1. (3 points) The algorithm given in Chapter 1 of the book to compute the convex hull of a set of points computes the upper convex hull and the lower convex hull separately. You could argue that this is not such an elegant solution. In the exercise we consider a variation where the whole convex hull is computed in the same style, but in once. The idea is to first determine the bottommost point $p$ of $S$. Then we sort all other points of $S$ around $p$ by angle. Let this sorted order be $p_1, \ldots, p_{n-1}$, in clockwise order seen from $p$. In other words, the vector $\vec{pp}_1$ makes the largest angle with the vector $(1,0)^T$, among all vectors $\vec{pp}_1, \vec{pp}_2, \ldots, \vec{pp}_{n-1}$.

Now we put $p_1$ and $p_2$ in a list, and use a for-loop very similar to the one on page 6 of the book, lines 3–6.

(a.) Which degenerate cases do you want to ignore at first?

*Most had this correct:* more than one bottommost point, more than one point with the same angle with respect to the bottommost point, and more than two collinear points (on the convex hull).

(b.) Give the pseudo-code for this algorithm, ignoring those degenerate cases.

*Some were sloppy with the notation.* If you start with a set $p_1, \ldots, p_n$, you have $n$ points in total. If you take out a bottommost point, the other $n-1$ points remain and will be sorted by angle around this bottommost point.

(c.) Argue that your algorithm returns the convex hull correctly.

*Since a proof was not required but an argumentation, it was quickly considered correct.* This question was more meant to reflect on why your answer would be correct (or not). This is something you should always do when you develop an algorithm (or any type of solution, for that matter).

(d.) Explain how the previously ignored degenerate cases should be handled.

*The most common mistake was with the second degenerate case,* when more than one point has the same angle with respect to the bottommost point. If this angle is not the largest or smallest one occurring, then the order does not matter. But you have to be very careful to get it right when the largest or smallest angle occurs for more than one point.

For example, suppose you took the leftmost of the bottommost points, and you sorted points with the same angle from far to near (or by $x$- or $y$-coordinate). Then it might happen that two points make the largest angle, and if the furthest one is treated first, then it will be deleted (incorrectly) in the next step, because $p, p_1, p_2$ does not make a right turn(!). You can easily check that the near-to-far order also works for multiple points with the same lowest y-coordinate (minimum angle), although here any actually order works. But you have to be careful how and when to handle $p$ itself, possibly to remove the last added point. If you use the near-to-far order, it works automatically correctly.

*Besides sorting from near to far w.r.t. the bottommost point $p$, you could also simply eliminate all points with the same angle except the furthest one.*
2. (3 points) A triangulation of a point set \( S \) is a maximal planar subdivision that has a vertex for every point in \( S \), and no other vertices. Since a triangulation is a planar subdivision, it can be stored in a doubly-connected edge list (DCEL). You may assume that not all points of \( S \) lie on a single line (which would be a highly degenerate case), and no two points of \( S \) coincide.

Write pseudo-code, using the default pointer names like \( \text{IncidentFace} \), etc., to perform the operation \( \text{EdgeSwap}(\vec{e}) \). This operation takes a half-edge \( \vec{e} \) of the DCEL as its argument. It tests if the edge \( e \) that \( \vec{e} \) (partially) represents can be removed from the subdivision and replaced by a different edge, restoring the property that the subdivision is a triangulation. If the test succeeds, the edge swap is also executed (in the DCEL of course).

You need not write code for a geometric test like: does \( r \) lie to the right of the directed line through \( p \) and \( q \)? This can be stated in normal English text. Make sure that your code is correct no matter what half-edge of the DCEL representation of the triangulation is passed to your code.

There are two cases where a flip is not possible: (i) when the half-edge \( \vec{e} \) or its twin is incident to the unbounded face, or (ii) the quadrilateral formed by the union of the two triangles incident to \( \vec{e} \) and \( \text{Twin}(\vec{e}) \) is not convex (or just convex in a degenerate way). Some tested (i) by analyzing whether the incident faces of \( \vec{e} \) and its twin were both triangles. But the unbounded face can be bounded by a triangle as well (all other points are inside the triangle)!

The most common mistake was forgetting to set some of the fields of the objects. When you draw the picture “before” and “after”, and go over all fields, you can see that your code can be correct only if you set (at least) 6 \( \text{Next} \), 6 \( \text{Prev} \), 2 \( \text{Origin} \), 2 \( \text{IncidentEdge} \), 2 \( \text{OuterComponent} \), and 2 \( \text{IncidentFace} \) fields. If you created new half-edges instead of re-using \( \vec{e} \) and its twin, you have to set 2 \( \text{Twin} \) fields as well, and you need to set two more \( \text{IncidentFace} \) fields.

Some confused \( \text{InnerComponent} \)s and \( \text{OuterComponent} \)s, some thought that the \( \text{Next of a half-edge is a vertex} \), some misused the standard names (Previous instead of \( \text{Prev} \)) and more things of this type went wrong.

3. (4 points) Given a set \( S \) of \( n \) disjoint line segments in the plane and a set \( C \) of \( m \) disjoint circles in the plane. The circles can contain each other, as long as they don’t intersect (meaning: they don’t have any point in common).

We want to report all circles of \( C \) that do not intersect any line segment of \( S \). For this you will develop a plane sweep algorithm that does this efficiently.

When you study this problem, the first thing you must realize is that the output size is always \( O(m) \) in size, also when there are \( \Theta(nm) \) intersections between line segments and circles. So you cannot argue that a running time of \( O((n + m + k)\log(n + m)) \) is good when you have \( k \) intersection points. The challenge is to get to a running time of \( O((n + m)\log(n + m)) \), which is indeed possible. If you had the \( k \) in your running time (or \( I \) for intersections), then your solution is not really efficient (in the worst case) and you could score only 3 out of 4 points.

(a.) Define the status of your plane sweep algorithm.
The status always refers to the situation at the current place of the sweepline. Depending on your approach, a few different answers can be correct. In any case, the sweepline can intersect a circle twice, so there does not seem to be a well-defined order of intersection along the sweep-line. Most took the good solution of splitting a circle into its left half and its right half. Then the status can be the intersected line segments and circle halves ordered from left to right by intersection along the sweep line, where only the circle halves for which no intersection is passed/discovered are still in the status.

Many forgot to mention the order, and some did not split the circles.

(b.) Describe the status structure and the event list.

The status structure is a balanced binary search tree that stores the status in its leaves. The event list stores the events by y-coordinate, and we choose it to be a balanced binary search tree as well.

(c.) Describe how events are handled.

Most identified the upper and lower endpoints of line segments and the upper and lower endpoints of circles (circle halves) as the events. Most also used the intersection points between a line segment and a circle as events; it is possible for a correct solution to have these events but it is not strictly necessary.

The only idea you need in the solution (and therefore in the event handling) is that you do not need a circle anymore as soon as you discover that it intersects a line segment. You can just eliminate it from the status structure altogether. Then you will not spend too much time.

You have to be careful: when you do not use segment-circle intersection events, you will remove the circle when the segment and the circle become horizontally adjacent. But when you remove the circle, you create a new horizontal adjacency which could again be a circle, which you again would remove. So you need a while-loop to code this correctly.

You also have to be careful that a circle is stored with its two halves. If you remove the whole circle when one half has an intersection, you must also pay attention to new horizontal adjacencies when you remove the other half! However, you could also just leave the other half in the status structure. At a bottom-of-circle event, you report the circle only if both halves are still in.

Some algorithms failed when circles were inside other circles. Some tried to adapt the status structure to handle this, but this gives inefficiency. Bottom line: use geometric observations (and verify them) to keep your algorithms and data structures simple.

(d.) Give the whole algorithm (which includes the initialization).

If your event handling was extensive and complete, this was easy.

(e.) State and prove the running time of the algorithm.

Common mistakes were stating $O(\log n)$ time for event handling instead of $O(\log(n + m))$, or not placing the brackets, as in $O(\log n + m)$.

Somebody claimed that the number of intersection points, $k$, could be as large as $n!$ (factorial). Somebody else claimed that the whole algorithm ran in $O(\log n \log m)$ time. And there were a few more gross errors (lack of understanding or background in algorithms analysis) of this type.