Problem 1

Let H be a set of $n > 3$ half-planes with a non-empty intersection such that none of their bounding lines are parallel. We call a half-plane $h \in H$ redundant if it does not contribute an edge to the common intersection of half-planes in H. Give an $O(n \log n)$-time algorithm for finding all the redundant half-planes in H.

We can modify the divide and conquer algorithm to keep track of the non-contributing half planes. The key idea is that if a half plane does not contribute at a given step, then it does not contribute overall. This is because the intersection after any given step is convex and can only get smaller with each step. We can then keep a running list of non contributing half planes $\overline{C}$ alongside C itself. We can then modify the merge step to add any half planes that arent used to $\overline{C}$. The algorithm then is

---

Algorithm 1 NonContributingHalfPlanes(H)

---

1. Perform the same divide and conquer algorithm for half plane intersection with the following changes
   - The list of non contributing half planes $\overline{C}$ is passed to every call so that it can be accessed/modified at any step
   - When merging, add any non contributing half plane to the running list $\overline{C}$

For time complexity, the merge step remains the same as it is linear with respect to the input size of half planes. Furthermore each non-contributing half plane is only processed once due to the merge step only handling intersecting halfplanes for the actual convex region output. Therefore the original divide and conquer algorithm's complexity of $O(n \log n)$ is maintained.

Problem 2

What is the worst-case running time of ParanoidMaximum? What is the expected running time (with respect to the random choice in line 3)?

For worst-case complexity, consider the case when the candidate x is always the maximum of the input set. Then when the check $x \leq x'$ occurs, it will always go to the else branch where it checks against the entire input set. We can then set up the recurrence relation

$$T(1) = C$$
$$T(n) = T(n-1) + (n-1)O(1)$$

which when iterated gives

$$\begin{aligned} T(n) &= T(n-1) + (n-1)O(1) \\ &= T(n-2) + [(n-1) + (n-2)]O(1) \\ &= T(n-3) + [(n-1) + (n-2) + (n-3)]O(1) \\ &\vdots \\ &= T(1) + [(n-1) + (n-2) + (n-3) + 2]O(1) \\ &= C + O(1) \sum_{i=2}^{n-1} i \end{aligned}$$

which trivially is $O(n^2)$.

Problem 3

A simple polygon $\mathcal{P}$ is called star-shaped if it contains a point q such that for any point p in $\mathcal{P}$ the line segment $\overline{pq}$ is contained in $\mathcal{P}$. Give an algorithm whose expected running time is linear to decide whether a simple polygon is star-shaped.

The key thing to note is that if such a point exists, then it must be able to see every edge of the polygon. If one imagined each edge as half planes directed towards the inside of the polygon, then the point must lie in each of those half planes which gurantees that it can see every edge. But this is simply the half plane intersection problem. If the intersection of all the half planes isn't empty, then such a point exists and if the intersection is empty then one does not. The algorithm then is

Algorithm 2 IsStarPolygon

1. Construct a half plane for every edge of $\mathcal{P}$ directed to the interior of the polygon
2. Use the iterative half plane intersection algorithm with any reasonable target function to get the intersection of all the half planes
3. Check if the linear program was feasible or unfeasible. If feasible, then return true, else return false

For time complexity, the iterative half plane algorithm has an expected run time of $O(n)$. As for creating the half planes, it is possible to do that in $O(n)$ time (if $\mathcal{P}$ is a DCEL it is constant time to get the inner twin edge and if $\mathcal{P}$ is a list in a rotational ordering then inwards is always left/right). Therefore in total the expected running time is just $O(n)$.

Problem 4

On n parallel railway tracks n trains are going with constant speeds $v_1, v_2, \ldots, v_n$. At time $t = 0$ the trains are at positions $k_1, k_2, \ldots, k_n$. Give an $O(n \log n)$ algorithm that detects all trains that at some moment in time are leading.

For each train we can construct a linear equation representing its position of the form $x_i(t) = v_i t + k_i$. Finding which train is in the lead at any given time t is the same as finding the upper envelope of all the $x_i(t)$ lines. Since each $x_i(t)$ is a line, we can define half planes for each by taking the plane pointing in the positive direction of train position (in the context of plotting position against time). By using the half plane intersection algorithm, we can get the convex region whose boundary is the upper envelope we want to find. Therefore the algorithm is

---

Algorithm 3 TrainLeader($\{v_1, \ldots, v_n\}, \{k_1, \ldots, k_n\}$)

---

1. Construct the set of half-planes $H = \{y \geq v_i t + k_i : 1 \leq i \leq n\}$
2. Get the convex region C from IntersectHalfPlanes(H)
3. For each segment in C, associate the time interval it represents to the corresponding half-plane/train
4. Return these interval-train associations

As for time complexity, constructing the half planes takes $O(n)$ time, intersecting the half planes takes $O(n \log n)$ time, and producing the final interval-train associations takes $O(n)$ time. Therefore in total the algorithm has $O(n + n \log n + n) = O(n \log n)$ time complexity.

Problem 5

Repeat the previous problem, but now suppose that the y-coordinate of train i at time t is determined by the equation $y = a_i t^3 + b_i t^2 + c_i t + k_i$, where a_i, b_i, and c_i are constants, and you want to compute, as efficiently as possible, the set of all trains that at some point in time are leading. You may assume that you can compute all the (at most 3) intersections of two such curves in constant time. What is the running time of your algorithm?

The position of each train gives a cubic equation of the form $x_i(t) = a_i t^3 + b_i t^2 + c_i t + k_i$. We are then interested in the function

$$\text{Leader}(t) = \underset{1 \leq i \leq n}{\text{argmax}}\, x_i(t).$$

Since each cubic equation is x-monotone, we can solve this by finding the upper envelope of all the $x_i(t)$. If A, B, C, K are the lists of the coeffiecients for each cubic, the algorithm is simply

---

Algorithm 4 TrainLeader(A, B, C, K)

---

1. Construct $T = \{-x_i(t) : 1 \leq i \leq n\}$ with the form of x_i outlined above
2. Find the lower envelope Leader(t) of T using the plane sweep algorithm from the slides
3. Return $-$Leader(t)

The algorithm negates each cubic so that the lower envelope corresponds to finding the negative of the upper envelope. This resulting lower envelope once negated will then be the upper envelope of the original set of cubics. Analyzing time complexity

- Constructing all the negated cubics takes $O(n)$ time
- Each cubic can only intersect another at most 3 times. Therefore the running time of the lower envelope algorithm will be

$$O(\lambda_3(n) \log n) = O(n \log^* n \log n).$$

- Negation takes $O(n)$ time.

Therefore in total the algorithm is $O(n + n \log^* n \log n + n) = O(n \log^* n \log n)$ time complexity.