Class work: Convex hulls

1. **Cubes and Hypercubes.** A zero-dimensional cube is a point. A one-dimensional cube is a segment, and can be viewed as a point swept in the first dimension. A two-dimensional cube is a square, and can be viewed as a segment swept in the 2nd dimension. A three-dimensional cube is a normal cube, and can be viewed as a square swept in the third dimension.

   A 4-dimensional cube (a *hypercube*) can be viewed as consisting of two copies of cubes in 3 dimensions: start with one 3d-cube and sweep it in the fourth dimension, dragging new edges between the original's cube vertices and the final cube.

   Compute the number of vertices and edges in a 4-dimensional hyper-cube. Try to think of this recursively in terms of the number of faces, edges and vertices in lower dimensions.

2. **Diameter.** (assume 2D) The *diameter* of a set of points $p_0, p_1, ...$ is defined to be the largest distance between any two points in the set: $max_{i,j}|p_i - p_j|$. Argue that the diameter of a set is achieved by two hull vertices.

   Note: It is known how to find the diameter of a point set in 2D in $O(n)$ time if the convex hull is known.

3. **Onion peeling.** (assume 2D) Start with a finite set of points $S = S_0$ in the plane, and consider the following iterative process: Let S_1 be the set $S_0 \setminus \delta H(S_0)$: S_0 minus the boundary of the convex hull of S_0. Similarly, define $S_{i+1} = S_i \setminus \delta H(S_i)$. The process continues until the empty set is reached. The hulls $H_i = \delta H(S_i)$ are called the *layers* of the set, and the process is called *onion peeling*. Any point on H_i is said to have *onion depth*, or just *depth*, i. Thus the points on the hull of the original set have depth 0.

   - Consider a set of points and draw its layers.
   - What is the maximum/minimum number of layers of a set of n points? Draw examples.
   - How would you go about computing the layers of a set of points and how long?

4. **Hull 3D.** The next assignment will be to implement the 3d-hull using: (1) the naive algorithm; and (b) either the gift wrapping or the incremental algorithm.

   Assume you start with a vector that stores the coordinates of the points.

One of the questions you need to think about is how to store the hull.

For this assignment, for simplicity, **you do not need to store the topology of the hull** (that is, the adjacency information for its faces, edges and vertices—for each edge, what faces it's contained into; for each vertex, what faces contain it, and what edges contain it). Instead you will store the hull as an array/vector of faces. Ideally the faces use pointers to points in P, rather than duplicating the points and their coordinates. For example:

```
typedef struct _face3d {
   point3d *p1, *p2, *p3;  //the vertices on this face (in ccw as looking from outside)
} face3d;
```

or, rather than pointers, store the indices of the points

```
typedef struct _face3d {
   int p1, p2, p3;  //the vertices on this face (in ccw as looking
from outside) are points[p1], points[p2], points[p3]
} face3d;
```

Or perhaps even simpler, a face can be just a vector of 3 (pointers to, or indices of) points.

(a) Pencil pseudocode for implementing the naive 3d hull algorithm. As discussed above, the output hull will be a vector of faces.

(b) **Degeneracies:** Consider what degeneracies might arise and how to handle them. In 2d, points that are collinear need to be handled separately and are considered degenerate cases. What are the degenerate cases in 3d? Come up with a (simple) way to handle them.

(c) Choose one of Gift wrapping and Incremental and write pseudo-code.

Note: In addition to the faces, you may need to keep track of what vertices are on the hull, what edges are on the hull, and whether an edge has found both its adjacent faces or only one of them. Come up with structures to keep track of these. Consider using hash tables and maps and such. For example you may want to keep a queue/stack that stores all the edges that are on the frontier of the search (in other words: all edges that have found only one adjacent face so far). As discussed, avoid storing a topological structure for the hull (to avoid the complexity or programming) —- your implementation does not have to be efficient, but it needs to work.