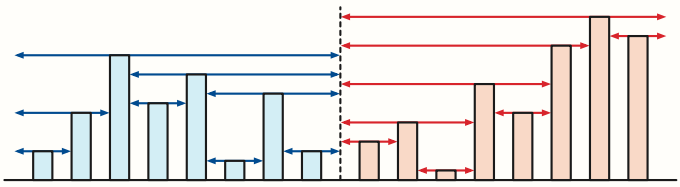
1. *[See the homework sheet for a detailed description of the problem.]*

   (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
      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 $T(n) \le 2T(n/2) + O(n)$, 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.

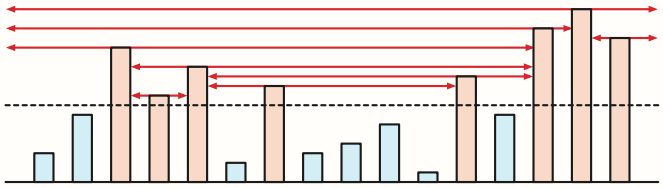
```
WHOTARGETSWHOM(Ht[1..n]):
  if n < 1000
      use brute force
  else
      m ← ⌊n/2⌋
      L[1..m], R[1..m] ← WHOTARGETSWHOM(Ht[1..m])
      L[m..n], R[m..n] ← WHOTARGETSWHOM(Ht[m..n])
      i ← m;  j ← m + 1

      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.

No, nobody would actually do this. ■

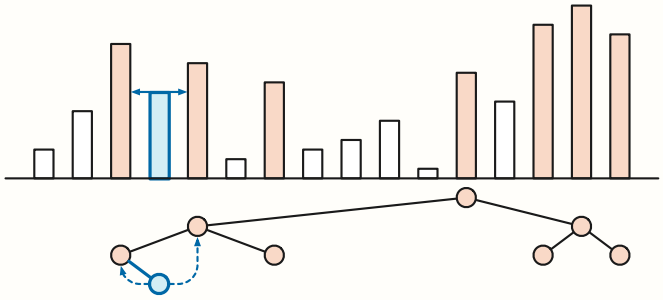
Solution (iterative by position): When all targets are assigned, the left target of any hero with no right target is either NONE or the next taller hero with no

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.

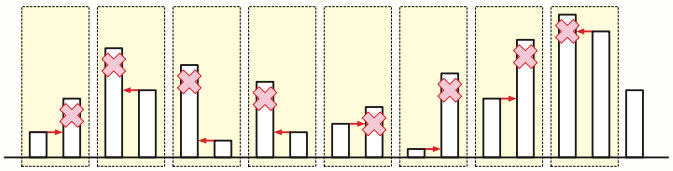


```
WHOTARGETSWHOM(Ht[1..n]):
  for i ← 1 to n
      Pos[i] ← i
  sort Ht downward and permute Pos to match
  ⟨⟨now Pos[i] is the position of the ith tallest hero⟩⟩
  D ← new ordered dictionary
  for j ← 1 to n
      L[Pos[j]] ← PREDECESSOR(D, Pos[j])
      R[Pos[j]] ← SUCCESSOR(D, Pos[j])
      INSERT(D, 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 $O(n^2)$.) ■

(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.



■

(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
      ⟨⟨identity first-round survivors⟩⟩
      L, R ← WhoTargetsWhom(Ht)
      for i ← 1 to n
          Survives[i] ← True
      for i ← 1 to n
          if L[i] ≠ None
              Survives[L[i]] ← False
          if R[i] ≠ None
              Survives[R[i]] ← False
      ⟨⟨copy survivors into new array⟩⟩
      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: 10 points:

- 5 for part (a) = 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.
- 2 for part (b)
- 3 for part (c) = 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. 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 given *precisely* in the problem statement. **Omitting this description is a Deadly Sin.**
- The meaning of every variable must be either clear from context (like n for input size and i and j for loop indices) or specified explicitly.

2. Describe and analyze a recursive algorithm to reconstruct an arbitrary binary tree, given its preorder and inorder node sequences as input. The input to your algorithm is a pair of arrays $Pre[1..n]$ and $In[1..n]$, each containing a permutation of the same set of n distinct symbols. Your algorithm should return an n-node binary tree whose nodes are labeled with those n symbols (or an error code if no binary tree is consistent with the input arrays).

Solution (7½/10): Here is a basic recursive algorithm:

```
TreeFromPreIn(Pre[1..n], In[1..n]):
  if n = 0
      return Null
  ⟨⟨Find inorder root index⟩⟩
  r ← 0
  for i ← 1 to n
      if In[i] = Pre[1]
          r ← i
  if r = 0
      abort immediately
  ⟨⟨Build the tree⟩⟩
  T ← new node
  key(T) ← Pre[1] ⟨⟨= In[r]⟩⟩
  left(T) ← BuildTree(Pre[2..r],     In[1..r−1])
  right(T) ← BuildTree(Pre[r+1..n], In[r+1..n])
  return T
```

The running time of this algorithm obeys the quicksort recurrence

$$T(n) \le O(n) + \max_{1 \le r \le n} \{T(r-1) + T(n-r)\},$$

so the algorithm runs in $\boldsymbol{O(n^2)}$ ***time*** in the worst case. ■

Solution: We speed up the previous algorithm by precomputing an array $PreToIn[1..n]$ such that $In[PreToIn[i]] = Pre[i]$ for all i. We can implement this preprocessing phase either via sorting copies of the input arrays or by using a dictionary data structure.

```
Preprocess(Pre[1..n], In[1..n]):
  for i ← 1 to n
      PreIndex[i] ← i
      InIndex[i] ← i
  sort Pre and permute PreIndex to match
  sort In and permute InIndex to match
  for i ← 1 to n
      PreToIn[PreIndex[i]] ← InIndex[i]
  return PreToIn[1..n]
```

```
Preprocess(Pre[1..n], In[1..n]):
  D ← new dictionary
  for i ← 1 to n
      Insert(D, i, In[i])
  for i ← 1 to n
      PreToIn[i] ← Lookup(D, Pre[i])
  return PreToIn[1..n]
```

The running time of this phase depends on our precise implementation. The sorting-based algorithm runs in $O(n \log n)$ time. The dictionary-based algorithm runs in

$O(n \log n)$ time if we implement the dictionary using a balanced binary search tree, or in $O(n)$ expected time if we implement the dictionary using a hash table.

We compute the actual binary tree in the main recursive algorithm. To simplify the presentation, I'll assume the arrays *Pre*, *In*, and *PreToIn* are global variables. (Sorry.)

```
⟨⟨ Construct the unique binary tree with                    ⟩⟩
⟨⟨    preorder sequence Pre[pstart..pstart+len-1]           ⟩⟩
⟨⟨    and inorder sequence In[istart..istart+len-1]         ⟩⟩
BUILDTREE(pstart, istart, len):
  if len < 0
      abort immediately
  if len = 0
      return NULL
  T ← new node
  key(T) ← Pre[pstart]
  iroot ← PreToIn[pstart]          ⟨⟨index of root in In[]⟩⟩
  leftlen ← iroot - istart         ⟨⟨size of left subtree⟩⟩
  rightlen ← len - leftlen - 1     ⟨⟨size of right subtree⟩⟩
  left(T) ← BUILDTREE(pstart + 1, istart, leftlen)
  right(T) ← BUILDTREE(pstart + leftlen + 1, iroot + 1, rightlen)
  return T
```

This algorithm performs $O(1)$ non-recursive work and builds one node in the output tree. Thus, the total running time including all recursive calls is $O(n)$. The recursion tree for this algorithm has *exactly* the same structure as the output tree!

Finally, here is the top-level algorithm:

```
TREEFROMPREIN(Pre[1..n], In[1..n]):
  PreToIn ← PREPROCESS(Pre, In)
  return BUILDTREE(1, 1, n)
```

The overall algorithm runs in either ***O(n log n) time*** or ***O(n) expected time***, depending on whether or not we use hashing. ■

Solution (iterative tree-traversal): But we don't actually need a dictionary, or sorting, or even recursion, if we're willing to use an explicit stack. The following iterative algorithm performs depth-first traversal of the tree, building the nodes as it discovers them. Each node in the tree is visited twice: once before its children, and again after its left subtree.

The algorithm maintains a stack S of all tree nodes that have been visited only once; these are the ancestors of the current node that appear *after* the current node in the inorder traversal, with deeper such nodes closer to the top of the stack. The algorithm also maintains indices p and i into the *Pre* and *In* arrays, which respectively point to the first node in pre-order that has not been visited and the next node in in-order that has been visited at most once. The basic logic is as follows:

- If the node at the top of the stack matches $In[i]$, visit that node for the second time—pop that node off the stack (as the new current node) and increment i. The next node we visit for the first time will be a right child.
- Otherwise, visit a child of the current node. Create the new node, make it the new current node, set its key to $Pre[p]$, push it onto the stack, and increment p. Until we visit some node for the second time, every node we visit for the first time will be a left child.

```
TreeFromPreIn(Pre[1..n], In[1..n]):
  S ← new stack of nodes
  root ← new node                          ⟨⟨root node of output tree⟩⟩
  cur ← root                               ⟨⟨current node⟩⟩
  p ← 1                                    ⟨⟨current index in preorder array⟩⟩
  i ← 1                                    ⟨⟨current index in inorder array⟩⟩

  cur.key ← Pre[1]                         ⟨⟨visit the root for the first time⟩⟩
  Push(S, cur)
  nextR ← False                            ⟨⟨Is the next node a right child?⟩⟩
  p ← p + 1

  while i < n                              ⟨⟨loop until tree finished and stack empty⟩⟩
      if IsEmpty(S) or Top(S).key ≠ In[i]
          if nextR = True                  ⟨⟨visit a right child for the first time⟩⟩
              cur.right ← new node
              cur ← cur.right
          else                             ⟨⟨visit a left child for the first time⟩⟩
              cur.left ← new node
              cur ← cur.left
          cur.key ← Pre[p]
          Push(S, cur)
          nextR ← False
          p ← p + 1
      else                                 ⟨⟨visit a node for the second time⟩⟩
          cur ← Pop(S)
          nextR ← True
          i ← i + 1
  return root
```

The main loop iterates exactly $2n-1$ times—once for the root, and twice for each node below the root—and each iteration requires $O(1)$ time. Thus, the entire algorithm runs in ***O(n) time.*** ■

Solution (clever): But why should we maintain our own stack when the Recursion Fairy will do it for us?! The following algorithm implements our standard recursive strategy, in $O(n)$ time, with no additional data structures. The key insight is that we don't need to compute the subtree sizes *before* we recursively build the subtrees; instead, we can compute subtree sizes on the fly, as a *side effect* of the recursive subtree construction!

Our recursive subroutine BuildTree$(p, i, stop)$ constructs the unique binary tree with preorder sequence $Pre[p..p+m-1]$ and inorder sequence $In[i..i+m-1]$,

where m is the unique integer such that $In[i+m] = stop$. (The symbol *stop* is the label of the lowest right ancestor of the subtree we are building.) The integer m is also the number of nodes in the output tree; let me emphasize that m is *not* known in advance. The algorithm returns both the tree and its size m.

```
BUILDTREE(p, i, stop):
  if In[i] = stop
      return (NULL, 0)
  else
      root ← new node
      (root.left, ℓ)  ← BUILDTREE(p + 1, i, Pre[p])
      root.key        ← Pre[p]    ⟨⟨= In[i + ℓ]⟩⟩
      (root.right, r) ← BUILDTREE(p + ℓ + 1, i + ℓ + 1, stop)
      return (root, ℓ + r + 1)
```

Each recursive call to this function performs $O(1)$ non-recursive work and either allocates one of the $m = \ell + r + 1$ nodes of the tree or visits one of the $m + 1$ *Null* pointers (to missing children). It follows that this algorithm runs in $O(m)$ time.

To reconstruct the entire tree, we add a special symbol \$ to the end of the inorder array, which we then use as the stop character. Intuitively, we are pretending that the tree we want is the left subtree of a node with label \$.

```
TREEFROMPREIN(Pre[1..n], In[1..n]):
  In[n + 1] ← $
  (root, m) ← BUILDTREE(1, 1, $)      ⟨⟨m = n⟩⟩
  return root
```

Because the reconstructed tree has n nodes, our overall algorithm runs in ***O(n) time***, as claimed. ■

Rubric: 10 points =

- \+ 2½ for speedup to $O(n \log n)$ =
  - \+ 2 for actual preprocessing algorithm
  - \+ ½ for preprocessing time analysis
    - ○ No penalty for "$O(n)$ time" instead of "$O(n)$ expected time" for the hashing-based preprocessing algorithm, even though "$O(n)$ time" is technically incorrect.
    - ○ The final $O(n)$-time algorithm gets full credit for this part.
  - – No credit for this part if the overall algorithm runs in $O(n^2)$ time.
- \+ 7½ for main recursive algorithm =
  - \+ 2 for English specification (if different from problem statement)
    - ∗ **Deadly Sin:** No credit for the problem if this description is missing (unless identical to the problem statement)
  - \+ ½ for top-level function call (if different from problem statement)
  - \+ ½ for base case
  - \+ ½ for root

- + 1 for left subtree
- + 1 for right subtree
- + 2 for time analysis
- ◦ No penalty for omitting error-checking

- For the stack-based iterative algorithm: 5 for the algorithm itself + 5 for explanation/justification. (I've seen many versions of the iterative algorithm on the web, but most of them are just raw code, with no explanation of what the algorithm is actually doing. I've also seen several bits of code that claim run in $O(n)$ time, but in fact run in $O(n^2)$ time in the worst case.)

3. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets:

   - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
   - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

   For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

   Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

   [Hint: Use or modify the linear-time selection algorithm described in class on Thursday.]

Solution: Our recursive algorithm actually solves the more general *weighted selection* problem: Given the two arrays $S[1..n]$ and $W[1..n]$ and a target weight *targetW*, this algorithm finds the largest element $x \in S$ such that $W(S_{<x}) < targetW$.

```
WeightedSelect(S[1..n], W[1..n], targetW):
  if n < 1000
      use brute force
  else
      m ← ⌊n/2⌋
      p ← Select(S[1..n], m)              ⟨⟨Find index of unweighted median⟩⟩
      Partition*(S[1..n], W[1..n], p)     ⟨⟨Partition both arrays⟩⟩
      leftW ← 0                           ⟨⟨Compute total weight left of median⟩⟩
      for i ← 1 to m − 1
          leftW ← leftW + W[i]
      ⟨⟨Binary search! Wait, sorry, I meant... One-armed quicksort!⟩⟩
      if leftW > targetW
          return WeightedSelect(S[1..m], W[1..m], targetW)
      else
          return WeightedSelect(S[m..n], W[m..n], targetW − leftw)
```

Here we use a modified version of the Partition algorithm from quicksort. Whenever Partition* swaps two elements of the value array $S[1..n]$, it also swaps the corresponding elements of the weight array $W[1..n]$; however, decisions about what to swap are based entirely on comparisons between elements of the value array. (In hindsight, maybe I should have specified the input as a single array of value/weight pairs.)

Because unweighted selection and partitioning each equire $O(n)$ time, the running time obeys the recurrence $T(n) = O(n) + T(n/2)$. The usual recursion-tree stuff now implies that the algorithm runs in $\boldsymbol{O(n)}$ ***time***.

Finally, to compute the weighted median, we invoke the weighted selection algorithm as follows:

```
WeightedMedian(S[1..n], W[1..n]):
  totalW ← 0
  for i ← 1 to n
      totalW ← totalW + W[i]
  return WeightedSelect(S, W, totalW/2)
```

In practice, we can speed up this algorithm by selecting an *approximate* median in the first step, for example using the median-of-medians technique from the unweighted selection algorithm, or even choosing the pivot element at random. However, these improvements only change the constant factor in the $O(n)$ running time. ■

Rubric: 10 points =

- \+ 2 for English specification of weighted selection.
  - – **Deadly Sin:** No credit for the problem if this is missing.
  - – This is not the workable specification. For example, we could also pass two weights lw and rw, and select any element x such that $W(S_{<x}) < lw$ and $W(S_{>x}) < rw$.
- \+ 1 for base case
- \+ 1 for partitioning by approximate median
  - — In particular, you need to explain how to partition *both* arrays.
- \+ 3 for recursive calls
  - — ½ for each off-by-one or boundary error
  - – No penalty for omitting early return case
- \+ 1 for top-level function call
- \+ 2 for time analysis