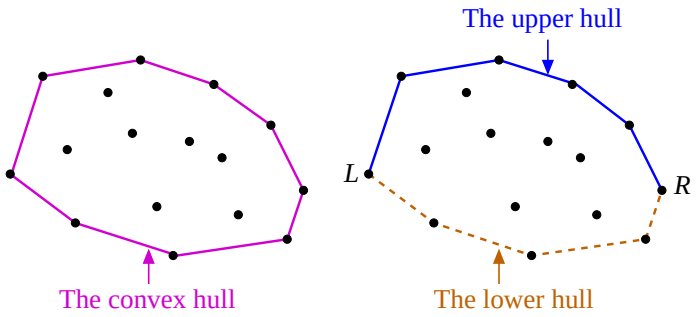
Convex hulls

We are given a (finite) set points in the plane. Our task is to compute the smallest convex region that contains all these points. It is easy to conceive that this smallest convex region must be a convex polygon (since the given set of points is finite). We call this smallest convex region the *convex hull* of the given set of points.

You may visualize a convex hull in the following way. Imagine that a nail is struck on a flat ground at each of the given points. We are given a sufficiently stretchable elastic rubber band. Our task is to place the rubber band in such a way that it encloses all the nails, and the total area enclosed by the stretched band is as small as possible. The next figure shows the convex hull of a set of sixteen points.



Being a convex polygon, a convex hull can be represented by a sequence of vertices appearing in the clockwise order. We can break this sequence in two parts. Let L and R be the leftmost and rightmost points in this polygon (clearly also in the given set of points). The clockwise listing of vertices on the polygon starting from L and ending at R is called the *upper hull* of the given points. Analogously, the clockwise listing of the vertices of the convex hull starting at R and ending at L is the *lower hull* of the given points. If n points are given, then the convex hull contains $\mathrm{O}(n)$ vertices (and edges). It then easily follows that given the convex hull, we can compute the upper and lower hulls in $\mathrm{O}(n)$ time. Conversely, given the upper and lower hulls, we can compute the entire convex hull in $\mathrm{O}(n)$ time. It, therefore, suffices to compute the upper and lower hulls individually.

For the sake of brevity, let me introduce some notations. Let S be the given set of points. The convex hull of S is denoted by $\mathrm{CH}(S)$, the upper hull of S by $\mathrm{UH}(S)$, and the lower hull by $\mathrm{LH}(S)$. We will assume that the points in S are in general position. In this context, this assertion indicates that the x-coordinates of the points in S are (pairwise) distinct, and also that no three of these points are collinear.

Preparata and Hong's divide-and-conquer algorithm

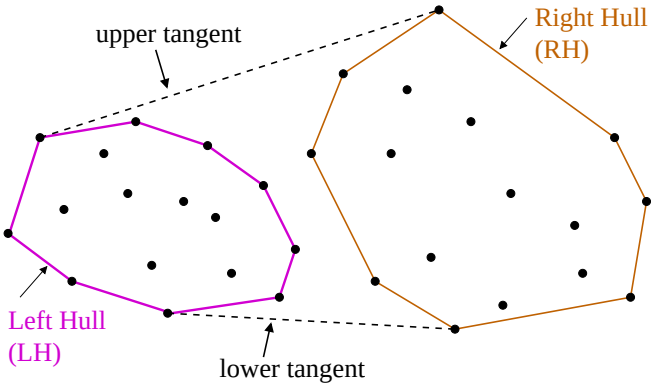
I now describe a recursive algorithm for computing $\mathrm{CH}(P_1, \ldots, P_n)$ that runs in time $\mathrm{O}(n \log n)$. Since sorting n points satisfies the same time bound, we may assume without loss of generality that the points $P_1, \ldots, P_n$ are already sorted with respect to their x coordinates. Moreover, we assume that the points are in general position so that the x coordinates of $P_1, \ldots, P_n$ form a strictly increasing sequence. Consider the following recursive algorithm.

```
if (n ≤ 3) manually compute and return CH(P_1, ..., P_n);
Recursively compute C_1 = CH(P_1, ..., P_⌈n/2⌉);
Recursively compute C_2 = CH(P_⌈n/2⌉+1, ..., P_n);
Merge C_1 and C_2 to C = CH(P_1, ..., P_n);
Return C;
```

Let $T(n)$ be the running time for this recursive algorithm. Assume that C_1 and C_2 can be merged in $\mathrm{O}(n)$ time. We then have:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathrm{O}(n) \quad \text{for} \quad n > 3\,.$$

We have seen that this divide-and-conquer recurrence has the solution $T(n) = \mathrm{O}(n \log n)$.

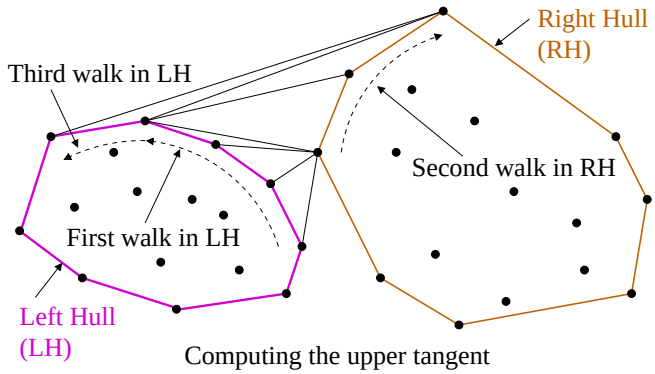


I need to supply an algorithm that can merge C_1 and C_2 in linear time. Since the points are sorted and have distinct x coordinates, the left hull C_1 is separated from the right hull C_2 by a vertical strip. In view of this, an idea illustrated in the above figure works.

A line L is called a *tangent* (or a *supporting line*) to a convex polygon if it touches the convex polygon and all vertices of the convex polygon lie on or to one side of L. We plan to compute the upper and the lower tangents common to both C_1 and C_2. We discard from both C_1 and C_2 all the points strictly between these two tangents. The remaining points can be easily presented in the form of a clockwise sequence defining the merged hull C.

Evidently, the merging algorithm runs in $\mathrm{O}(n)$ time, provided that we can compute the two tangents in $\mathrm{O}(n)$ time. Here I describe an algorithm for the computation of the upper tangent only. The determination of the lower tangent can be symmetrically handled.

The following figure illustrates the computation of the upper tangent. We let two points P_1, P_2 march along the perimeters of C_1 and C_2 respectively. Initially, P_1 is the rightmost point of C_1 and P_2 the leftmost point of C_2. The point P_1 jumps from a vertex in C_1 to the next vertex in the counterclockwise order, whereas P_2 moves in the clockwise order along the boundary of C_2. At every intermediate instant, we check whether the oriented segment $\overline{P_1P_2}$ is an upper tangent to C_1 and the oriented segment $\overline{P_2P_1}$ is an upper tangent to C_2. This means that we check whether both the neighboring points of P_1 lie to the right of $\overline{P_1P_2}$ and both the neighboring points of P_2 lie to the left of $\overline{P_2P_1}$.[1] If this condition is not satisfied at P_1 (resp. P_2), we move P_1 (resp. P_2) to the next vertex in the counterclockwise (resp. clockwise) direction along C_1 (resp. C_2). Eventually, P_1 and P_2 reach the vertices defining the upper tangent. As the following figure illustrates, the vertices defining the common tangent need not be the topmost vertices of the two hulls (See the left hull).



Computing the upper tangent

[1] Let L be the oriented line segment from a point $P_1 = (x_1, y_1)$ to a point $P_2 = (x_2, y_2)$. Assume that $P = (h, k)$ is any point in the plane. Consider the determinant:

$$\mathrm{side}(P_1, P_2, P) = \det \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & h & k \end{pmatrix}.$$

It turns out that the point P lies on, to the left, or to the right of the oriented line L according as the determinant $\mathrm{side}(P_1, P_2, P)$ is zero, positive, or negative, respectively.

The following code snippet summarizes the algorithm for computing the upper tangent.

```
Initialize P_1 to the rightmost point of the left hull C_1;
Initialize P_2 to the leftmost point of the right hull C_2;
while (P_1P_2 is not a common tangent to C_1 and C_2) {
   while (overline(P_1P_2) is not an upper tangent of C_1)
      advance P_1 to the next (counterclockwise) vertex of C_1;
   while (overline(P_2P_1) is not an upper tangent of C_2)
      advance P_2 to the next (clockwise) vertex of C_2;
}
Return (P_1, P_2);
```

Though intuitively clear, it demands a formal proof to settle that the above algorithm correctly computes the upper tangent, i.e., the moving points P_1, P_2 do not overshoot the respective vertices of tangency on C_1 and C_2. I leave the proof to the reader as an exercise and concentrate on the running time of the algorithm.

Let h_1 and h_2 be the numbers of vertices in C_1 and C_2 respectively. Though we have a nested loop in the code, every step in the walk advances either P_1 or P_2. Therefore, after at most $h_1 + h_2$ steps, the walk stops. Since $h_1 + h_2 \leq n$, we conclude that the upper tangent can be computed in $\mathrm{O}(n)$ time.