Please read the instructions carefully:

1. There are two stages to each assignment:

   (a) For the first stage, you need to submit your homework solution as a single pdf file to NYU Classes. It can be either a scanned copy of your written solution or photos joined into a pdf. You are also allowed to typeset the solutions in MS Word/LaTeX. In case of scanned copies it is your responsibility that the solution is readable. There are no late submissions accepted, as sample solution outlines (to be used for the second stage of the homework) will be given out on the day of the submission of the first stage of the homework. You should leave around 5 to 6 cm of space between each part of the answers for the second stage of the homework.

   (b) For the second stage of the assignment, you are required to critique your own solutions and analyze the mistakes that you made. Based on the outlines provided, you need to write the self criticism in the empty space that you left in your homework. Do not rewrite your solution, you need to understand and point out the mistakes that you made when writing the original solutions. A good analysis of your solution should summarize the key steps, ideas or techniques that you previously missed. It should prompt you to reorganize your thoughts and think about what you can do to improve the solutions. Even in the case where you completely missed the solution it is insightful to point out why the approach adopted by you does not work and identify the step where it fails. This will allow you to get full credit on the self evaluation part of the homeworks and of course help you in understanding the question.

   Mark correct solutions in green, incorrect solutions in red and if you are unsure about any solutions write it in blue.

2. For each question where you are required to design an algorithm, you need to clearly explain the algorithm and prove the running time complexity as asked in the question.

3. You can assume $O(1)$ complexity of arithmetic of numbers (e.g. multiplication/division of numbers and calculations of $n!$) unless otherwise stated in the question.
Question 1. (10 points) Recursion tree analysis

1. \( T(n) = O(n^{0.51}) \)
2. \( T(n) = O(n^{\log_6 7}) \)
3. \( T(n) = O(n \log^2 n) \)
4. \( T(n) = O(n) \)

Question 2. (10 points) Alternating norms

First observe that even if we had all the input vectors in the first quadrant on the \( x - y \) plane, the output of the answer would still be the same as we can still get the same 4 reflections from all these vectors.

Now observe that 
\[
\|v_i^x + v_j^y\| = d(v_i^x, v_j^{(y+2)\%4}), \text{ for any } i, j.
\]
Hence we can change the problem from minimizing the norm of the sum to minimizing the distance between some reflections of these vectors.

Now observe that given two vectors, \( v_i, v_j \), if we want to get the reflections that minimize the distance, we should bring the reflections in the same quadrant. Hence, we can first reflect all vectors into the first quadrant and just use the closest pair of points algorithm to get the minimum distance. To get back the original answer just reflect one of the vectors to the third quadrant and get back the answer.

Question 3. (25 points) Min-maxing utilities

For each index \( i \), we make a point \( p_i = (i, \sum_{j=1}^{i} a_i) \).

Now notice that \( u(l, r) = d(p_l, p_r) \).

1. Find the two closest pair of points and return their indices.

2. Notice that the two farthest pair of points will need to be on the convex hull.
   Hence first find the convex hull of the point that we constructed.
   Now after we get the convex hull, we need to find the farthest point on the hull for every point also on the hull.
   Naive searching will make this \( O(n^2) \), but we can do better by using the rotating calipers algorithm.
   https://en.wikipedia.org/wiki/Rotating_calipers This gives us the diameter of the hull in \( O(n) \).

Question 4. (15 points) Drawing a tree

First choose the leftmost point, \( l \), as the root of the tree.

Now sort all other points, \( p_i \), based on the angle that the line \( \overline{lp} \) makes with the x-axis.

Choose the first half of these points as the left subtree and the second half as the right subtree.

As we are sorting based on the angle, we are guaranteed that the subtrees will not intersect.

To analyze the time complexity, we have to sort at each level and solve two subproblems of \( n/2 \) as well.

\[
T(n) = 2T(n/2) + O(n \log n)
\]

which we showed in question 1 is \( O(n \log^2 n) \).
Question 5. (25 points) Dynamic inversions

1. At each node on these levels, the two numbers get swapped.

2. At each level, these two numbers get swapped.

3. Do the merge sort algorithm and create the inversion tree.
   For each level store the number of cross inversions and cross sorted pairs, by summing up the numbers at each node on this level.
   When we get a flip($l$) operation we only need to flip the numbers at the levels where the array sizes are less than or equal to $2^l$.
   To give back the number of inversions, we just need to add up the cross inversions at each level.
   As each update operation only requires us to flip at most $O(\log n)$ numbers, we can do the updates in $O(\log n)$ and re calculate the total inversions by summing up the $O(\log n)$ numbers. Hence each query can be answered in $O(\log n)$ to give a total running time of $O(m \log n)$.

Question 6. (30 points - Extra credit) Dominating uniqueness

1. We can store all numbers in an AVL tree and return the position of each number to uniquely map it to an element in $\{1, \ldots, n\}$. Construction of the tree takes $O(n \log n)$ and to answer each query takes $O(\log n)$, so we can remap $a_i$ to $b_i$ in $O(n \log n)$.
   This has the same dominating pairs because if $a_i \leq a_j$, we have that $b_i \leq b_j$.

2. We show how to make the $lreg$ array.
   We store a count array, which we will populate by traveling from left to right in the array $[a_1, \ldots, a_n]$. Here count[i] will store the number of times we have encountered $i$ so far. Every time we see a new element $a_j$, we increase the count of $a_j$. And to make lreg[i], when we have traveled to index $i$, we set lreg[i] = count[a]

3. We will modify the merge sort algorithm so that we are sorting the two lists lreg and rreg at the same time. Because we have that the number of dominating pairs is the number of cross inversions between the two lists, we will use divide and conquer to calculate this number.
   Let's say we have broken the two arrays into 4 arrays lreg, lregr, rreg, rregr. We calculate the number of cross inversions between lreg, rregl and lregr, rregr recursively. But we might still have some inversions between lreg, rreg. We now do a third merge step to calculate the inversions between the two arrays lreg, rreg (but we don't actually merge them).
   Now merge lreg ↔ lreg and rreg ↔ rreg.
   This needs us to do 3 merge steps at each level instead of 2 but it is still $O(n)$, which gives us the same complexity as merge sort.