

- (a) Towers 2, 5, 6, and 8 have an uninterrupted view to the right.

```
(b) max_east_heights = [0] * n
    i = n
    max = 0

    while i >= 2
        if A[i] > max then
            max = A[i]

        max_east_heights[i - 1] = max
        i = i - 1
```

```
(c) view_to_left = []
    view_to_right = []
    max = 0

    for i = 1 to n
        if A[i] > max then
            max = A[i]
            view_to_left.push(i)

    max = 0
    i = n
    while i >= 1
        if A[i] > max then
            max = A[i]
            view_to_right.push(i)

    i = i - 1
```

To find which towers have an uninterrupted view to the left, we scan the heights from left to right, keeping track of the maximum height so far. Any tower which is taller than the current maximum has an uninterrupted view, so we record its index and update the maximum. We do the same in reverse to find the towers with a view to the right.

- (d) [4, 5, 6, 3, 3, 1, 4, 1, 1, 1]

- (e) The optimal time cost to compute  $D(i)$  is  $O(n)$ , since we only need to scan the towers array from  $i + 1$  to  $n$  once and count how many towers are strictly shorter than the one we care about.

```
(f) d = [1] * n

    for i = 1 to n
        if A[i] > A[i + 1] then
            k = i
```

```

break

j = n - 1
i = k

while i >= 0
    while A[j] >= A[i] and j > k
        j -= 1

    d[i] = 1 + j - k
    i -= 1

```

Assuming all but one tower is lower-value tells that the towers form a sawtooth pattern with a single drop somewhere in the middle. Call the tallest tower before the drop index  $k$  and split the list into left and right parts.

We scan the left part from right to left and keep track of an index  $j$ , which also moves from right to left. This tracks how many towers in the right half are shorter than the current tower. We only have to look for shorter towers in the right half because the left half is increasing before the drop.

(g)  $d = [1] * n$

```

function populate_d(l, r)
    if l == r then
        return

    m = (l + r) / 2
    populate_d(l, m)
    populate_d(m + 1, r)

    right_half = copy_slice(A, m + 1, r + 1)
    sort(right_half)

    for i = l to m + 1
        shorter = binary_search(right_half, A[i])
        d[i] = d[i] + shorter

populate_d(1, n)

```

Recursively, we split the list into 2 parts at each step. When we merge, we sort the right half to find the index of the first shorter tower if it would exist (in Python this is `bisect.bisect_left`). This index gives us the count of shorter towers.