1. (a) According to @512, "divide and conquer" is just a suggestion, so we provide a faster algorithm that runs in $O(n)$ time.

   We will describe an algorithm ComputeLeft, which computes the resulting array L for Ht. Then, we can observe that $R = \text{ComputeLeft}(\text{Reverse}(Ht[1..n]))$. This is because right targets are left targets if we reverse the order of the heroes. Thus, the algorithm for WhoTargetsWhom can be given as follows (we abbreviate $Ht = H$).

```
1: procedure WhoTargetsWhom(H[1..n])
2:     H' ← Reverse(H)
3:     L ← ComputeLeft(H)
4:     R ← ComputeLeft(H')
5:     return L, R
6: end procedure
7: procedure ComputeLeft(H[1..n])
8:     L[1..n] is an array of n elements, all of which are None
9:     S is an empty stack
10:    for all i ∈ {1, ..., n} do
11:        while ¬IsEmpty(S) ∧ H[Peek(S)] < H[i] do
12:            Pop(S)
13:        end while
14:        if ¬IsEmpty(S) then
15:            L[i] ← Peek(S)
16:        end if
17:        Push(S, i)
18:    end for
19: end procedure
```

   The running time of ComputeLeft is $O(n)$, because every element of H is pushed and popped once (the while loop performs n operations throughout the entirety of ComputeLeft) and the loop performs n iterations. Reverse is also $O(n)$, so it is easy to see that WhoTargetsWhom is also $O(n)$.

   We now argue about the correctness of ComputeLeft. For each i, S will contain only contain indices $j < i$ such that $H[j] > H[i]$. In particular, each index $j \in S$ and every index $j' < j$ in S will also satisfy $H[j'] > H[j]$. If there are elements to the left of j smaller than $H[j]$, those indices are removed from the stack on iteration j of the outer loop, because even if those elements may be larger than $H[i]$, j is closer to i, so we will pick j. If S is empty, then $H[i]$ is larger than every element to i's left, so it targets nobody. Otherwise, we target the correct hero. In some sense, S is a stack containing all possible "future targets", and we always keep this stack up to date and correct.

   (b) We split the heroes into $\left\lfloor \frac{n}{2} \right\rfloor$ adjacent pairs from left to right. If n is odd, this leaves the rightmost hero without a pair, but this does not matter. Between each pair, if they have heights h_1, h_2, either $h_1 < h_2$ or $h_2 < h_1$ so one of them must be the target of the other. Since we have $\left\lfloor \frac{n}{2} \right\rfloor$ pairs, we clearly see that *at least* $\left\lfloor \frac{n}{2} \right\rfloor$ heroes are targetted.

(c) We make the following observation. A hero at position k is not killed if and only if the heroes at positions $k-1$ and $k+1$ (if they exist) are taller than they are. Otherwise, one of the adjacent heroes would kill him. Therefore, we can find all the heroes that live through each round and run the simulation again. Here, we make the abbreviation $Ht[1..n] = H[1..n]$.

```
1: procedure ITERATIONS(H[1..n])
2:     if n = 1 then
3:         return 0
4:     else
5:         Z[1..n] is an array of length n initialized to all null values
6:         n' ← 1
7:         for all k ∈ {1, ..., n} do
8:             if NOTKILLED(H[1..n], k) then
9:                 Z[n'] ← k
10:                n' ← n' + 1
11:            end if
12:        end for
13:        return 1 + ITERATIONS(Z[1..n'])
14:    end if
15: end procedure
```

NOTKILLED is the $O(1)$ algorithm that checks if hero k is not killed. Recall that $n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil$. Also recall that the complement of Z in H has at least $\left\lfloor \frac{n}{2} \right\rfloor$ elements, so Z has *at most* $\left\lceil \frac{n}{2} \right\rceil$ elements. Thus the running time for this algorithm satisfies

$$T_n \leq n + T_{\lceil n/2 \rceil} \qquad T_1 = 1.$$

This recurrence relation is known to be $O(n)$.

2. We make the following observation. A pre-order traversal performs operations as root, left, right, while an in-order traversal performs operations as left, root, right. Therefore, we can recover the root by examining the pre-order traversal and recover the traversals of the left and right subtrees by examining the in-order traversal.

   The tree reconstruction algorithm can be given as follows, with $Pre[1..n] = P[1..n]$ and $In[1..n] = I[1..n]$.

```
1: procedure ReconstructTree(I[1..n], P[1..n])
2:     if n = 0 then
3:         return EMPTY
4:     end if
5:     r ← P[1]
6:     k ← Find(I[1..n], r)
7:     if ¬SetEquals(I[1..k − 1], P[2..k]) then
8:         error
9:     else if ¬SetEquals(I[k + 1..n], P[k + 1..n]) then
10:        error
11:    end if
12:    L ← ReconstructTree(I[1..k − 1], P[2..k])
13:    R ← ReconstructTree(I[k + 1..n], P[k + 1..n])
14:    return r, L, R
15: end procedure
```

   The algorithm functions as follows.

   (i) Check to see if the inputs are empty traversals, in which case we return an empty binary tree.

   (ii) The root of the tree is the first element of the pre-order traversal.

   (iii) We find the splitting index k in I, which takes $O(n)$ time. Everything to the left of k is in the left subtree and everything to the right of k is in the right subtree.

   (iv) Check to see that the left and right subtrees (predicted by number of elements) of P agree, by ensuring that they contain exactly the same elements. This also takes $O(n)$ time.

   (v) Recursively reconstruct the left and right subtrees L and R.

   (vi) Finally, return the resulting binary tree as its root and its left and right subtrees.

   The running time for this algorithm satisfies $T_n = 3n + T_{k-1} + T_{n-k}$ and $T_0 = 1$. In the worst case this algorithm performs at $O(n^2)$ (in every instance, we get a subtree of size 0 and subtree of size $n - 1$).

3. We are not asked to show that the weighted median exists. Therefore, we will just assert that such an element exists and we will find one such element.

   Let $w = \sum_i W[i] = 1$. Replace $W[i]$ with $\frac{W[i]}{w}$ so that $\sum_i W[i] = 1$. Notice that this does not change the result of the problem.

```
1:  procedure WMSelect(S[1..n], W[1..n])
2:      if n ≤ 2 then
3:          return a brute forced value
4:      else
5:          m ← QuickSelect(S, ⌊n/2⌋)
6:          r ← Partition2(S, W, m)
7:          W_1 ← Σ_{i=1}^{r-1} W[i]                (W_1 = w(S_{<x}))
8:          W_2 ← Σ_{i=r+1}^{n} W[i]                (W_2 = w(S_{>x}))
9:          if W_1 ≤ 1/2 and W_2 ≤ 1/2 then
10:             return S[r]
11:         else if W_1 > 1/2 then
12:             W[r] ← W[r] + W_2
13:             return WMSelect(S[1..r], W[1..r])
14:         else
15:             W[r] ← W[r] + W_1
16:             return WMSelect(S[r..n], W[r..n])
17:         end if
18:     end if
19: end procedure
```

   Partition2 is the partition function from quicksort that partitions S, but moves the elements of W in the same way as their corresponding elements in S. This takes $O(n)$ time. Selection of the median of S also takes $O(n)$ time. Computation of the sum of the weights also takes $O(n)$ time.

   This algorithm then satisfies $T_n = T_{\lceil n/2 \rceil} + O(n)$, which is known to be $T_n = O(n)$.

   This algorithm functions as follows. We first examine the median as a pivot. If this median is the weighted median, we immediately return. Otherwise, we know that the weighted median is the "heavier" set, so we set our pivot to have the entire weight of the lighter set plus its own weight (notice that this doesn't change the sum of the weights in the subproblems, so the resulting answer is correct!), and recurse into a size $\left\lceil \frac{n}{2} \right\rceil$ subproblem.