1. The following variant of the infamous StoogeSort algorithm was discovered by the British actor Patrick Troughton during rehearsals for the 20th anniversary *Doctor Who* special "The Five Doctors".

```
WhoSort(A[1..n]):
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      WhoSort(A[1..3k])          ⟨⟨Hartnell⟩⟩
      WhoSort(A[2k+1..n])        ⟨⟨Troughton⟩⟩
      WhoSort(A[1..3k])          ⟨⟨Pertwee⟩⟩
      WhoSort(A[k+1..4k])        ⟨⟨Davison⟩⟩
```

(a) Prove by induction that WhoSort correctly sorts its input. *[Hint: Where can the smallest k elements be?]*

Solution: Let n be an arbitrary no-negative integer, let $A[1..n]$ be an arbitrary array, and assume that WhoSort correctly sorts any array of size less than n. Assume that $n \geq 13$, since otherwise correctness is trivial.

Intuitively, WhoSort partitions the input array A into five chunks, four of size $k = \lceil n/5 \rceil$ and one of size $n - 4k \leq n/5 \leq k$:

$$A[1..k], \quad A[k+1..2k], \quad A[2k+1..3k], \quad A[3k+1..4k], \quad A[4k+1..n]$$

The inequality $n \geq 13$ implies $4\lceil n/5 \rceil \leq n$, so all five chunks lie within the bounds of the original input array. (When $n = 16$, the last chunk is empty, but that's not a problem.) Each recursive call to WhoSort processes three consecutive chunks, which comprise an subarray of size at most $3k$. The assumption $n \geq 13$ also implies $3\lceil k/5 \rceil < n$, so by the induction hypothesis, the recursive calls correctly sort their respective subarrays.

Call an element of the input array ***small*** if it is one of the k smallest elements, ***large*** if it is one of the $n - 4k \leq k$ largest elements, and ***medium*** otherwise. Consider the locations of these classes of elements after each recursive call to WhoSort.

- Hartnell moves all small elements in the first three chunks to chunk 1, and moves all large elements in the first three chunks to chunk 3. Thus, after Hartnell's sort, all small elements are in chunks 1, 4, and 5, and all large elements are in chunks 3, 4, and 5.
- After Troughton's sort, all small elements are in chunks 1 and 3, and all large elements (and nothing else) are in chunk 5. Moreover, chunk 5 is sorted; thus, all large elements are in their correct final positions. The rest of the algorithm does not modify chunk 5.
- After Pertwee's sort, all small elements (and nothing else) are in chunk 1, in sorted order; thus, all small elements are in their correct final positions. The rest of the algorithm does not modify chunk 1. At this point, chunks 2, 3, and 4 contain all the medium elements and nothing else.

- Finally, Davison sorts chunks 2, 3, and 4, moving all the medium elements into their final correct positions.

We conclude that WHOSORT correctly sorts the array $A[1..n]$, as claimed. ■

Rubric: 6 points = 1 for explicitly considering an *arbitrary* input array + 1 for trivial base case + 1 for bounds checking (against parts (b) and (c)) + 1 for small elements + 1 for large elements + 1 for medium elements

(b) Would WHOSORT still sort correctly if we replaced "if $n < 13$" with "if $n < 4$"? Justify your answer.

Solution: ***No.*** When $n = 6$, the modified algorithm would fall into an infinite loop. Specifically, $k = \lceil 6/5 \rceil = 2$, so the first recursive call would attempt to recursively sort the entire array. ■

Solution: ***No.*** When $n = 11$, we have $k = \lceil 11/5 \rceil = 3$, so the last recursive call would attempt to recursively sort the "subarray" $A[4..12]$, which exceeds the bounds of the original input array. ■

Rubric: 1 point. One justification is necessary and sufficient.

(c) Would WHOSORT still sort correctly if we replaced "$k = \lceil n/5 \rceil$" with "$k = \lfloor n/5 \rfloor$"? Justify your answer.

Solution: ***No.*** The modified algorithm would fail when $n = 14$ (and therefore $k = 2$), because there is not enough overlap between the recursive subarrays to carry all large elements to the last chunk. For example, the algorithm would modify the array `NMLKJIHGFEDCBA` as follows:

Sort $A[1..6]$	`IJKLMNHGFEDCBA`
Sort $A[5..14]$	`IJKLABCDEFGHMN`
Sort $A[1..6]$	`ABIJKLCDEFGHMN`
Sort $A[3..8]$	`ABCDIJKLEFGHMN`

In contrast, the original algorithm (with $k = 3$) sorts the same array as follows.

Sort $A[1..9]$	`FGHIJKLMNEDCBA`
Sort $A[7..14]$	`FGHIJKABCDELMN`
Sort $A[1..9]$	`ABCFGHIJKDELMN`
Sort $A[4..12]$	`ABCDEFGHIJKLMN`

■

Rubric: 1 point. Full credit requires an explicit counterexample; ½ for intuitive argument about overlaps. This is not the only correct solution.

(d) What is the running time of WHOSORT? (Set up a running-time recurrence and then solve it, ignoring the floors and ceilings.)

Solution: The running time obeys the recurrence $T(n) = 4T(3n/5) + O(1)$. The level sums of the recursion tree form an increasing geometric series, so the solution is $T(n) = O(4^{\log_{5/3} n}) = O(n^{\log_{5/3} 4}) = O(n^{2.7138309})$. ■

Rubric: 2 points = 1 for recurrence + 1 for solution.

2. In the lab on Wednesday, we developed an algorithm to compute the median of the union of two sorted arrays size n in $O(\log n)$ time.

   But now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to compute the median of $A \cup B \cup C$ in $O(\log n)$ time. (You can assume the arrays contain $3n$ distinct integers.)

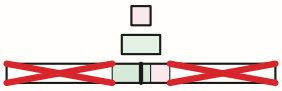
Solution: As in the lab problem, our main strategy is to repeatedly throw away two equal-size subarrays, one larger than the median and the other smaller than the median. In each iteration, we discard a constant fraction of at least one of the three arrays. Thus, after $O(\log n)$ iterations, all three arrays have constant size, at which point we can find the median by brute force.

Although the original input arrays all have the same size, our recursive calls will not keep the sizes equal, so we must explicitly allow different sized arrays. So suppose our input consists of three sorted arrays $A[1..a]$, $B[1..b]$, $C[1..c]$. Let $N = a+b+c$ and $M = \lceil N/2 \rceil$. We define the median of $A \cup B \cup C$ to be its Mth smallest element. Without loss of generality, we can assume $a \le b \le c$; if not, permute the variable names.

There are four cases to consider.

(i) ***Suppose $a+b+c \le 100$.*** In this case, we compute $median(A \cup B \cup C)$ by brute force in $O(1)$ time.

(ii) ***Otherwise, suppose $a+b \le 25$.*** In this case, the algorithm discards only elements of C, as suggested by the following figure.



Each element $C[i]$ has rank between i and $i+(a+b)$ in $A \cup B \cup C$, so

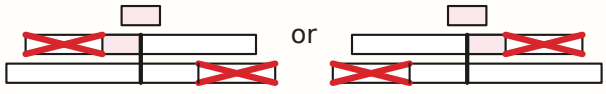
$$C[M-(a+b)] \;\le\; median(A \cup B \cup C) \;\le\; C[M].$$

The definition $M = \lceil (a+b+c)/2 \rceil$ implies that $M-(a+b) \ge c-M$. Thus, we can safely discard the top and bottom $c-M$ elements of C without changing the median:

$$median(A \cup B \cup C) \;=\; median(A \cup B \;\cup\; C[c-M+1..M]).$$

We compute the latter median recursively. The subarray $C[c-M+1..M]$ has size $2M-c \le N+1-c = a+b+1 \le 26$, so the recursive call fits case (i). Thus, the algorithm also runs in $O(1)$ time in case (ii).

(iii) ***Otherwise, suppose $a \le 5$.*** In this case, we discard the same number of elements from one end of B and the other end of C, as suggested by the following figure.



Let $j = \lceil b/2 \rceil$ and $k = \lceil c/2 \rceil$. As in part (a), there are two subcases to consider:

- Suppose $B[j] < C[k]$. In this case, we discard the bottom $j-a-1$ elements of B and the top $j-a-1$ elements of C:

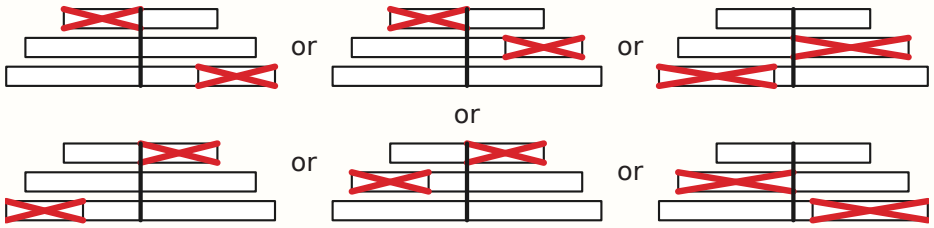
$$median(A \cup B \cup C) = median(A \cup B[j-a\,..\,b] \cup C[1\,..\,c-j+a+1]).$$

- If $B[j] > C[k]$, we discard the top $(b-j)-a$ elements of B and the bottom $(b-j)-a$ elements of C:

$$median(A \cup B \cup C) = median(A \cup B[1\,..\,j+a] \cup C[b-j-a+1\,..\,c]).$$

We must have $b > 20 \geq 4a$, since otherwise we would be in case (i) or (ii). Thus, in both subcases, we discard at least one fourth of the elements of B before recursing. It follows that we spend at most $O(\log b) = O(\log N)$ time in case (iii) before falling to case (i) or (ii).

(iv) ***Finally, suppose $a > 5$.*** In this case, we discard elements from some pair of arrays, as suggested by the following figure:



Let $i = \lceil a/2 \rceil$, $j = \lceil b/2 \rceil$ and $k = \lceil c/2 \rceil$, and sort the elements $A[i]$, $B[j]$, and $C[k]$. There are six nearly identical cases to consider.

- If $A[i] < B[j] < C[k]$, we discard the bottom $i-1$ elements of A and the top $i-1$ elements of C and recurse.
- If $A[i] < C[k] < B[j]$, we discard the bottom $i-1$ elements of A and the top $i-1$ elements of B and recurse.
- If $C[k] < A[i] < B[j]$, we discard the top $b-j$ elements of B and the bottom $b-j$ elements of C and recurse.
- If $C[k] < B[j] < A[i]$, we discard the top $a-i$ elements of A and the bottom $a-i$ elements of C and recurse.
- If $B[j] < C[k] < A[i]$, we discard the top $a-i$ elements of A and the bottom $a-i$ elements of B and recurse.
- If $B[j] < A[i] < C[k]$, we discard the bottom $j-1$ elements of B and the top $j-1$ elements of C and recurse.

In each case, we discard the same number of elements above and below $median(A \cup B \cup C)$, so the median element does not change. Moreover, in each case, we discard roughly half of one of the three arrays. Thus, we spend at most $O(\log N)$ time in this case before falling to one of the previous cases.

In all four cases, the algorithm runs in $O(\log N)$ time, as required. ■

Rubric: 10 points = 4 for correct algorithm + 4 for correctness argument + 2 for time analysis. This solution is more detailed than necessary for full credit. This is not the only correct solution; there are at least three other fruitful approaches:

- Change the case boundaries so that all three arrays decay together. If $c > 3b$, we can safely discard the top and bottom $c - M$ elements of C, as in case (ii) here. Similarly, if $3a < b$, we can safely discard $a - b/2$ elements from both B and C, as in case (iii) here. Otherwise, all three arrays are roughly the same size (specifically, $c \le 3b \le 9a \le 9c$), and we can proceed as in case (iv).
- Repeatedly apply case (iv) above until the smallest array has size $O(1)$. Compute the median of the other two arrays using the algorithm developed in lab, modified to handle different-size arrays. Then scan the arrays for $\mathit{median}(A \cup B \cup C)$ in $O(1)$ time.
- Instead of carefully balancing the sizes of the discarded subarrays, develop an algorithm that finds the rth smallest element of $A \cup B \cup C$ for arbitrary r. This variant requires slightly less bookkeeping, but still requires all four cases.

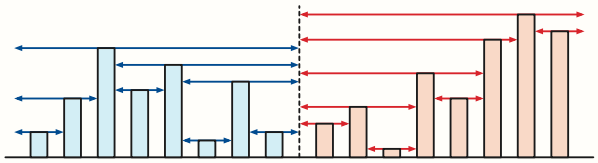
Ignore off-by-one errors in recursive arguments, but watch for the following bugs:

- Assuming (implicitly) that the sizes of the input arrays are equal in recursive calls.
- Incorrectly handling the case where one array has size 1 (or 0) and thus can't shrink any more.

3. *[See the homework handout for the complete problem statement.]*

(a) Describe a divide-and-conquer algorithm that computes the output of WHOTARGETSWHOM in $O(n \log n)$ time.

Solution (divide and conquer by position): Unless n is small enough to handle by brute force, we recursively compute targets within the left half and the right half of the input array, and then find all targets that cross the split. After the two recursive calls, the target assignments look like this:



Call a hero *unfinished* if they are in the left half and have no right target, or in the right half and have no left target. The correct right target for every unfinished left hero is either NONE or an unfinished right hero; similarly, the correct left target for every unfinished right hero is either NONE or an unfinished left hero.

```
WHOTARGETSWHOM(Ht[1..n]):
  if n < 1000    ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌊n/2⌋
      L[1..m], R[1..m] ← WHOTARGETSWHOM(Ht[1..m])
      L[m..n], R[m..n] ← WHOTARGETSWHOM(Ht[m..n])
      i ← m;  j ← m+1

      while i ≥ 1 and j ≤ n
          if R[i] ≠ NONE
              i ← i−1
          else if L[j] ≠ NONE
              j ← j+1
          else if Ht[i] < Ht[j]
              R[i] ← j
              i ← i−1
          else ⟨⟨if Ht[i] < Ht[j]⟩⟩
              L[j] ← j
              j ← j+1

      return L[1..n], R[1..n]
```

Each iteration of the while loop increases the difference $j - i$ by 1, so the loop ends after at most n iterations. Thus, the running time of WHOTARGETSWHOM obeys the mergesort recurrence $T(n) \leq 2T(n/2) + O(n)$, so the algorithm runs in $O(n \log n)$ time, as required.

Rubric: This is enough for full credit.

With more care, we can reduce the running time to $\boldsymbol{O(n)}$, using the following observation: The *left* target of any unfinished *left* hero is either NONE or

the nearest taller unfinished *left* hero. Symmetrically, the *right* target of any unfinished *right* hero is either NONE or the nearest taller unfinished *right* hero. Thus, we can use the recursively computed subarrays of L and R to short-circuit the while loop.

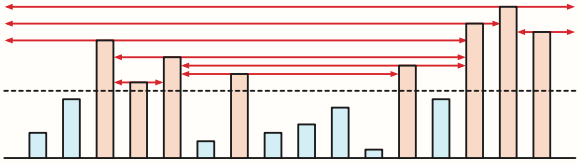
```
WHOTARGETSWHOM(Ht[1..n]):
  if n < 1000
      use brute force
  else
      m ← ⌊n/2⌋
      L[1..m], R[1..m] ← WHOTARGETSWHOM(Ht[1..m])
      L[m..n], R[m..n] ← WHOTARGETSWHOM(Ht[m..n])
      i ← m;  j ← m + 1

      while i ≥ 1 and j ≤ n
          if Ht[i] < Ht[j]
              R[i] ← j
              i ← L[i]
          else ⟨⟨if Ht[i] < Ht[j]⟩⟩
              L[j] ← i
              j ← R[j]

      return L[1..n], R[1..n]
```

Now each iteration of the while loop sets at least one target. Thus, the total number of iterations of the while loop, *summed over all recursive calls*, is at most $2n$. We conclude that the entire algorithm runs in $O(n)$ time. ■

Solution (divide and conquer by height): Unless n is small enough to handle by brute force, we find the median height (using the linear-time selection algorithm described in class), recursively compute targets for all the "tall" heros, and then recursively compute targets for each interval of "short" heroes between two "tall" heros. After the first recursive call, the target assignments look like this:



The recursive subroutine WHATTHEWHAT takes an array of heights that includes two sentinel values at the beginning and end that are larger than any other values. The output of WHATTHEWHAT is a pair of arrays that assigning targets to all heroes in the given interval except the sentinels. The sentinels guarantee that WHATTHEWHAT assigns *every* hero left and right targets, so the main algorithm WHOTARGETSWHOM must post-process the output of WHATTHEWHAT to replace sentinel targets with NONE.

```
WHOTARGETSWHOM(Ht[1..n]):
  Ht[0] ← ∞                          ⟨⟨add sentinels⟩⟩
  Ht[n+1] ← ∞
  L[1..n], R[1..n] ← WHATTHEWHAT(Ht[0..n+1])
  for i ← 1 to n                     ⟨⟨remove sentinels⟩⟩
      if L[i] = 0
          L[i] ← NONE
      if R[i] = n+1
          R[i] ← NONE
```

```
WHATTHEWHAT(Ht[0..n+1]):
  if n < 1000
      use brute force
  else
      m ← MEDIAN(Ht[1..n])

      ⟨⟨Recursively compute targets for the tall heroes⟩⟩
      j ← 0
      for i ← 0 to n+1
          if Ht[i] ≥ m
              TallHt[j] ← Ht[i]          ⟨⟨height of jth tall hero⟩⟩
              TallPos[j] ← i             ⟨⟨position (index) of jth tall hero⟩⟩
              j ← j+1
          t ← j−1                        ⟨⟨#tall heroes, excluding sentinels⟩⟩
          LT[1..t], RT[1..t] ← WHATTHEWHAT(TallHt[0..t+1])   ⟨⟨Recurse!⟩⟩
          for j ← 0 to t
              L[TallPos[j]] ← LT[t]
              R[TallPos[j]] ← RT[t]

      ⟨⟨Recursively compute targets for each interval of short heroes⟩⟩
      for j ← 0 to t
          ℓ ← TallPos[j]+1               ⟨⟨first short hero in jth interval⟩⟩
          r ← TallPos[j+1]−1             ⟨⟨last short hero in jth interval⟩⟩
          L[ℓ..r], R[ℓ..r] ← WHATTHEWHAT(Ht[ℓ−1..r+1])   ⟨⟨Recurse!⟩⟩

      return L[1..n], R[1..n]
```

The running time of this algorithm obeys the following recurrence, where n_j is the number of heroes between the jth and $(j+1)$th tall heroes.

$$T(n) = O(n) + T(n/2) + \sum_{j=0}^{t} T(n_j)$$

Because there are $n/2$ short heroes, we have $\sum_{j=0}^{t} n_j \le n/2$. Thus, *no matter how the heights are distributed*, the total time at each level of the recursion tree is at most $O(n)$. Moreover, because every recursive problem size is at most $n/2$, the recursion tree has at most $O(\log n)$ levels. We conclude that the overall running time is $O(n \log n)$, as required.

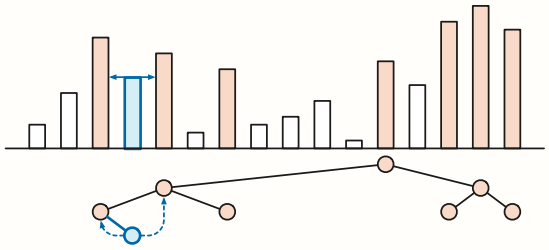
I will be *very* surprised if anyone actually does this. ■

Solution (iterative by position): When all targets are assigned, the left target of any hero with no right target is either NONE or the next taller hero with no right target. Thus, we can visit all heroes with no right targets, in increasing order by height, by following left-target pointers. The following algorithm treats this implicit linked list as a stack.

```
WHOTARGETSWHOM(Ht[1..n]):
  i ← 0                          ⟨⟨top of the stack⟩⟩
  for j ← 1 to n
      R[j] ← NONE
      if i = 0
          L[j] ← NONE
          i ← j                  ⟨⟨push j⟩⟩
          j ← j + 1
      else if Ht[j] < Ht[i]
          L[j] ← i
          i ← j                  ⟨⟨push j⟩⟩
          j ← j + 1
      else
          R[i] ← j
          i ← L[i]               ⟨⟨pop⟩⟩
  return L[1..n], R[1..n]
```

The algorithm clearly runs in ***O(n) time***!! ■

Solution (iterative by height): The left and right targets for any hero x are the predecessor and successor of x, among al heroes taller than x. Our algorithm inserts the *positions* of the heroes one at a time into an ordered dictionary, in decreasing order of *height*. After each insertion, we report the predecessor and successor of the newly inserted item.



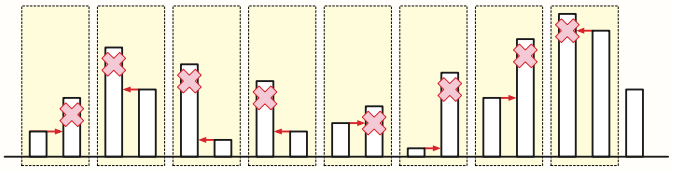
```
WHOTARGETSWHOM(Ht[1..n]):
  for i ← 1 to n
      Pos[i] ← i
  sort Ht downward and permute Pos to match
  ⟨⟨now Pos[i] is the position of the ith tallest hero⟩⟩
  D ← new ordered dictionary
  for j ← 1 to n
      L[Pos[j]] ← PREDECESSOR(D, Pos[j])
      R[Pos[j]] ← SUCCESSOR(D, Pos[j])
      INSERT(D, Pos[j])
  return L[1..n], R[1..n]
```

If we implement the ordered dictionary using a balanced binary search tree (for example, an AVL tree or a red-black tree), each insertion, predecessor query, and successor query takes $O(\log n)$ time, and thus the entire algorithm runs in ***$O(n \log n)$ time***. (If we just use an standard off-the-shelf binary search tree, the worst-case running time is $O(n^2)$.) ■

Rubric: 5 points = 3 for algorithm + 1 for justification + 1 for time analysis
- +2 extra credit for a correct $O(n)$-time algorithm.
- Max 3 points for a correct algorithm with running time between $\omega(n \log n)$ and $o(n^2)$; scale partial credit.
- Max 2 points for a correct algorithm that runs in $O(n^2)$ time; scale partial credit.
- Zero points for an incorrect $O(n)$-time algorithm.
- These are not the only correct solutions, for either $O(n \log n)$ time or $O(n)$ time! In particular, each of these four solutions has several minor variants.

(b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0..n-1]$ and $L[0..n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than NONE).

Solution: Split the array into $\lfloor n/2 \rfloor$ chunks of size 2. (When n is odd, one hero is not in any chunk; ignore them.) Within each chunk, the taller hero is a target of the shorter hero.



■

Rubric: 2 points.

(c) Describe an algorithm that computes the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in ***$O(n)$*** time.

Solution: To simplify the algorithm, we add two sentinel values $Ht[0] = Ht[n+1] = \infty$.

Fix an arbitrary index i between 1 and n. If $Ht[i] > Ht[i-1]$, then $R[i-1] = i$. Otherwise, for every index $j < i$, either $Ht[j] > Ht[i]$ or $Ht[j] < Ht[i-1]$, so $R[j] \neq i$. Symmetrically, i is a left target if and only if $Ht[i] > Ht[i+1]$. We conclude that the ith hero survives the first round if and only if $Ht[i]$ is a ***local minimum***: $Ht[i-1] > Ht[i]$ and $Ht[i] < Ht[i+1]$.

To count the number of rounds, we copy the subsequence of local minima into a new array $LMHt[1..m]$, recursively count the number of rounds to clear $LMHt$, and add 1. The recursion bottoms out when $n = 1$.

```
CountRounds(Ht[1..n]):
  if n = 1
      return 0
  else
      m ← 0                    ⟨⟨#local mins found so far⟩⟩
      Ht[0] ← ∞                ⟨⟨sentinel values⟩⟩
      Ht[n+1] ← ∞
      for i ← 1 to n
          if Ht[i−1] > Ht[i] and Ht[i] < Ht[i+1]
              m ← m + 1
              LMHt[m] ← Ht[i]
      return 1 + CountRounds(LMHt[1..m])
```

Our solution to part (b) implies that there are *at most* $\lceil n/2 \rceil$ local minima; thus, $m \le \lceil n/2 \rceil$ when we recursively call CountRounds. Thus, the running time of CountRounds obeys the recurrence $T(n) \le T(\lceil n/2 \rceil) + O(n)$, and therefore $T(n) = O(n)$ by the usual recursion-tree method. ■

Solution: We compute the survivors of the first round in $O(n)$ time using our fast solution to part (a), copy those survivors into a new array $Ht'[1..m]$, recursively count the number of rounds to clear Ht', and add 1. The recursion bottoms out when $n = 1$.

```
CountRounds(Ht[1..n]):
  if n = 1
      return 0
  else
      ⟨⟨identify first-round survivors⟩⟩
      L, R ← WhoTargetsWhom(Ht)
      for i ← 1 to n
          Survives[i] ← True
      for i ← 1 to n
          if L[i] ≠ None then Survives[L[i]] ← False
          if R[i] ≠ None then Survives[R[i]] ← False
      ⟨⟨copy survivors into new array and recurse⟩⟩
      m ← 0                    ⟨⟨#survivors found so far⟩⟩
      for i ← 1 to n
          if Survives[i] = True
              m ← m + 1
              Ht'[m] ← Ht[i]
      return 1 + CountRounds(Ht'[1..m])
```

Our solution to part (b) implies that there are *at most* $\lceil n/2 \rceil$ local minima; thus, $m \le \lceil n/2 \rceil$ when we recursively call CountRounds. Thus, the running time of CountRounds obeys the recurrence $T(n) \le T(\lceil n/2 \rceil) + O(n)$, and therefore $T(n) = O(n)$ by the usual recursion-tree method. ■

Rubric: 3 points = 1 for proving (survivors = local minima) + 1 for algorithm + 1 for analysis. Max 2½ points for $O(n \log n)$ time; max 2 points for $O(n^2)$ time. No credit for an incorrect algorithm that runs in $O(n)$ time. Full credit for the second solution requires a correct $O(n)$-time algorithm for part (a).

General instructions for grading algorithms. Full credit for ***every*** algorithm in the class requires a clear, complete, unambiguous description, at a sufficient level of detail that a strong student in CS 225 could implement it in their favorite programming language, using a software library containing every algorithm and data structure we've seen in CS 125, 225, and previously in 374. In particular:

- Watch for hand-waving, pronouns (especially "this") without clear antecedents, and the **Deadly Sin** "Repeat this process".
- Do not regurgitate algorithms we've already seen.
- Every algorithm must be clearly specified in English, unless the specification is already given *precisely* in the problem statement. In particular, if the algorithm solves a more general problem than requested, the more general problem must be specified explicitly. Similarly, if the algorithm assumes any conditions that are not explicit in the given problem statement, those assumptions must be stated explicitly. **Omitting this description is a Deadly Sin.**
- The meaning of every variable must be either clear from context (like n for input size, or i and j for loop indices) or specified explicitly.