/** Note for the grader:
 * Important details are highlighted in blue for your convenience :)
 */

Question1:

1. **Example:**

   Array A[5,7,4,2] n=4

   Before calling straighten(1,4) we have a “partially” sorted array [5,7,2,4]. We assume that A[1,2] and A[3,4] are correctly sorted after the recursion, and all we need to do is to correctly straighten A[1,4]

   Calling Straighten(A[5,7,2,4]), A[2] and A[3] will need to be swapped.

   we know that A[2] is the local maximum of [5,7] and A[3] is the local minimum of [2,4], we need to put the local max backward and local min forward, so we can have [local min1, local min2, local max1, local max2] such that [5,2,7,4] , and we can perform further straighten operations. If line 6-8 are deleted, then we are unable to put local min and max of the sub-array to the proper position of the big array, and this would lead to a final result of [5,2,7,4] which is a false result.

2. **Example:**

   The example [5,7,4,2] that I used in the previous question also works when line 10&11 are swapped. After the weirdSort recursion we got [5,2,7,4], and if line 10&11 are swapped, that is we first sort A[1] and A[2], then A[2] and A[3], and then A[3] and A[4]. The final result we got is [2,7,4,5] which is a false result.

3. **Proof By Induction**

   a) Base case:

   1) If there’s only one element in array A, there’s no need to sort it.
   2) If there are two elements in the array A, we call weirdSort(A[1]), weirdSort(A[2]), and straighten(A[1,2]). Since A[1] is just one element, calling weirdSort(A[1]) produces no effect, and so does calling weirdSort(A[2]). Calling straighten(A[1,2]) would compare the value of A[1] and A[2] and swap them if they are not in the correct order. Therefore, the weirdSort algorithm sorts array correctly for n=2.

   b) Induction hypothesis:

   Assume that the weirdSort and Straighten algorithm holds for any sorted or unsorted array of size n= 2^k.

c) Induction step:

We need to prove that the weirdSort algorithm also works for array size n = 2^k+1.

1) Straighten() takes in A[1,2^1], A[1,2^2], A[1,2^3] and so on as parameter. And we made an assumption in the induction hypothesis that straighten method works for array[1,2^k], and now we need to prove that straighten also works for array[1,2^k+1].
2) Straighten() method will take in an array of size = 2^k+1 and code from line 6 to 12 will execute since 2^k+1 > 2. And then we will perform swapping of the elements of A[i+n/4] and A[i+n/2].
3) The three Straighten() recursion from line 9-11 each takes in an array of size n =2^k and according to the induction hypothesis, straighten will sort the array A[1, 2^k] correctly. Therefore, Straighten also sorts array size N = 2^k+1 correctly as well.

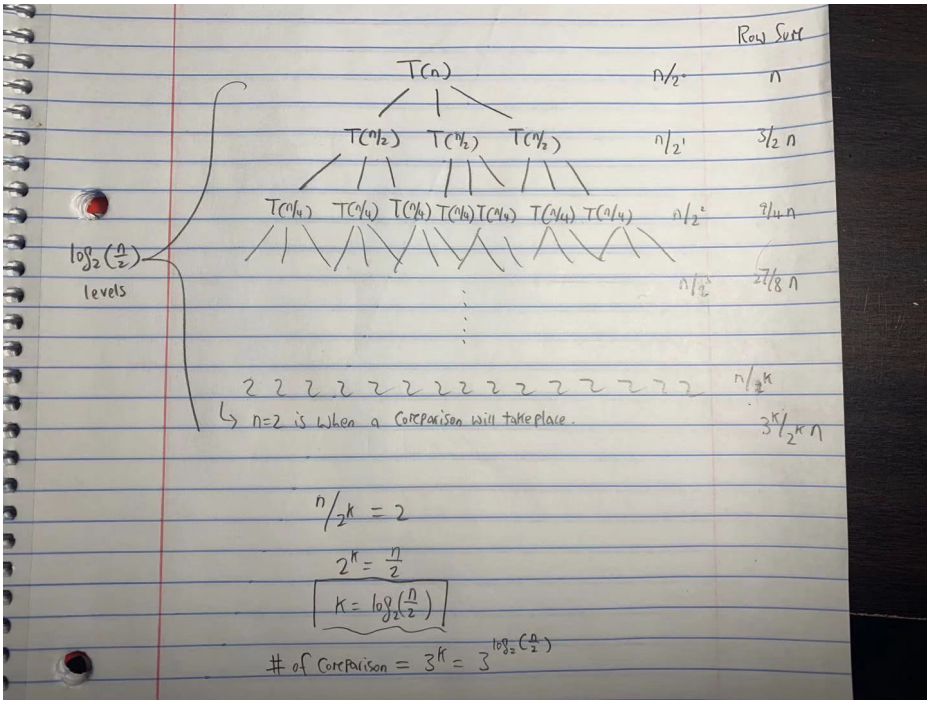
d) Conclusion

Therefore, according to the math induction, weirdsort and straighten sort array of size power of two correctly.

4. Running Time analysis

a) Straighten() analysis:

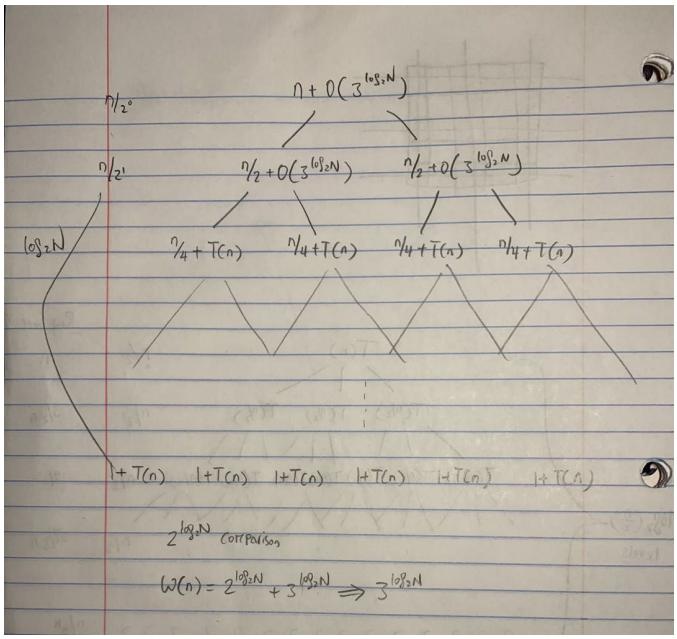
1) If n=2, straighten will only make one comparison, so T(n) = 1.
2) If n>2, straighten will perform 3 Straighten() recursions, and each has a problem size = n/2. And T(n) can be represented as T(n) = 3T(n/2). This can be solved using a recursion tree.



We will have log2 (n/2) levels of recursion until we get to the base case (n=2). When reached the base case, there will be 3^k comparisons where k is levels of recursion tree. So we can write T(n) as O(3^log(N/2)) which can be further generalized to be O(3^log(N)).

b) WeirdSort analysis:

WeirdSort recursively calls itself with size = n/2 twice every time, and it will call straighten once, so we can write it as W(n) = 2W(n/2) + T(n) where T(n) is the running time of straighten(). And we get W(n) = 2W(n/2) + O(3^log2(N)). And W(n/2) is solved to be 2^log2(N) using the recursion tree method. Because the term 3^log2(N) dominates the term 2^log2(N). Therefore, W(n) can be expressed as W(n) = O(3^log2(N)).



Question2:

A brief description of the algorithm: The idea for finding a local max in a matrix (2D array) came from finding a local max in a 1D array.

1) Using the Divide & Conquer algorithm, so we can divide the matrix into quadrants (sub-problems)
2) Look at the central column and row, and try to find a local max. The reason for doing this is that we will only need to compare upper&lower neighbor or left&right neighbor.
3) Once we find a local max in the central column and row, we compare its value with its neighbors. If it is the biggest, we return it as local max, else, we go to its largest neighbor and from that row and column we continue recursing.

Pseudocode:

```
// this is the recursion method that will divide the problem into smaller sub-problems
FindLocalMax ( row, col, 2Darray):
        centralRow = row/2
        centralCol = col/2
        compare (centralRow, centralCol, 2Darray);
        // now we have a local max, so we will compare it to its neighbors

        If (2Darray[centralRow][centralCol] is the largest element in the “neighborhood”):
                Return 2Darray[centralRow][centralCol];

        Else if ( 2Darray[centralRow+1][centralCol] > 2Darray[centralRow][centralCol]):
                // recurse to the upper left quadrant
                FindLocalMax ( centralRow+1, centralCol, 2Darray)
        Else if (2Darray[centralRow-1][centralCol] > 2Darray[centralRow][centralCol]):
                // recurse to the lower left quadrant
                FindLocalMax ( centralRow-1, centralCol, 2Darray)
        Else if(2Darray[centralRow][centralCol+1] > 2Darray[centralRow][centralCol]):
                // recurse to the upper right quadrant
                FindLocalMax ( centralRow, centralCol+1, 2Darray)
        Else if(2Darray[centralRow][centralCol-1] > 2Darray[centralRow][centralCol]);
                // recurse to the lower right quadrant
                FindLocalMax ( centralRow, centralCol-1, 2Darray)
        Else :
                // the current element is the largest
                Return 2Darray[centralRow][centralCol];
```

// this method will compare if there exists any element that is larger than the current element
compare (centralRow, centralCol, 2Darray):

If (some element > 2Darray[centralRow][centralCol]) :
Record the position of that element and update centralRow and centralCol

Else:
Return 2Darray[centralRow][centralCol]

Prove the correctness:

a) Base case:

If the input matrix is 1*1, there is only one element in the matrix, so it is the local/global maximum, so FindLocalMax() correctly finds the local max.

b) Induction hypothesis:

The FindLocalMax() algorithm will correctly find a local max for 2^k * 2^k matrix

c) Induction steps:

To prove that FinaLocalMax() algorithm also correctly finds a local max for 2^k+1 * 2^k+1 matrix. The FinaLocalMax() method divides the input matrix n*n into four n/2 * n/2 matrix, and then try to find a local max between centralRow and centralCol. The sub-problem (sub-matrix) we got as a result is of size 2^k * 2^k. According to the induction hypothesis, FindLocalMax() algorithm will correctly find a local max for 2^k * 2^k matrix, thus, FindLocalMax() algorithm will also correctly find a local max for 2^k+1 * 2^k+1 matrix.

Running Time analysis:

// T(n/2) is due to each recursion dividing the width and height of the matrix by 2
// and 2n is due to finding the local max between the centralCol and centralRow

$T(n) = T(n/2) + 2n$

-> we can solve the recursion by unrolling

$T(n) = T(n/2) + 2n$

$T(n) = T(n/4) + 2n + n$

$T(n) = T(n/8) + 2n + n + n/2$

$T(n) = T(n/16) + 2n + n + n/2 + n/4$

…

$T(n) = T(1) + 2n + n + n/2 + n/4 + \ldots\ldots$ -> converge to a constant

$T(n) => O(n)$