Prob. 1	Prob. 2	Prob. 3

Problem 1.

Let's begin with the main intuition of the algorithm before expliciting it. For each edge of our polygon, the inside of our simple polygon lies only on one side of this edge. If there exists a point within the polygon from which it is possible to draw a segment to every vertex of the polygon without crossing the boundary of the polygon, then this point must be on the inside-side of each edge. Let's formulate this as a property and then prove it.

Proposition
There exists a point within a simple polygon from which it is possible to draw a segment to every vertex of the polygon without crossing the boundary of the polygon **iff** the intersection of the half-planes defined as the inside-side of each and every edge of the polygon is non-empty.

Let's prove it. First, let's assume there exists such a point P. The line between this point and a given vertex is in the inside of the polygon, then this point is the inside-side of the two edges associated to this vertex. Therefore, this point P has to be on the inside-side of every edge of the polygon.

Reciprocally, let's assume the intersection of the half-planes defined as the inside-side of each and every edge of the polygon is non-empty. Let's consider a point P in this intersection. We need to show that the whole line between this point and any vertex is fully inside the polygon. Let's consider our line crosses an edge. As our line is going to be inside at its beginning (near P) and at its end (it reaches a vertex by the inside-side of the two edges defining this vertex), it needs to cross at least another edge to re-enter the polygon. This means that our line goes from the outside to the inside of this edge. Therefore P is on the outside-side of this edge which is impossible. Thus, the whole line between P and any vertex is fully inside the polygon.

Now that we have found a necessary and sufficient condition for our problem, we can describe an algorithm to solve it using the randomized incremental construction algorithm we saw in class to solve 2D linear optimisation problems. Our algorithm is the following :

- We first need to find where is the interior of the polygon given the list of the edges. This is easily done in linear time, for instance by suming the angles knowing that the sum of the interior angles is less than the sum of the exterior ones.
- Then we transform our problem into an optimization one, considering that beeing on the interior-side of edge i is giving us an inequality of the form $a_{i,1}x + a_{i,2}y \leqslant b_i$. Now we choose a function to optimize, we can take whatever we want so let's pick $f(x,y) = x$. This transformation can be done in time linear to the number of edges.

- Finally, we solve this problem with the 2D linear optimisation algorithm seen in class. This algorithm runs in expected linear time. The existence of a solution to the optimisation problem is equivalent to the existence of a point within the polygon from which it is possible to draw a segment to every vertex of the polygon without crossing the boundary of the polygon.

The three steps of our algorithm run in linear time or expected linear time. Therefore our algorithm runs in expected linear time.

Problem 2.

Just like for the question 1, we will transform this problem into a 2D linear optimization problem, so that we can use the (expected) linear runing-time 2D linear optimization algorithm already studied during the course.

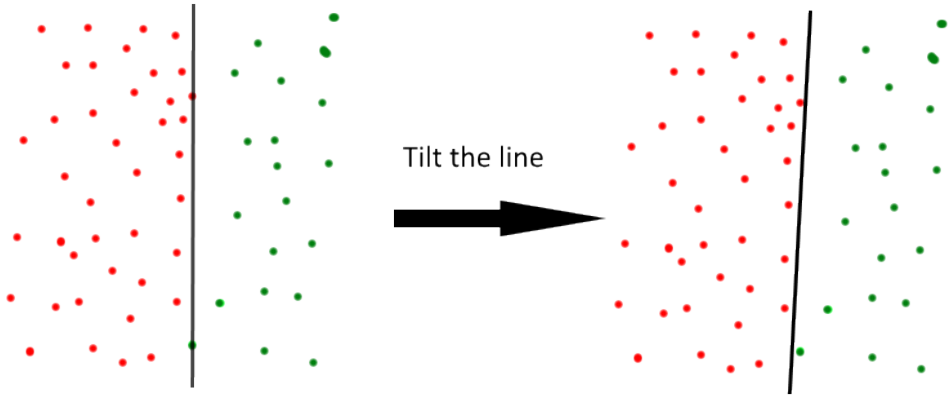
Let suppose that we have n red points, of coordinates $(rx_i, ry_i) \forall i \in 1,..,n$ and m green points of coordinates $(gx_i, gy_i) \forall i \in 1,..,m$. Our goal is to find a line separating the green points from the red points (we suppose that our computer is not affected by daltonism).

Generally speaking, the cartesian equation of a line is of the form $n_x x + n_y y = d$. It is very much like the equation of a plane in 3D: here (n_x, n_y) is the "normal" of the line, and d is the distance of the line to the origin.

However, the general equation of a line in 2D uses three parameters. To be able to use the 2D linear optimization algorithm, we would like to have only two parameters. In fact, we can use a parametric representation of a line with only two parameters to represent the lines that are not vertical (the lines for which $n_y \neq 0$). Therefore, a non-vertical line can be represented by an equation of the form $y = n_1 x + n_2$.

Is it legit to restrict the line that we will find to a non-vertical line? Yes it is, because if a vertical line separates the red points from the green points, we can find a non-vertical line that will fit too: as we have a finite number of distinct points, we know that there is a guaranteed distance ε from one point to another one. Moreover we know that three points are never aligned, therefore at most two points are on a line, the others are at least at a distance μ to the line: we can "tilt" the vertical line so that we have a non-vertical line separating the red points from the green points (as show the figure 1).

Figure 1: Tilting of a vertical line to a non-vertical one.



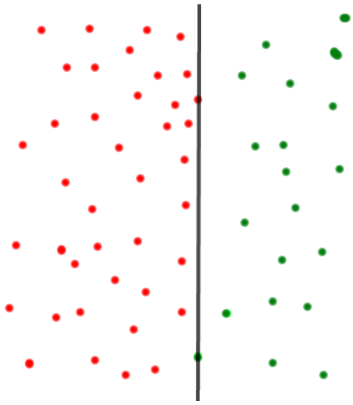
Because of that, we will be searching a line represented by the equation $y = n_1 x + n_2$ of parameters (n_1, n_2).

A point is said "under" a line of parameters (n_1, n_2) if its coordinates (x, y) verify $y \leq n_1 x + n_2$, while it is considered "above" the line if we have $y \geq n_1 x + n_2$.

This can be rewritten as $(-x)n_1+(-1)n_2 \leq -y$ for "under", and $xn_1+n_2 \leq y$ for "above". Those are constraints for a 2D linear optimization problem!

Thus, we would like to specify our constraints like this: $(-rx_i)n_1+(-1)n_2 \leq -ry_i \forall i \in 1,..,n$ for red points ("under") and $gx_in_1+n_2 \leq gy_i \forall i \in 1,..,m$ for green points ("above"). However, there are cases where we can not find a line such as the red poins are "under" and the green points "above", for example in the case presented by the following figure.

Figure 2: In this case, red points can not be under the line.



In these cases we must run our optimization algorithm another time to check if we can find a line which separates the points in the other way, with the red points "above" and the green points "under" (reversing the constraints).

As the function to optimize must be linear and there is no line better than another one, we will optimize the function $f(n_1,n_2)=0$.

Therefore, we have reformulated the problem in terms of a 2D optimization problem, that we may try to solve two times (one time where we are searching for a line where the red points are "under"/green points "above", the other time where we search for a line where the red points "above"/green points "under").

Thanks to this reformulation of the problem we can claim that we have an $O(n+m)$ expected running time algorithm solving that problem: we use the 2D linear optimization algorithm as seen during the class where we incrementally find a solution satisfying more and more constraints, at least one time, at most two times.

Problem 3.

We consider without loss of generality that our array contains the n numbers from 1 to n. Let X_{ij} be the random variable that takes value 1 if we compared i and j during our quick sort and 0 if not. Two elements can't be compared more than once because, when there is a comparison that means one of them has been chosen as the pivot and will be at his right place for the remaining operations of the sort algorithm. Since we do $O(1)$ operation after each comparison, the expected complexity will be : $O(E[\sum_{i<j} X_{ij}])$

We compare i and j ($j > i$) if and only if no element between them is taken as pivot before them. Since we choose our pivots at random, we get $P(X_{ij} = 1) = 2/(j-i+1)$. Then we can compute :

$$\begin{aligned} E[\textstyle\sum_{i<j} X_{ij}] &= \textstyle\sum_{i<j} E[X_{ij}] \\ E[\textstyle\sum_{i<j} X_{ij}] &= 2 \times \textstyle\sum_{i<j} 1/(j-i+1) \\ E[\textstyle\sum_{i<j} X_{ij}] &= 2 \times \textstyle\sum_{i=1}^{n} \sum_{k=1}^{n-i+1} 1/k \\ E[\textstyle\sum_{i<j} X_{ij}] &= 2 \times \textstyle\sum_{i=1}^{n} \sum_{k=1}^{i} 1/k \end{aligned}$$

However $\sum_{k=1}^{i} 1/k = O(ln(i))$, so we can conclude that the overall average complexity of quicksort is $O(n \times ln(n))$.