$ time as follows. First we compute $\min A[i\,..\,k]$ and $\max A[i\,..\,k]$ by brute force. If $\max A[i\,..\,k] - \min A[i\,..\,k] \neq k-i$, we can immediately conclude that $\mathit{Garbled}[i,k] = \textsc{False}$. Otherwise, for each index j between i and k, we evaluate the expression $\mathit{GoodRoot}(i,j,k) \wedge \mathit{Garbled}[i,j-1] \wedge \mathit{Garbled}[j+1,k]$ in $O(1)$ time.

Altogether, our algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Rubric: Max 15 points for any correct algorithm that runs in $O(n^3)$ time; scale partial credit and award any points over 10 as extra credit. This is not the only correct algorithms with this running time. These are not the fastest algorithms for this problem; see below.

Solution (+10 extra credit): We can speed up the *first* $O(n^3)$-time solution (but *not* the second) using a greedy optimization.

Consider an interval $A[i\,..\,k]$. For any index j such that $i \le j \le k$, we call j a *forward split* if $A[j] = \max A[i\,..\,j]$ and $A[j] = \min A[j\,..\,k]$, and a *backward split* if $A[j] = \min A[i\,..\,j]$ and $A[j] = \max A[j\,..\,k]$. Every good split is either a forward split or a backward split.

In any interval $A[i\,..\,i]$ of length 1, index i is vacuously both a forward split and a backward split. The next lemma shows that aside from this trivial case, no interval contains both types of splits.

Lemma 1. *No interval $A[i\,..\,k]$ with $i+1 < k$ has both a forward split and a backward split.*

Proof: Fix indices i and k such that $i+1 < k$. For the sake of argument, suppose some index j is a forward split for $A[i\,..\,k]$ and some other index $j' \ne j$ is a backward split for $A[i\,..\,k]$.

The minimum element $\min A[i\,..\,k]$ must lie in the interval $A[1\,..\,j-1]$, because j is forward split, but it must also lie in the interval $A[j'+1\,..\,k]$, because j' is a backward split. Thus, $j'+1 \le j-1$.

Similarly, the maximum element $\max A[i\,..\,k]$ must like in both $A[j+1\,..\,k]$ and $A[1\,..\,j'-1]$, so $j+1 \le j'-1$.

We conclude that $j+1 \le j'-1 \le j-3$, which is impossible. □

Call an index j a *valid root* for $A[i\,..\,k]$ if there is a mangled binary search tree whose root is $A[j]$ and whose inorder traversal sequence is $A[i\,..\,k]$. Every valid root is either a forward split or a backward split.

Lemma 2. *If one forward split for $A[i\,..\,k]$ is a valid root for $A[i\,..\,k]$, then every forward split for $A[i\,..\,k]$ is a valid roots for $A[i\,..\,k]$.*

Proof: We prove the lemma by induction on the size of the array. For every proper subinterval $A[i'\,..\,k']$ of $A[i\,..\,k]$, assume that if one forward split for $A[i'\,..\,k']$ is a valid root for $A[i'\,..\,k']$, then every forward split for $A[i'\,..\,k']$ is a valid root for $A[i'\,..\,k']$.

Suppose some indices j and j' are both forward splits for $A[i\,..\,k]$, and that j is a valid root for $A[i\,..\,k]$. Without loss of generality, suppose $j < j'$. Then j' is also a forward split for the smaller interval $A[j+1\,..\,k]$.

Let T_0 be a mangled binary search tree for $A[i\,..\,k]$ with root $A[j]$, and let $A[r]$ be the root of the right subtree of T_0. Then r is a split for $A[j+1\,..\,k]$. Because j' is a *forward* split for that same interval, Lemma 1 implies that r is a *forward* split. Moreover, r is (by definition) a valid root for $A[j+1\,..\,k]$. The induction hypothesis now implies that j' is a valid root for $A[j+1\,..\,k]$.

It follows that there is a mangled binary search tree T for $A[i\,..\,k]$ where the root stores $A[j]$ and the right child of the root stores $A[j']$. Let T' be the tree obtained from T by rotating the right child of the root up to the root. T' is a mangled binary search trees for the array $A[i..k]$ (with exactly the same mangled nodes as T) with $A[j']$ at the root. We conclude that j' is a valid root for $A[i..k]$. □

It follows that if even one forward split is *not* a valid root, then *no* forward split is a valid root. A similar argument implies that some backward split $A[i..k]$ is a valid root if and only if every backward split is a valid root.

These observations imply the following simpler recurrence for the function *Garbled*:

$$Garbled(i,k) = \begin{cases} \text{TRUE} & \text{if } i \geq k \\ \text{FALSE} & \text{if there are no good splits} \\ Garbled(i, j-1) \wedge Garbled(j+1, k) & \text{for an arbitrary good split } j \end{cases}$$

Crucially, this recursive algorithm identifies a *single* good split j, *assumes* that j is a valid root, and recursively tries to construct left and right subtrees under that assumption. If either of the recursive calls returns *False*, then j is *not* a valid root, and then Lemma 2 implies that there are no valid roots at all.

Because this recursive greedy algorithm never backtracks, there is no need to memoize anything.

Our greedy algorithm spends linear time looking for a good split and then makes two recursive calls. We have no control over the sizes of the two recursive subproblems—the good splits are wherever they are—so the running time obeys the quicksort recurrence:

$$T(n) \leq O(n) + \max_j \left\{ T(j-1) + T(n-j) \right\}$$

We conclude that our greedy algorithm runs in $\boldsymbol{O(n^2)}$ ***time.*** In the worst case, the garbled binary tree is just a path, in which every other node has flipped children; then the interval for each subtree has only one good root, at the beginning or end of that interval. For example: $[1,3,5,7,9,11,13,15,17,16,14,12,10,8,6,4,2]$.

Rubric: +10 extra credit, but **only with the proof of correctness**. "Partial extra credit" at Jeff's discretion. This is not the fastest algorithm for this problem; see below!

- Lemma 2 does *not* hold if we define "good split" as in the "check root *position*" solution above. For example, if the input array is $[1,4,3,2,5]$, then *every* index is a "good split by position", but only 1 and 5 are valid roots.
- The proof of Lemma 2 is an example of an *exchange argument*, which is the most common method for proving greedy algorithms correct. See Chapter 4 for more information about greedy algorithms and exchange arguments.

Solution (boom goes the dynamite): With even more work, we can actually decrease the running time of previous algorithm to $O(n \log n)$! The trick is to use preprocessing to decrease the time require to identify good splits.

Again, let $A[1..n]$ be the initial input array, which we have been promised is a permutation of the integers 1 through n. We construct two new arrays $L[0..n]$ and $R[1..n+1]$ as follows:

```
InitRL(A[1..n]):
    $L[0] \gets 0$
    for $i \gets 1$ to $n$
        $L[i] \gets A[i] + L[i-1]$
    $R[n+1] \gets 0$
    for $i \gets n$ down to 1
        $R[i] \gets A[i] + R[i+1]$
```

Now consider an interval $A[i..k]$ that stores a permutation of some $k-i+1$ consecutive integers, say m through $m+k-i$.

- Index j is a good *forward* pivot for $A[i..k]$ if and only if $A[j] = m+j-i$ ***and*** the prefix $A[i..j-1]$ stores a permutation of the integers m through $m+j-i-1$. The second condition is equivalent to

$$L[j-1] - L[i-1] \;=\; \sum_{\ell=0}^{j-i-1}(m+\ell) \;=\; (j-i-1)m + \binom{j-1}{2}$$

- Similarly, index j is a good *backward* pivot for $A[i..k]$ if and only if $A[j] = m+k-j+1$ ***and*** the suffix $A[j+1..k]$ stores a permutation of the integers m through $m+k-j$. The second condition is equivalent to

$$L[k] - L[j] \;=\; \sum_{\ell=0}^{k-j}(m+\ell) \;=\; (k-j)m + \binom{k-j+1}{2}$$

Because we already computed $L[\cdots]$ and $R[\cdots]$, both of these conditions can be checked in $O(1)$ time, given the indices i, j, k and the minimum value m.

The following recursive algorithm takes the first and last indices of an interval $A[i..k]$ and the minimum value $m = \min A[i..k]$ as input, and returns True if and only if $A[i..k]$ is the inorder sequence of a garbled binary tree for the integers m through $m+k-i+1$. We need to compute Garbled$(1, n, 1)$.

```
GARBLED($i, k, m$):
  if $k \le i$
      return TRUE
  for $j = 1$ to $k - i$
      if $j$ is a good forward pivot for $A[i\,..\,k]$
          return GARBLED($i, j-1, m$) $\wedge$ GARBLED($j+1, k, m+j-i$)
      if $j$ is a good backward pivot for $A[i\,..\,k]$
          return GARBLED($i, j-1, m+k-j$) $\wedge$ GARBLED($j+1, k, m$)
      $j' \leftarrow k - j + 1$
      if $j'$ is a good forward pivot for $A[i\,..\,k]$
          return GARBLED($i, j'-1, m$) $\wedge$ GARBLED($j'+1, k, m+j'-i$)
      if $j'$ is a good backward pivot for $A[i\,..\,k]$
          return GARBLED($i, j'-1, m+k-j$) $\wedge$ GARBLED($j'+1, k, m$)
  return FALSE
```

The algorithm scans the interval inward from both ends until it finds a good pivot, declares that good pivot to be the root, and recursively tries to build the left and right subtrees under that assumption. Now the time to find a good pivot is proportional to the size of the *smaller* recursive subproblem, rather than the entire interval. Thus, ignoring big-Oh constants, the running time of the algorithm obeys the recurrence

$$T(n) \le \max_j \big(T(j) + T(n-j) + \min\{j, n-j\}\big)$$

(Yeah, to be precise, those js should be $(j-1)$s, but I'm not trying to be precise.) Now I claim that the solution to this recurrence is $T(n) \le n \log_2 n$. We can verify this claim by induction as follows:

$$\begin{aligned}
T(n) &\le \max_{1\le j\le n} \big(T(j) + T(n-j) + \min\{j, n-j\}\big) \\
&\le \max_{1\le j\le n/2} \big(T(j) + T(n-j) + j\big) && \text{by symmetry} \\
&\le \max_{1\le j\le n/2} \big(j\log_2 j + (n-j)\log_2(n-j) + j\big) && \text{by ind. hyp.} \\
&\le \max_{1\le j\le n/2} \big(j\log_2(\boldsymbol{n/2}) + (n-j)\log_2 \boldsymbol{n} + j\big) && \text{because } 0 \le j \le n/2 \\
&= \max_{1\le j\le n/2} \big(j(\log_2 n - 1) + (n-j)\log_2 n + j\big) \\
&= \max_{1\le j\le n/2} \big(n\log_2 n\big) \\
&= n\log_2 n
\end{aligned}$$

We conclude that the modified greedy algorithm runs in ***$O(n\log n)$ time.***

The *worst* case for this algorithm is a *perfectly balanced* binary tree, where every node with odd depth has flipped children, and at every level of recursion (sufficiently far from the base case), the root is the middle element of its interval. For example: $[21, 22, 23, 20, 17, 18, 19, 24, 29, 30, 31, 28, 25, 26, 27, 16, 5, 6, 7, 4, 1, 2, 3, 8, 13, 14, 15, 12, 9, 10, 11]$. ■

Rubric: Have you thought about graduate school? (I'm sure there's a cleaner way to do this. I have no reason to believe that further improvements are impossible.)