1 Sorting Points

There are a couple 'techniques' or, more appropriately, algorithms used to sort points. We can start by sorting them either 'up-down' and then 'left-right' or any other similar combination. Sorting points like this is rather simple, every time we need to compare a pair we first check their y coordinate and if they're equal we check their x coordinate.

```
void sort_points(Point2D *points, size_t length)
{
  for (size_t i = 0; i < length - 1; ++i) {
    for (size_t j = 0; j < length - i - 1; ++j) {
      if (points[j].y < points[j + 1].y) {
        SWAP(Point2D, points[j], points[j + 1]);
      } else if (points[j].y == points[j + 1].y && points[j].x >
          points[j + 1].x) {
        SWAP(Point2D, points[j], points[j + 1]);
      }
    }
  }
}
```

Please excuse the use of 'bubble sort' variation, but it is the fastest thing to implement and show for demo purposes. 'Bubble sort' is also not that bad as it works perfectly for smaller data sets since it practically has no overhead.

This kind of sorting is usually used when we're trying to store sorted data in some data structure or hash-map. Surprisingly a variant of this kind of sorting is one of the steps in a triangulation algorithm described in a paper titled: 'An implementation of Watson's algorithm for computing 2-dimensional Delaunay triangulations' (1984).

First step in the aforementioned algorithm is: 'Sorting points in an ascending order of their x-coordinate'. You can imagine that's relatively simple, all we have to do is change the inner condition to just be:

```
if (points[j].x > points[j + 1].x) {
  SWAP(Point2D, points[j], points[j + 1]);
}
```

2 Sorting Points Clockwise

In my infinite wisdom when researching this topic and trying to figure out something on paper, I have ultimately concluded: 'Wait... this is actually pretty computationally heavy'. Indeed when we're trying to sort points clockwise, sorting them like we did previously would not achieve the desired result. Instead we have to sort them by their angle, meaning we have to use the arctan function.

I'll spare the unnecessary details and just get to the point. All we have to do is calculate the angle of one point and compare it with an angle of another point. There's an interesting case however when both angles are equal, I personally just check their distance from the center/centroid and compare those instead.

```c
void sort_points_clockwise(Point2D *points, size_t length)
{
  Point2D centroid = point_get_centroid(points, length);

  for (size_t i = 0; i < length - 1; ++i) {
    for (size_t j = 0; j < length - i - 1; ++j) {
      float p0_angle = point_get_angle(points[j], centroid);
      float p1_angle = point_get_angle(points[j + 1], centroid);

      if (p0_angle < p1_angle) {
        SWAP(Point2D, points[j], points[j + 1]);
      } else if (p0_angle == p1_angle) {
        float p0_dist = point_get_sqrdistance(points[j],
            centroid);
        float p1_dist = point_get_sqrdistance(points[j + 1],
            centroid);

        if (p0_dist > p1_dist) {
          SWAP(Point2D, points[j], points[j + 1]);
        }
      }
    }
  }
}
```

Of course the case of two points having the same angle will occur when they're both lying on the same x coordinate as our centroid.

There are ways to optimize this, for example, recalculating only when we add a point, or when you add a point you check where it should be place in a sorted array. It all depends on your problem domain, are the points moving? Are they standing still? Will we be adding points? etc. Those are questions that can help you optimize for common cases and work within the problem domain a little better.