CS 598 JGE Spring 2025 Homework 0 Problem 1 Solution

Solution: (a) The effect of an increment on the runnary representation of an integer N depends on whether it has one run, a larger odd number of runs, or an even number of runs, and whether certain runs near the end of the sequence have length 1. In each of the following cases, "…" represents an arbitrary number of runs (possibly zero) that do not change.

one run	*even # runs*	*odd # runs* > 1
$[z] \mapsto [1,z]$	$[\ldots,y,1] \mapsto [\ldots,y+1]$	$[\ldots,x,1,z] \mapsto [\ldots,x+1,z]$
	$[\ldots,z] \mapsto [\ldots,z-1,1]$	$[\ldots,y,z] \mapsto [\ldots,y-1,1,z]$

Because we'll need it in part (b), I'll also implement decrement, which has a nearly identical set of cases (assuming the input number is strictly greater than 1):

two runs	*odd # runs*	*even # runs* > 2
$[1,z] \mapsto [z]$	$[\ldots,y,1] \mapsto [\ldots,y+1]$	$[\ldots,x,1,z] \mapsto [\ldots,x+1,z]$
$[y,z] \mapsto [y-1,1,z]$	$[\ldots,z] \mapsto [\ldots,z-1,1]$	$[\ldots,y,z] \mapsto [\ldots,y-1,1,z]$

To implement these cases efficiently, we store the runnary representation in a *stack* of run-lengths, with the least significant runs on top, that supports the operations PUSH, POP, and SIZE (= number of runs). We can implement the stack using a singly-linked list, or if we're happy with amortized time bounds, using a dynamic array that doubles in size whenever it becomes full and halves in size whenever it becomes less than 1/4 full. The following algorithms run in ***O*(1) *time***.

```
RUNNARYINC(N):
  if SIZE(N) = 1
      z ← POP(N)
      PUSH(N, 1)
      PUSH(N, z)
  else if SIZE(N) is even
      z ← POP(N)
      if z = 1
          PUSH(N, POP(N) + 1)
      else
          PUSH(N, z − 1)
          PUSH(N, 1)
  else ⟨⟨if SIZE(N) is odd > 1⟩⟩
      z ← POP(N)
      y ← POP(N)
      if y = 1
          PUSH(N, POP(N) + 1)
      else
          PUSH(N, y − 1)
          PUSH(N, 1)
      PUSH(N, z)
```

```
RUNNARYDEC(N):
  if SIZE(N) = 2
      z ← POP(N)
      y ← POP(N)
      if y > 1
          PUSH(N, y − 1)
          PUSH(N, 1)
      PUSH(N, z)
  else if SIZE(N) is odd
      z ← POP(N)
      if z = 1
          PUSH(N, POP(N) + 1)
      else
          PUSH(N, z − 1)
          PUSH(N, 1)
  else ⟨⟨if SIZE(N) is even > 2⟩⟩
      z ← POP(N)
      y ← POP(N)
      if y = 1
          PUSH(N, POP(N) + 1)
      else
          PUSH(N, y − 1)
          PUSH(N, 1)
      PUSH(N, z)
```

Alternatively, we can delegate some cases to the recursion fairy:

```
RunnaryInc(N):
  if Size(N) = 0
      Push(N, 1)
  else if Size(N) is even
      z ← Pop(N)
      if z > 1
          Push(N, z − 1)
          Push(N, 1)
      else
          Push(N, Pop(N) + 1)
  else ⟨⟨if Size(N) is odd⟩⟩
      z ← Pop(N)
      RunnaryInc(N)
      Push(N, z)
```

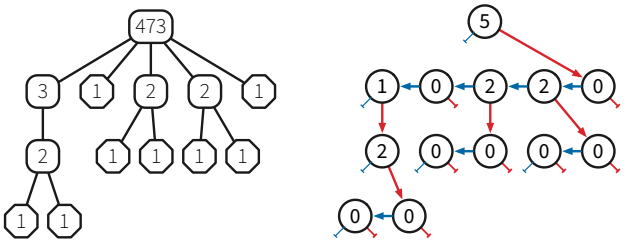
```
RunnaryDec(N):
  if Size(N) is odd
      z ← Pop(N)
      if z > 1
          Push(N, z − 1)
          Push(N, 1)
      else if Size(N) > 0
          Push(N, Pop(N) + 1)
  else ⟨⟨if Size(N) is even⟩⟩
      z ← Pop(N)
      RunnaryDec(N)
      Push(N, z)
```

Alternatively, if we keep a sentinel ∞ at the bottom of the stack (with the semantics $\infty + 1 = \infty - 1 = \infty$), we can simplify to just two cases

- increment even or decrement odd: $[\ldots, z] \mapsto [\ldots, z-1, 1]$
- increment odd or decrement even: $[\ldots, y, z] \mapsto [\ldots, y-1, 1, z]$

followed by a cleanup phase $x, 0, 1 \mapsto x + 1$ to remove any zeros from the last four runs.

(b) For each node v in the recursive runnary representation, we store the parity $v.odd$ of the number represented by its subtree, a pointer $v.right$ to its rightmost child (if any), and a pointer $v.left$ to its next sibling to the left (if any). The left-sibling pointers mirror a linked-list representation of the stack in part (a); in particular, a node with no rightmost child represents the integer 1.



Left: Abstract recursive runnary representation of 473
Right: Concrete representation; right-child pointers are red; left-sibling pointers are blue.

We need two mutually recursive algorithms ReRunnaryInc and ReRunnaryDec, which respectively increment and decrement a recursive runnary number. ReRunnaryDec is only called on integers greater than 1. Both algorithms take a node in a rerunnary tree as input. The interesting work happens in the helper function Tweak. The function NewOne( ) allocates and returns a new node v with $v.left = v.right =$ Null and $v.odd =$ True, representing the integer 1; unused nodes are freed via garbage collection.

```
IsOne(v):
  return v.right = Null
```

```
IsTwo(v):
  return (v.right ≠ Null
          and v.right.right = Null
          and v.right.left ≠ Null
          and v.right.left.right = Null)
```

```
TWEAK(y):
  if ISONE(y)          ⟪[..., x, 1] ↦ [..., x+1]⟫
    RERUNNARYINC(y.left)
    return y.left
  else                 ⟪[..., y] ↦ [..., y−1, 1]⟫
    RERUNNARYDEC(y)
    z ← NEWONE( )
    z.left ← y
    return z
```

```
RERUNNARYINC(v):
  if v = NULL          ⟪∞ + 1 = ∞⟫
    return
  else if ISONE(v)     ⟪1 ↦ [1, 1]⟫
    v.right ← NEWONE( )
    v.right.left ← NEWONE( )
    v.odd ← FALSE
  else if v.odd
    v.right.left ← TWEAK(v.right.left)
    v.odd ← FALSE
  else
    v.right ← TWEAK(v.right)
    v.odd ← TRUE
```

```
RERUNNARYDEC(v):
  if v = NULL          ⟪∞ − 1 = ∞⟫
    return
  else if ISTWO(v)     ⟪[1, 1] ↦ 1⟫
    v.right ← NULL
    v.odd ← TRUE

  else if ¬v.odd
    v.right.left ← TWEAK(v.right.left)
    v.odd ← TRUE
  else
    v.right ← TWEAK(v.right)
    v.odd ← FALSE
```

(c) Let $depth(N)$ denote the depth of the recursive runnary representation of N. The binary representation of N has exactly $\lceil \log_2(N+1) \rceil$ bits, so each run trivially has length at most $\lceil \log_2(N+1) \rceil$. Thus,

$$depth(N) \le 1 + depth(\lceil \log_2(N+1) \rceil).$$

It follows that $depth(N) \le O(\log^* N)$, where $\log^*$ is the iterated logarithm:

$$\log^* N = \begin{cases} 0 & \text{if } N \le 1 \\ 1 + \log^*(\log_2 N) & \text{otherwise} \end{cases}$$

(The ceiling and +1 in the depth recurrence makes no difference asymptotically.) Thus, RERUNNARYINC(N) and RERUNNARYDEC(N) both run in at most $\boldsymbol{O(\log^* N)}$ ***time***.

I'll first prove that this analysis is tight for RERUNNARYDEC. Consider the following recursive sequence:

$$W_d = \begin{cases} 1 & \text{if } d = 0 \\ 2 & \text{if } d = 1 \\ 2^{W_{d-1}} - 1 & \text{otherwise} \end{cases}$$

The first several values in this sequence are $1, 2, 3, 7, 127, 2^{127}-1, \ldots$.[1] For any $d \ge 2$, The runnary representation of W_d is $[W_{d-1}]$, so by induction, the rerunnary representation of W_d is a path of length $d-1$, whose last node has two leaf children. Specifically, we have

$$depth(W_d) \;=\; 1 + depth(W_{d-1}) \;=\; 1 + depth(\log_2(W_d + 1))$$

for all $d \ge 3$, which implies $depth(W_d) = d = \Theta(\log^* W_d)$. (The extra +1 in the recurrence makes no difference asymptotically.)

[1] OEIS calls this sequence, except for the initial 1, the Catalan-Mersenne numbers. It is an open question whether all of these numbers (except of course 1) are prime.

For all $d \geq 3$, the runnary representation of $W_d - 1$ is $[W_{d-1} - 1, 1]$, so ReRunnaryDec(W_d) must recursively call ReRunnaryDec(W_{d-1}). It follows that ReRunnaryDec(W_d) runs in time $\Omega(depth(W_d)) = \Omega(d) = \Omega(\log^* W_d)$. Our earlier upper bound is tight!

Symmetrically, ReRunnaryInc(N) runs in $\Theta(\log^* N)$ time when $N = W_d - 1$. ■

CS 598 JGE Spring 2025 Homework 0 Problem 2 Solution

Describe and analyze an algorithm to transform an arbitrary binary tree with distinct node values into a binary search tree, using only rotations and swaps.

Solution (divide and conquer):

```
TreeSort(root):
  if root = Null then return
  stop ← parent(root)
  ⟨⟨—— Convert tree to rightward path ——⟩⟩
  v ← root
  while v ≠ Null
      while left(v) ≠ Null
          u ← left(v); Rotate(u); v ← u
      v ← right(v)
  ⟨⟨—— Push median node to the bottom ——⟩⟩
  m ← node with median value
  Swap(m)
  v ← m
  while left(v) ≠ Null
      u ← left(v); Rotate(u); v ← u
  ⟨⟨—— Rotate median to root and partition ——⟩⟩
  while parent(m) ≠ stop
      p ← parent(m)
      if value(p) > value(m) then Swap(m)
      Rotate(m)
      if value(p) > value(m) then Swap(m)
  ⟨⟨—— Recurse! ——⟩⟩
  TreeSort(left(m)); TreeSort(right(m))
```

First, we convert the tree to a rightward path. Specifically, as long as there is a node on the right spine that has a left child, rotate that child upward. Each such rotation increases the length of the right spine by 1, so this phase ends after at most $n-1$ rotations.

In the second phase, we move the median node m to the *bottom* of this path. First we swap the subtrees at m, so that m is the rightmost node. Then we repeatedly rotate to convert the tree back to a rightward path. At all times during this phase, the tree contains at most one node v with a left child, so we don't need a nested loop. This phase requires one swap and at most $n-1$ rotations.

In the third phase, we rotate m up to the root of the tree. This rotation moves the parent of m into m's left subtree, so if the parent's value is larger than m's value, we swap the subtrees of m before and after the rotation. By induction, all nodes in m's left subtree have smaller value than m, and all nodes in m's right subtree have larger value than m. This phase requires exactly $n-1$ rotations and at most $2n-2$ swaps. We use the *stop* node to avoid escaping the scope of any recursive calls.

Finally, we recursively sort the left and right subtrees, each of which has about $n/2$ nodes. Ignoring floors and ceilings, the overall number of operations satisfies the recurrence $T(n) = 2T(n/2) + 5n - 5$. We conclude that the algorithm uses ***at most*** $\mathbf{5n \log_2 n + O(n) = O(n \log n)}$ operations overall. (This is about five times faster than the next solution.) ■

Solution (exchange with parent): The following incremental strategy uses $O(n \log n)$ operations in the worst case. Without loss of generality, assume the values in the nodes are the integers 1 through n.

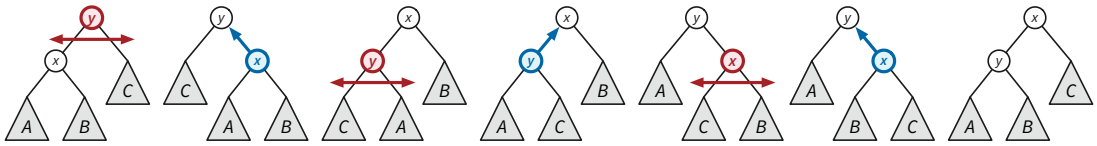
```
TreeSort(T):
  make T perfectly balanced
  for i ← 1 to n
      exchange the node with value i with the node in position i
```

In the preprocessing phase, we perform $O(n)$ rotations to make the T perfectly balanced as possible. Specifically, we first change T to a path of left children as follows: While any node on the left spine of T has a right child, rotate that child to the left. Each left rotation increases the length of the left spine by 1, so the while loop ends after at most $n-1$ rotations. By running the same algorithm backward in time, we can transform the leftward path into a perfectly balanced tree in at most $n-1$ more rotations.

To complete the description of the algorithm, we need to describe how to swap two arbitrary nodes (just the nodes, not their subtrees) in a balanced binary tree using at most $O(\log n)$ operations.

We can exchange any node in T with its parent using three swaps and three rotations, as follows. (In fact, there are several different sequences of six operations that will perform this exchange, but nothing shorter; see the graph on the next page.)

```
ExchangeWithParent(x):
  y ← parent(x)        ⟨⟨ (AxB)yC ⟩⟩
  Swap(y)              ⟨⟨ Cy(AxB) ⟩⟩
  Rotate(x)            ⟨⟨ (CyA)xB ⟩⟩
  Swap(y)              ⟨⟨ (AyC)xB ⟩⟩
  Rotate(y)            ⟨⟨ Ay(CxB) ⟩⟩
  Swap(x)              ⟨⟨ Ay(BxC) ⟩⟩
  Rotate(x)            ⟨⟨ (AyB)xC ⟩⟩
```



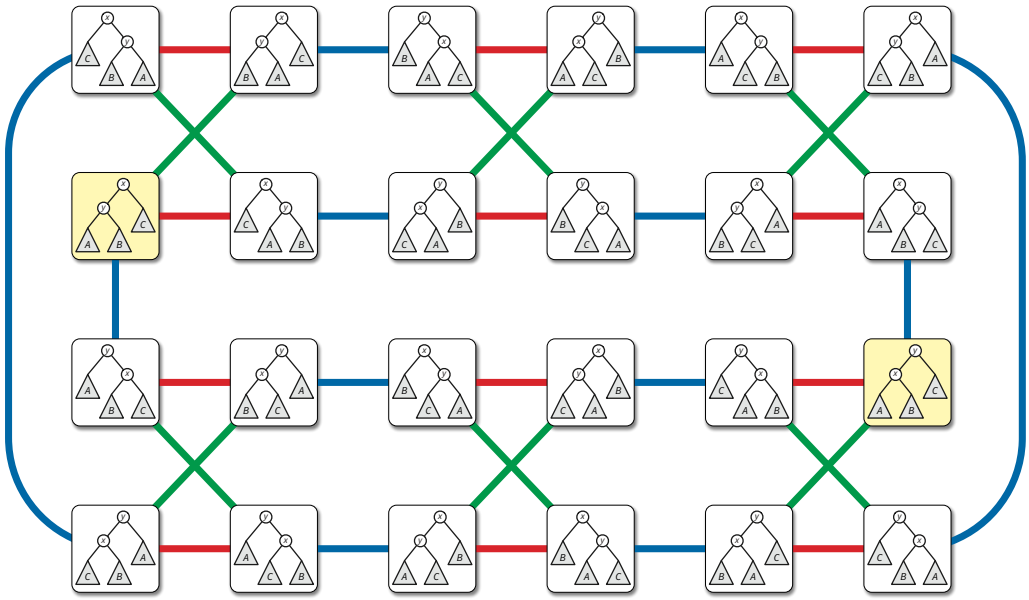
Exchanging a node with its parent.

Now consider arbitrary simple path $w_0 \to w_1 \to w_2 \to \cdots \to w_\ell$ in T; each node w_i is either the parent or a child of its successor w_{i+1}. We can exchange the endpoints w_0 and w_ℓ of this path as follows. First, for each i from 1 to ℓ, exchange w_0 with w_i. Then, for each i in decreasing order from $\ell-1$ to 1, exchange w_ℓ with w_i. The total number of node-exchanges is $2\ell-1$, and therefore the total number of rotations and swaps is $12\ell-6$. Finally, because we balanced T in the preprocessing phase, any two nodes in T are connected by a path of length $O(\log n)$. It follows that we can swap any two nodes in T in $O(\log n)$ steps, as required.

Alternatively, we can exchange any two nodes x and y in T using $O(\log n)$ rotations and only $O(1)$ swaps as follows, assuming without loss of generality that x is *not* a descendant of y. Rotate x upward until it is an ancestor of y; rotate y upward until it is a child of x; exchange x and y; and finally, undo all the rotations that brought x and y together. Using this second

strategy to exchange nodes, TREESORT uses a total of ***O(n log n) rotations and O(n) swaps*** in the worst case.

Alternatively, if we move nodes into place level by level from the leaves upward (or according to a postorder traversal) instead of in sorted order by value, we can cut the total number of operations roughly in half. Instead of exchanging two nodes x and y, we now only have to move x to y's position without disturbing the *shape* of the tree. This strategy moves every node along the path from x to y, but except for x, all those nodes will be moved into place later. ■



The graph of all possible configurations of two adjacent nodes in a node-labeled binary tree.
Blue edges indicate rotations. Red edges indicate swaps at the parent. Green edges indicate swaps at the child.
Exchanging a node with its parent requires six steps.

Matching lower bound: The $O(n \log n)$ upper bound in the previous solutions is actually optimal! A matching $\Omega(n \log n)$ worst-case lower bound follows from a result of Fredman [1]. Fredman considered an operation he calls a *non-standard rotation*, which swaps the child pointers of a node before rotating it upward. (These non-standard rotations are used in some efficient priority-queue and dynamic prefix-coding data structures.) Fredman defines a "rotation graph" whose nodes are labeled binary trees and whose edges correspond to standard and non-standard rotations, and then proves that this graph has diameter $\Theta(n \log n)$. The $\Omega(n \log n)$ lower bound ultimately comes from the number of bits required to encode a permutation of n items; I'll discuss further details in class later in the semester.

For any node v with parent p, we can perform a swap at v using a non-standard rotation at v followed by a standard rotation at p. We can also swap the children of the root with $O(1)$ standard and non-standard rotations, but the details are more complex. Thus, the "rotation and swap graph" $\boldsymbol{G}_n$, whose nodes are labeled n-node binary trees and whose edges correspond to rotations and swaps, also has diameter $\Theta(n \log n)$. On the other hand, any binary *search* tree can be converted into any other binary *search* tree with the same n keys using $O(n)$ rotations. (In fact, we need at most $2n - 6$ rotations, and this bound is tight in the worst case for all $n \geq 9$ [2,3].)

Thus, the subgraph of $\boldsymbol{G}_n$ induced by binary *search* trees has diameter $O(n)$. It follows that some tree is $\Omega(n \log n)$ edges away from a binary search tree.

[1] Michael L. Fredman. Generalizing a theorem of Wilber on rotations in binary search trees to encompass unordered binary trees. *Algorithmica* 62(3):863–878, 2012.

[2] Lionel Pournin. The diameter of associahedra. *Advances in Mathematics* 259:13–42, 2014. arXiv:1207.6296.

[3] Daniel D. Sleator, Robert E. Tarjan., and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *Proc. 18th STOC* 122–135, 1986.

CS 598 JGE Spring 2025 Homework 0 Problem 3 Solution

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: I'll develop the algorithm in four stages.

- First I'll describe a straightforward dynamic programming algorithm that runs in $O(n^4)$ time.
- Then I'll improve the running time to $O(n^3)$ by precomputing all values of a helper function, using a second dynamic-programming algorithm. (This is the target solution.)
- Next I'll describe a greedy optimization that reduces the algorithm from dynamic programming to divide-and-conquer, further improving the running time to $O(n^2)$.
- Finally, I'll improve the running time of the greedy divide-and-conquer algorithm to $O(n \log n)$ using careful preprocessing and more careful recursive analysis.

Yes, I realize that this violates my usual three-page limit, but in this case, I think jumping straight to the punchline would be less clear.

Basic dynamic programming: 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:

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

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 $\boldsymbol{O(n^4)}$ ***time***. ■

Preprocessing good splits: For any fixed indices i and k, we can evaluate $\mathit{GoodSplit}(i, j, k)$ 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.)

```
⟨⟨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 ← −∞; min ← +∞
  for j ← i to k
      if A[j] > max
          max ← A[j]; Lmax[j] ← True
      else
          Lmax[j] ← False
      if A[j] < min
          min ← A[j]; Lmin[j] ← True
      else
          Lmin[j] ← False
  ⟨⟨Find all j where either A[j] = max A[j..k] or A[j] = min A[j..k]⟩⟩
  max ← −∞; min ← +∞
  for j ← k down to 1
      if A[j] > max
          max ← A[j]; Rmax[j] ← True
      else
          Rmax[j] ← False
      if A[j] < min
          min ← A[j]: Rmin[j] ← True
      else
          Rmin[j] ← False
  ⟨⟨Avengers, assemble!⟩⟩
  for j ← i to k
      SplitHere[j] ← (Lmax[j] ∧ Rmin[j]) ∨ (Lmin[j] ∧ Rmax[j])
  return SplitHere[i..k]
```

For any indices i and k, after calling FindGoodSplits(i, k), we can evaluate $\mathit{Garbled}[i, k]$ in only $O(n)$ time. As a result, our overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

Greedy divide and conquer: 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 we call j 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 base case, no interval contains both types of splits.

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

Proof: Fix arbitrary indices $i \le k$. (Yes, $\le$ not $<$). Suppose some index j is a forward split for $A[i..k]$ and some index j' is a backward split for $A[i..k]$. To prove the lemma, it suffices to show that $i = j = j' = k$.

The minimum element $\min A[i..k]$ must lie in the prefix $A[i..j]$, because j is a forward split, but it must also lie in the suffix $A[j'..k]$, because j' is a backward split. It follows that $j' \le j$.

Symmetrically, the maximum element $\max A[i..k]$ must lie in both the suffix $A[j..k]$ and the prefix $A[i..j']$, which implies $j \le j'$. It follows that $j = j'$. So the same index j is both a forward split and a backward split.

It follows that $A[j] = \min A[i..j] = \max A[i..j]$, which is only possible if $j = i$. But symmetrically, we have $A[j] = \min A[j..k] = \max A[j..k]$, which is only possible if $j = k$. We conclude that $i = k$, which completes the proof. □

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 root 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 $A[j+1..k]$. To prove the lemma, we need to prove that j' is also a valid root.

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.[2] This new tree T' is a mangled binary search tree for the array $A[i..k]$ (with exactly the same swapped 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*:

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

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!

[2] This is an example of an *exchange argument*, which is the standard method for proving greedy algorithms correct. See Chapter 4 of Jeff's textbook!

Because this recursive algorithm never backtracks, there is no need to memoize anything. We've changed from dynamic programming to greedy divide-and-conquer!

Our divide-and-conquer 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 standard quicksort recurrence:

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

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: $[1, 3, 5, 7, 9, 11, 13, 15, 17, 16, 14, 12, 10, 8, 6, 4, 2]$.

Boom goes the dynamite: Finally, we can decrease the running time of our divide-and-conquer algorithm to $O(n \log n)$ by preprocessing the input array to identify good splits more quickly (the "divide" part) and by being more careful with our recursive analysis.

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 preprocess the array A for $O(1)$-time range-minimum queries, as described in class; we also symmetrically preprocess A for $O(1)$ time range-*maximum* queries. If we use sparse lookup tables, the total preprocessing time is $O(n \log n)$. (There's no point in building more complex $O(n)$-space data structures, because the rest of the algorithm takes $\Theta(n \log n)$ time in the worst case.)

Now consider an interval $A[i..k]$ that stores a permutation of some $k - i + 1$ consecutive integers, say m through $m + k - i$. We can compute m in $O(1)$ time using a single range-minimum query.

- Index j is a good *forward* pivot for $A[i..k]$ if and only if $A[j] = m + j - i$ ***and*** $\min A[i..j] = m$ ***and*** $\max A[i..j] = m + j + 1$. The min and max conditions hold if and only if $A[i..j]$ stores a permutation of the integers m through $m + j - i$.

- Similarly, index j is a good *backward* pivot for $A[i..k]$ if and only if $A[j] = m + k - j$ ***and*** $\min A[j..k] = m$ ***and*** $\max A[j..k] = m + k - j$. The min and max conditions hold if and only if $A[j..k]$ stores a permutation of the integers m through $m + k - j$.

Both of these conditions can be checked in $O(1)$ time, given the indices i, j, and k, using our range-minimum and range-maximum data structures.

The following recursive algorithm takes the first and last indices of an interval $A[i..k]$ that is guaranteed to contain $j - i + 1$ consecutive integers, and returns TRUE if and only if $A[i..k]$ is the inorder sequence of a garbled binary tree. We need to compute GARBLED$(1, n)$.

```
GARBLED(i, k):
  if k ≤ i
      return TRUE
  for j = i to k
      if j is a good forward pivot for A[i..k]
          return GARBLED(i, j−1) ∧ GARBLED(j+1, k)
      if j is a good backward pivot for A[i..k]
          return GARBLED(i, j−1) ∧ GARBLED(j+1, k)
      j' ← k − j + 1
      if j' is a good forward pivot for A[i..k]
          return GARBLED(i, j'−1) ∧ GARBLED(j'+1, k)
      if j' is a good backward pivot for A[i..k]
          return GARBLED(i, j'−1) ∧ GARBLED(j'+1, k)
  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. 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

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

We can verify that the solution to this recurrence is $T(n) \le n \lg n$ (where lg means $\log_2$ as usual) by induction as follows:

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

We conclude that our modified greedy divide-and-conquer algorithm runs in $\boldsymbol{O(n \log n)}$ ***time.***

The *worst* input 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: $[21, 22, 23, 20, 17, 18, 19, 24, 29, 30, 31, 28, 25, 26, 27, 16, 5, 6, 7, 4, 1, 2, 3, 8, 13, 14, 15, 12, 9, 10, 11]$. ■