Emre Oytun
2018040040

1. Algorithm:

findMinDistanceBetweenDrones(coordinates):

→ Firstly, it sorts the given coordinates by x and y so that there is no need to sort in each recursive call. — O(nlogn)

→ Then, it calls "findMinDistanceRec" method by passing sorted x and y list and lower index = 0, higher index = len-1. (last element's index)

findMinDistanceRec(sortedXList, sortedYList, li, hi):

→ Calculates n = hi - li + 1

→ Base Case: If n ≤ 3, then find the closest pair by brute force. — O(1)

→ Calculates mid = (li + hi) // 2 — O(1)

→ Constructs sortedYListLeft and sortedYListRight arrays as: — O(n)

- Create a set with elements sortedXList[li : mid] — O(1)
- Then iterate through sortedYList and put the elements into left or right array. — O(n)

→ Makes two recursive calls and saves the results. — 2T(n/2)

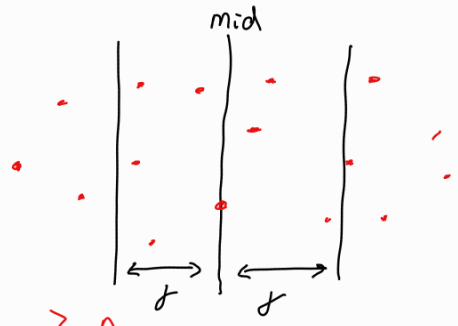
→ Delta = min(deltaLeft, deltaRight) — O(1)
Gets min of the results as delta to construct a strip of drones with x indexes [mid.x - delta, mid.x + delta]

→ Iterates through sortedYList at hand, and constructs the strip by comparing the absolute distances of points' x and mid x — O(n)

→ Finally, it iterates through the points in the strip and compares with at most 15 elements in each iteration to find the min distance since it is proven that at most 15 points/drones comparing is enough. — $O(n)$

Analysis:

Strip:



→ For rec. method:

Let assume $n = 2^k$ $(k \geq 0)$:

$T_1(n) = 2T_1(n/2) + O(n)$

By Master's Theorem: $\log_b a = \log_2 2 = 1 > n$

$f(n) = O(n)$ } $\Theta(n \log n)$

By interpolation: $T_1(n)$ is non-decreasing ✓
$n \log n$ " " ✓
$n \log n$ is Θ-invariant ✓
} $T_1(n) \in \Theta(n \log n)$ holds for all n.

→ Outer method:

$T_2(n) = \underbrace{T_1(n)}_{O(n \log n)} + n \log n \in O(n \log n)$ //

---

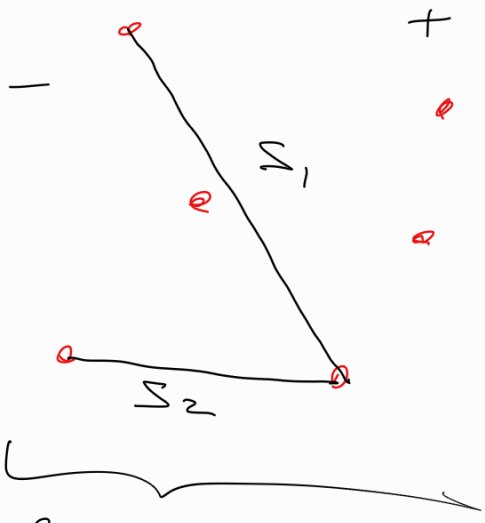
2. This question is similar to convex hull problem as described in the lecture because the purpose is to find minimum number of sensors to enclose the area.

Pseudocode:

Sort the points by X coordinate (once)
For input list L:
Base Case: If $n \leq 5$, find the points on convex segments with brute-force. — $O(1)$ explained below
Compute convex points for A and B recursively — $2T(n/2)$

Merge the results — $O(n)$

findConvexPointsExhaustive(points):

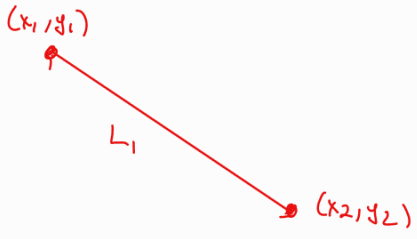


S_1 is not a segment.

S_2 is a segment.

i) Try all pair of points and check if the line is a segment of convex.

ii) Find the coefficients of line equation as:



L_1 line equation:

$$a_1 x + b_1 y + c = 0$$

$$a_1 = y_2 - y_1$$

$$b_1 = x_1 - x_2$$

$$c_1 = y_1 x_2 - x_1 y_2$$

iii) Then put all remaining points into the line eq. and check the sign.

iv) If all points at one side of the line then it is the segment of convex.

v) After finding the points on segments, find the center point by iterating through the points by adding x & y coordinates and scaling the points by multiplying each one with size.

vi) Then sort the points around the center-point in counter-clockwise order.

- Comparing points:
  - → Subtract center point from points.
  - → Find the cross product sign. (point1 on point2)
  - → If cross product sign is negative then line from origin to point1 is in the left of line from origin to point2. Return -1.
  - → Otherwise, return 1.

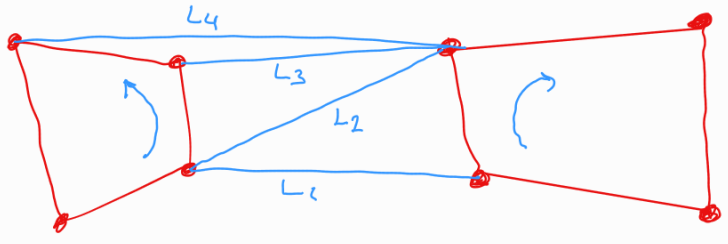
vii) Normalize the resulting points by dividing them

$$T_1(n) = n.(n-2) + (n-1)(n-2) \cdots + 1(n-2)$$
$$= (n-2)\frac{n(n+1)}{2} \in O(n^3) \text{ for loop}$$

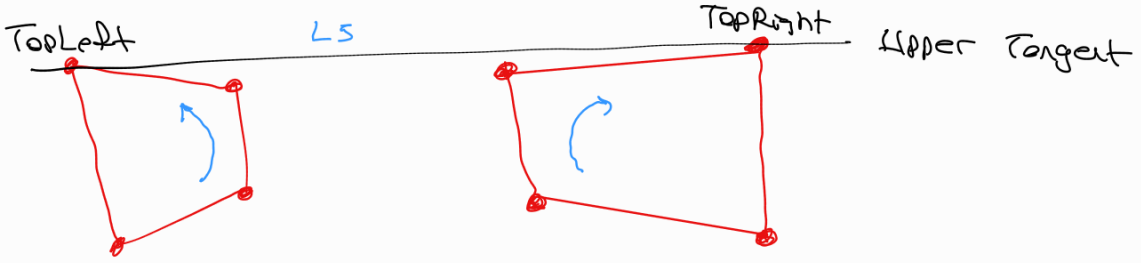
$$T(n) = T_1(n) + \underbrace{O(n \log n)}_{\text{for sorting}} \in O(n^3)$$

But since the n is max 5, brute-force base case complexity is O(1).

findUpperTangent:



!! Last Step



findUpperTangent (rightMostLeft, leftMostRight, leftPoints, rightPoints):

```
indexLeft = rightMostLeft
indexRight = leftMostRight

done = false
while (not done):
    done = True
    while the next clockwise point on
          right convex is above the line (indexLeft - indexRight):
        indexRight = next clockwise point on right
    end            convex
    while the next anti-clockwise point on
          left convex is above the line (indexLeft - indexRight):
        indexLeft = next anti-clockwise point on left
                    convex
        done = False
    end
end
return indexLeft, indexRight
```

Analysis: At most, it goes through whole points.
— O(n+m)

findLowerTangent: This is the same algorithm as above but this time left point goes clockwise and right points goes anti-clockwise. Also, the next points are checked if they're under the line.
— O(n+m)

mergeTwoConvex(leftPoints, rightPoints): — O(n+m)

→ Find the rightmost point in leftPoints — O(n)
→ " " leftmost " " rightPoints — O(m)
→ Find upper tangent — O(n+m)
→ " lower " — O(n+m)
→ Find the convex points like: — O(n+m)
- Start from topLeft and go to bottomLeft — O(n)
- " " bottomRight " " " topRight — O(m)

→ Return the result

Total Complexity :

Let assume $n = 2^k$ $(k \geq 0)$ format :

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

By Master's Theorem: $\log_b a = \log_2 2 = 1 > n$

So, $T(n) \in \Theta(n \log n)$

By Interpolation: $T(n)$ is non-decreasing ✓

$n \log n$ " " ✓

$n \log n$ " Θ - invariant ✓

So. $T(n) \in \Theta(n \log n)$ holds for all n.

$T(n) \in O(n \log n)$ //

(3.) Finding minimum number of operations to transform one string to another requires finding the longest common subsequence of these strings.

findMinOperationsTransformString (Str1, Str2) :

commonChars = None

Find the characters in longest common subsequence and fill the commonChars array — $O(n * m)$ explained below

Then create two stack consisting of these characters. — $O(1)$

For each ch in Str1 :

if (ch != stack1.peek()): } $O(n)$

Mark ch for delete

else :

stack1.pop()

For each ch in Str2 :

if (ch != stack2.peek()) : } $O(m)$

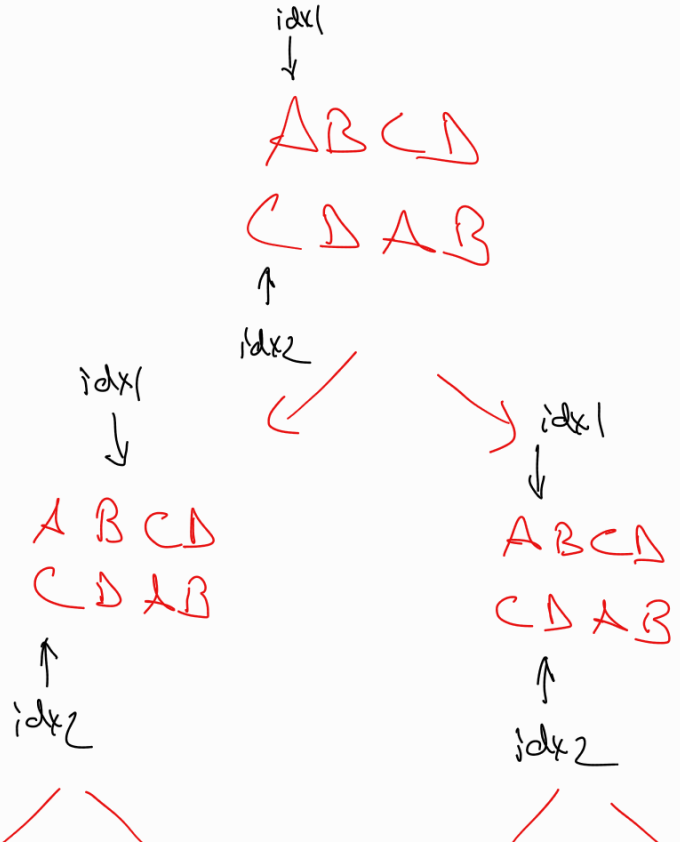
Mark ch for insertion

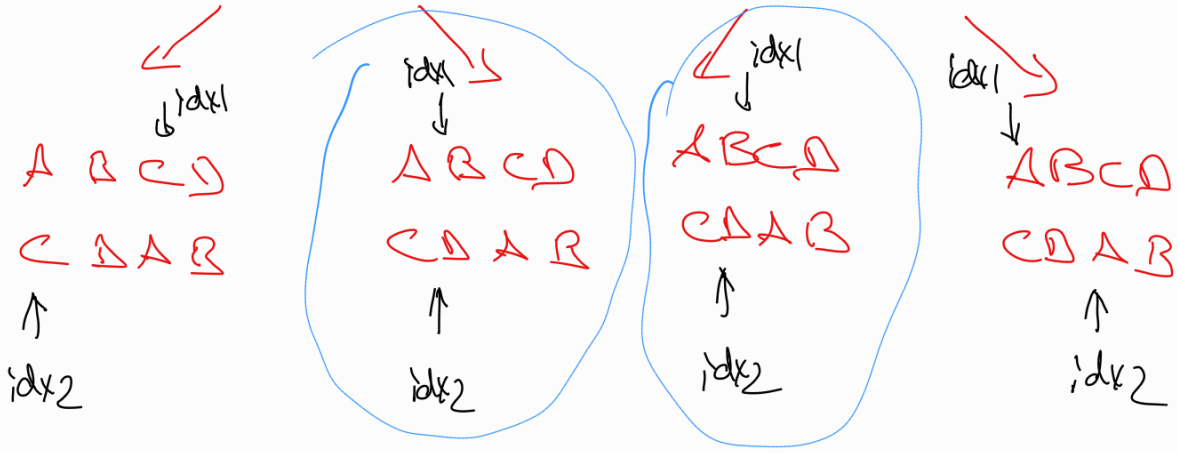
```
        else:
            stack2.pop()

    return numOperations, result
end
```

```
findCommonSubsequence(str1, str2, idx1, idx2):
    if (idx1 ≥ len(str1)) or (idx2 ≥ len(str2)):
        return [], 0
    if (str1[idx1] == str2[idx2]):
        return 1 + findCommonSubsequence(str1, str2, idx1+1, idx2+1)
    res1 = findCommonSubsequence(str1, str2, idx1+1, idx2)
    res2 = findCommonSubsequence(str1, str2, idx1, idx2+1)
    return max(res1, res2)
```

This recursive way takes $O(nm)$ time to find the longest common subsequence.

If we examine the recursive calls,





As we see, the subproblems overlap so the problem has optimal substructure.

We can build a DP Table as:

	0	1	...	n+1
0	0	0	...	0
1	0			
⋮	⋮			
n+1	0			

→ max. subsequence char. number

For DP[i][j] (i >= 1, j >= 1):

```
if (str1[i-1] == str2[j-1]):
    DP[i][j] = DP[i-1][j-1] + 1
else:
    DP[i][j] = Max(DP[i-1][j], DP[i][j-1])
```

* So, filling DP takes O(n * m)
* After we filled the table the algorithm starts

from i = len(str1), j = len(str2) and goes back to find characters in the subsequence according to the DP table filling algorithm. This takes O(min(n,m)).

* Therefore:

Filling DP Table + Finding characters $\in O(n*m)$

$O(n*m)$ $O(\min(n,m))$

* The overall algorithm is:

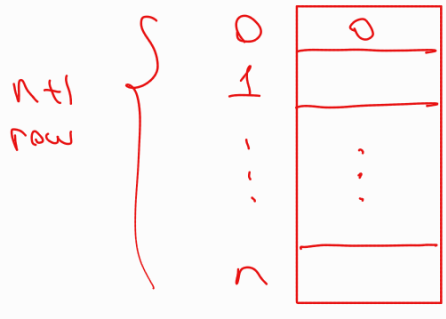
$T(n) = O(n*m) + O(1) + O(n) + O(m)$

$T(n) \in O(n*m)$ //

(4.) We can think the recursive way first:

```
Procedure maxDiscount(stores[0...n-1], i):
    if(i == n):
        return 0

    res = maxDiscount(stores, i+1)
    return max(res, res + calc_disc(stores[i]))
end
```

=> Since the subproblems overlap, we can use DP technique to efficiently solve this problem:



DP[i] = max(DP[i-1], DP[i-1] + calc-discount(stores[i-1]))

Analysis:

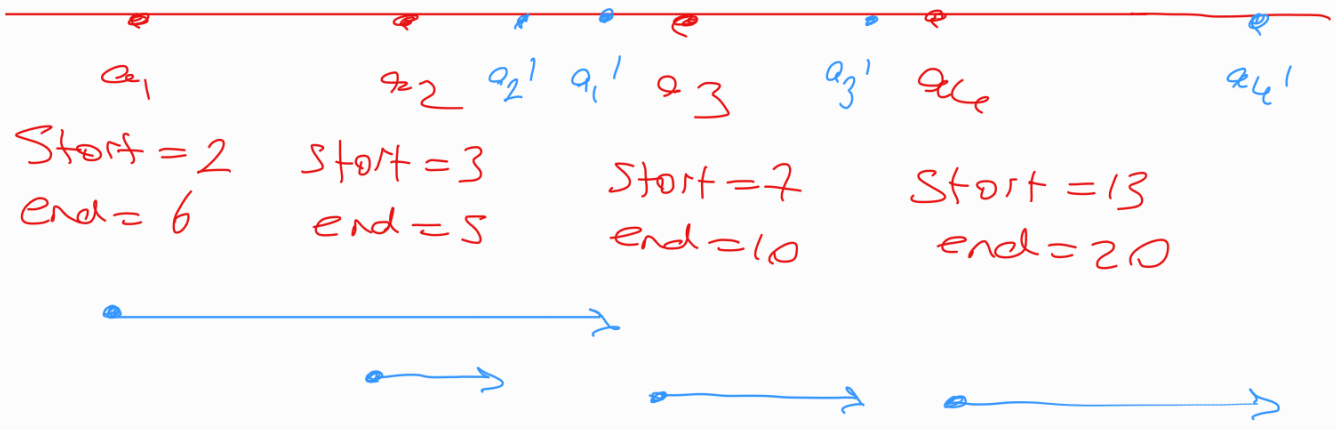
⇒ The algorithm iterates through the rows of DP table from 1 to n.

⇒ Each iteration takes constant time.

Therefore, time complexity is:

$T(n) = n \in O(n)$ //

(5.)



Max Antenna Activated (antennas):

```
if there is no antenna:
    return 0

Sort antennas according to end point
curAvailablePoint = 0
totalAnt = 0
for each antenna in sorted antennas:
    if (antenna.startPoint >= curAvailablePoint):
        totalAnt += 1
        curAvailablePoint = antenna.endPoint + 1
end for
end totalAnt
```

* It's a greedy algorithm such that it sorts the antennas by end points and select as much as antenna possible.

* Sorting antennas + Looping over sorted antennas

  $O(n \log n)$ $O(n)$

* Therefore, complexity is $O(n \log n)$.
