COMPUTATIONAL GEOMETRY.

GEORGE SALMAN

Contents

Remark. This book cover nice topics in Computational Geometry which worth to shed the light on. I am aware that such errors in text are possible. Don't hestitate to email me for fixing certain mistakes in case found.

Email: George.sa@campus.technion.ac.il.

Convex Hull and Mixing things.

Problem. (Mixing things). Given an oil well i.e., oil that is a mixture of a few different components however, the fraction of these component may be different and we are intereseted in mixing them in such a way in which we obtain a specific fraction of all components.

Example. Given the following fraction in table,

Substances	Frac A	Frac B
s_1	10%	35%
s_2	20%	5%
s_3	40%	25%

We want to mix two substances with a different fraction of A, B,

Substances	Frac A	Frac B
q_1	25%	28%
q_2	15%	15%

Using s_1, s_2, s_3.

Solution. The solution is to problem, is that not possible since we can't obtain q_2 substance however q_1 is obtained, let see how, the appealing approach is geometric approach, we may discribe s_1, s_2, s_3 as points in the plane, note that we may thing about s_1 as cooardinated $(s_1(Frac\ A), s_1(Frac\ B))$ similarly for s_2, s_3. Now, all possible fraction we can obtain from both is a line that connects $s_i, s_j, i \neq j$ defines by $\gamma(t) = (1-t)s_1 + s_2 t, t \in [0,1]$

Remark. for $t = 1$ we get s_2 and for $t = 0$ we get s_1 i.e., the meaning is that we only use s_1 and only use s_2 to mixture in each of this two values. Note that if we use $50 - 50$ we get a mixture with

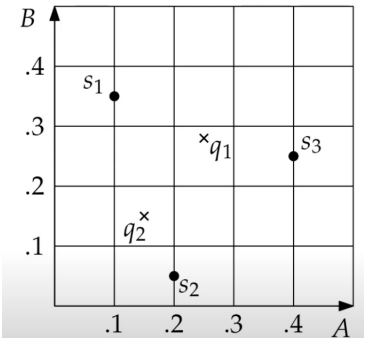
$$\gamma_{Mixture}(t) = (1-t)\,(s_1(Frac\ A), s_1(Frac\ B)) + t(s_2(Frac\ A), s_2(Frac\ B))$$

$$= (0.5 \cdot 10\% + 0.5 \cdot 20\%, 0.5 \cdot 35\% + 0.5 \cdot 5\%)$$

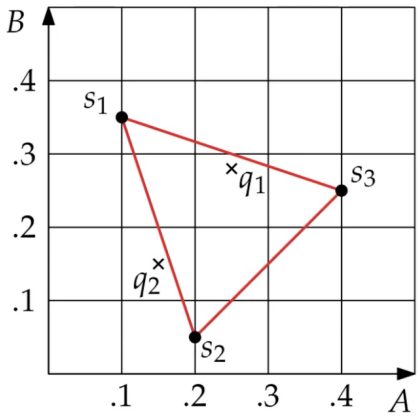
$$= (25\%, 20\%)$$

this is a point lies on the line connect s_1, s_2.

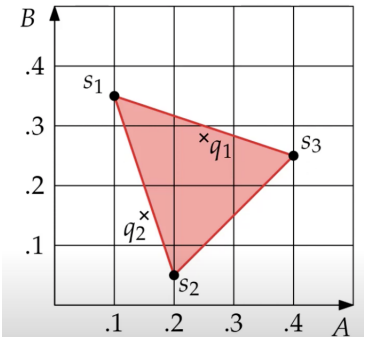
Now,



As mention above, each of the points s_1, s_2, s_3 we be connected and generated a triangle that looks as follows,



But there still all the points we obtain by mixturing all different substances, this points are the interior points of the triangle T, hence,

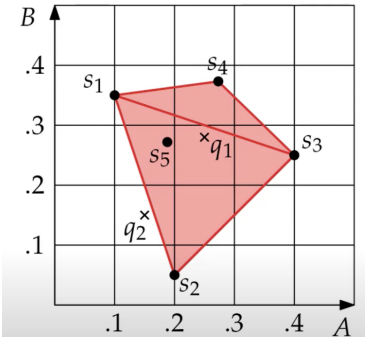


Now, its clear that $q_2 \notin (\partial T \cup Int(t))$ but q_1 do hence proposed mixture can't be obtained.

Claim. Given a set $S \subset \mathbb{R}^d$ of substances, we can mix a substance $q \in \mathbb{R}^d$ using the substances in $S \iff q \in CH(S)$ (CS- convex hull).

Proof. we will see after we define convex hull. $\Rightarrow$: Let $\vec{q}$ be a point in $\mathbb{R}^d$, we will show that $q \in CH(S)$. in which we can mix it using substances in S recall $s_1, ..., s_m$ i.e., $\exists a_1, ..., a_m, \sum_{i=1}^{m} a_i = 1$ where $q = a_1 s_1 + ... + a_m s_m$. Assume toward contradiction that $q \notin CH(S)$ so we can define $CH(S \cup q)$ but since $CH(S)$ is convex hull then is must give any mixture in $\mathbb{R}^d$ therefore, $CH(S) \subseteq CH(S \cup q) \subseteq CH(S)$ therefore, $CH(S \cup q) = CH(S) \Rightarrow q \in CH(S)$ so contradiction to the minimality of $CH(S)$. $\Leftarrow$ assume that $q \in CH(S)$ □

Example. for $d = 2$, If we add s_4 substance then we will get a corresponding convex hull would look like this,

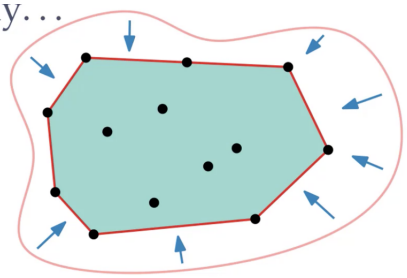


Note that if we add a point s_5 which is in the convex hull then it doesn't contribute i.e., we may ignore it because this mixture is already obtained with s_1, s_2, s_3, s_4 i.e., more formally $span\{s_1, s_2, s_3, s_4\} = span\{s_1, s_2, s_3, s_4, s_5\}$ where the span $span\{s_1, s_2, s_3, s_4\}$is defined by all $a, b, c, d \in \mathbb{R}$ where $as_1 + bs_2 + cs_3 + ds_4$ with constrain $a + b + c + d = 1$.

Formally. Given $S \subset \mathbb{R}^2$, how we define convex hull $CH(S)$.

- Physics approach - take large enough elastic rope - stretch and let go then take are inside and on the rope as in the following figure,

ıy. . .



- However, we can see that its hard to define mathmatically what the real definition of convex hull.

Definition. (1.1) (Math approach) a convex hull is the smallest convex set which contain all the points we are given i.e.,

$$CH(S) = \bigcap_{C \subseteq S:\ \mathrm{C}\ convex} C$$

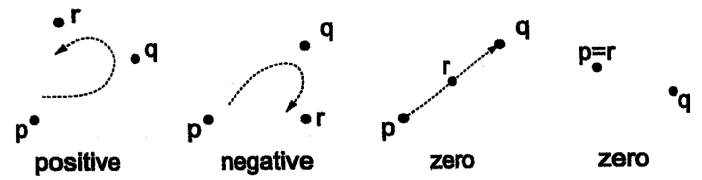
Definition. A half-plane is a planar region consisting of all points on one side of an infinite straight line, and no points on the other side.If the points on the line are included, then it is called an closed half-plane. If the points on the line are not included, then it is called an open half-plane.

Orientation: In order to make discrete decisions, we would like a geometric operation that operates on points in a manner that is analogous to the relational operations $(<, =, >)$ with numbers. There does not seem to be any natural intrinsic way to compare two points in d-dimensional space, but there is a natural relation between ordered $(d+1)$-tuples of points in d-space, which extends the notion of binary relations in 1-space, called orientation. Given an ordered triple of points (p, q, r) in the plane, we say that they have positive orientation if they define a counterclockwise oriented triangle, negative orientation if they define a clockwise oriented triangle, and zero orientation if they are collinear (which includes as well the case where two or more of the points are identical).

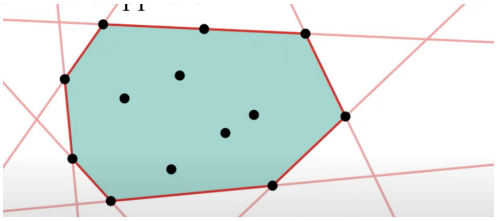
Remark. Note that orientation depends on the order in which the points are given.

Orientation is formally defined as the determining sign of the given points in homogeneous coordinates, i.e. by setting coordinate 1 to the each coordinate. For example, in the plane, we define

$$orient\,(p,q,r) = det \begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}$$



What the problem with definition (1.1)? the set is huge, and not feasible in cumputational terminology. Maybe we can do less? we can reduce the number as intersection of half-planes, by extending all egdes on the boundaries.



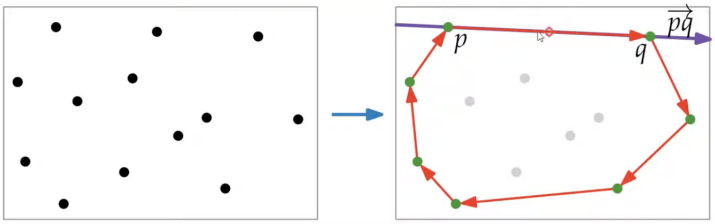
Each of the extended line on edges describes a half plane, and they are the only ones that we need to describle convex hull where CS is the intersection of the half planes we have created above, so in order to compute it we will look at the half plane that contain the input points neither the set of all convex set, this will reduce a lot however it still infinite amount, however, we can see that the extended edge cross at least two points on the boundaries of the extended edges.

Definition. Convex hell we can define it as,

$$CH(S) = \bigcap_{\substack{S \subseteq H \\ H \text{ Closed Halfplane}}} H = \bigcap_{\substack{S \subseteq H, |\partial H \cap S| \geq 2 \\ H \text{ Closed Halfplane}}} H$$

Algorithmic Approach. Our input is set S of n points in the plane, that is, $S \subset \mathbb{R}^n$. Our output is list of vertices of $CH(S)$ in clockwise order.

Observation. As in the definition we intriduced earlier, we can see that CS is not more that intersection of all half-plane with two points of S at leasts lie on it. Focus may on the following figure,



we can see that $\vec{pq}$ define a closed half-plane, because all the points in set S are below it or rely on it so indeed it split the plane. We can examine also that if we connect gray points we can never get half plane becuase there will be points above and below so we can see why the defintion work using that observation that for each edge (p, q) of $CH(S) \iff$ if each point in S lies strictly to the right on the directred line $\vec{pq}$ or on the line segment $\vec{pq}$. So each can take each pair of the points and check this property i.e., we will have $\left(\binom{n}{2} - n\right)(n-1)$ since we will have to check if all other $n-1$ points satisfy the required property, then we can take all of the segments that satisfy it out of the $\binom{n}{2} - n$ options.

Algorithm.

Remark. The logical condition marked in yellow is checked (meanrion in observation) by checking whether for vertices $r \in S$ with cooardinates x_r, y_r imply that

$$det \begin{pmatrix} x_r & y_r & 1 \\ x_q & y_q & 1 \\ x_p & y_p & 1 \end{pmatrix} < 0$$

FirstConvexHull(S):

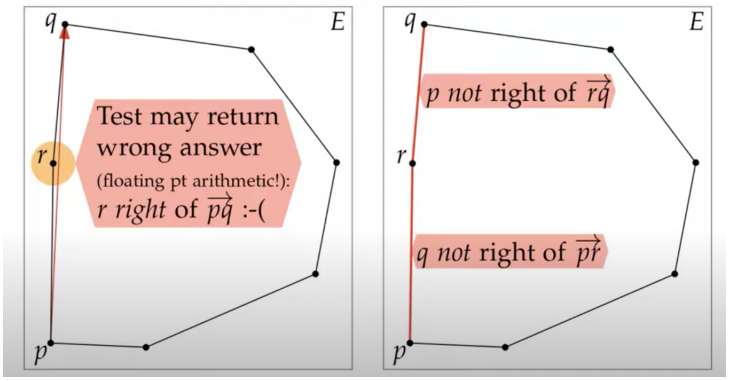
$E \leftarrow \emptyset$
foreach $(p,q) \in S \times S$ with $p \neq q$ **do** // test: (p,q) edge of CH(S)?
  $valid \leftarrow true$
  foreach $r \in S$ **do**
    if not (r strictly right of $\overrightarrow{pq}$ **or** $r \in \overline{pq}$) **then**
      $valid \leftarrow false$
  if *valid* **then**
    $E \leftarrow E \cup \{(p,q)\}$
from E construct sorted list L of vertices of CH(S)
return L

Algorithm 1: FirstConvexHull(S)

if $det = 0$ this means its on the line, and if bigger then it mean its strictly left to of the line segment $\vec{pq}$.

Complexity; The check in the observation cost is $O(1)$ *and for outer loop we have* $\binom{n}{2} - n$ *because we are looking for all pairs* $(p,q) \in SxS, p \neq q$ *and we have* $\Theta(n)$ *in which we check if point lies right or left or on the line hence in total* $\left(\binom{n}{2} - n\right)(n-1) = \Theta(n^3)$.

Discussion. The algorithm is not robust since the calculation of determinant may give wrong answer and consider segment is not valid when it is, or other sencario where $r \in S$ *will be right of* $\vec{pq}$ *becuase of floating point arithmetic.*



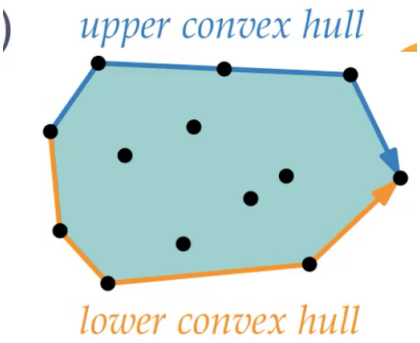
Example. Consider,

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 + \frac{1}{10^3} \end{pmatrix}$$

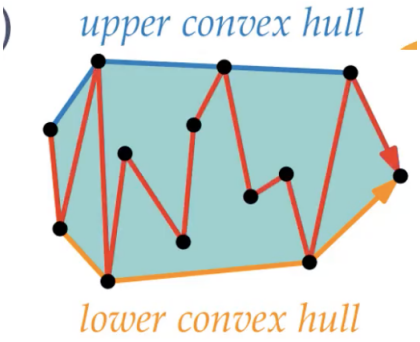
then $det(A) = 1 \cdot \left(1 + \frac{1}{10^3}\right) - 1 > 0$ but since our computer ignote that $\frac{1}{10^3}$ then we get $1 \cdot 1 - 1 \cdot 1 = 0$ which is not the expected, so we can see how that algorithm could fail for $3x3$ matrix by giving similar matrix bahviour.

Graham scan.

- Split computation in two



- bring pts in lexicographic order



- proceed incrementally

....

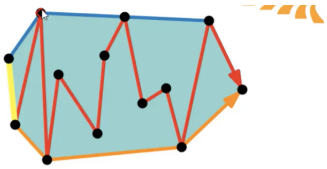
UpperConvexHull(S):

$\langle p_1, p_2, \ldots, p_n \rangle \leftarrow$ sort S lexicographically $\quad O(n \log n)$
$L \leftarrow \langle p_1, p_2 \rangle$
for $i \leftarrow 3$ **to** n **do** $\quad (n-2)\cdot$
  L.append(p_i)
  while $|L| > 2$ **and** last 3 pts in L make a left turn **do**
    remove second last pt from L ?
return L

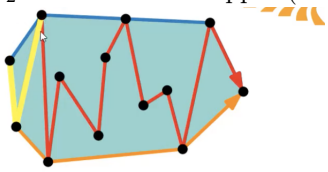
Algorithm 2: GrahamScan - UpperConvexHull(S):

Psude code - Graham scan.

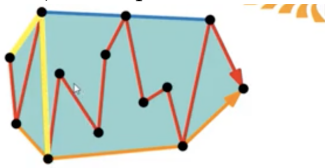
Running Example. First we start with p_1 which the smalles point with x corradination, if we do left turn then this means that the point we are going at is not correct so we delete it and try other point which give right turn, how this will look?



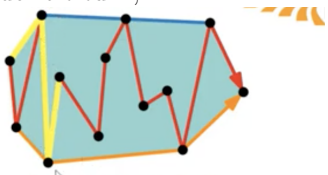
Observe that we first define the first edge edge for p_1, p_2 then we continue to next vertiex p_3, now we connect p_2, p_3 but now we can see that the convex hull os wrong since the upper convex hull make left turn when it should do only lrft turn so p_2 can't lie on the upper (did left turn to reach p_3) convex hull so we remove it



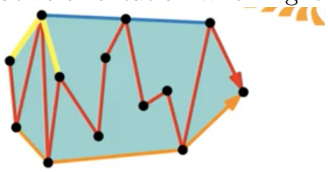
Now, we keep on from vertex p_3 we connect it to p_4 as follows,



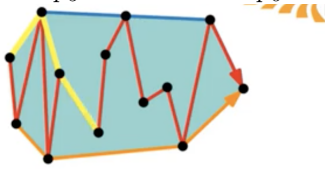
Now, its legal convex hull, but when we continue from p_4 to p_5 we can see that we do left turn,



so we remove p_4 and tro to connect p_3 to p_5 instead as follows becase we got a negative orientation when right turn is positive orientation i.e., clockwise direction



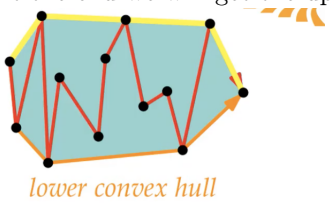
from p_5 we connect to p_6 as follows,



Now, we throw p_5 because we test latest three point on the convex hull whther they make left turn and indeed it did from p_5 to p_6.

Remark. In order to know whether its left or right turn we look at the specific vertice and check that with logical conditnion.

At the end we will get the upper convex hull marked in yellow,



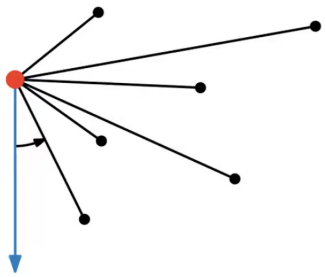
Running time analysis. the sorting part cosr $O(nlogn)$ then two nested loos where puter is $n-2$ and inner loop order pf n so we will have $O(n^2)$ howeverm this is not accurate so we will use amortized time analyisis because the inner loop can't happen too often since when we remove a point from upper convex hull it will never gets in again so each point can only be added once and removed once so we can formailze it,

- Each pt $p_2, ...p_{n-1}$ pays one euro for its potential removal later on
- this pays for total effor of all execution os the while loop
- So the two total time of two loops is $O(n)$

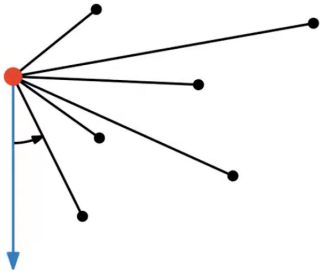
Theorem. *We can compute the convex hull of n pts in the plane in $O(nlogn)$ time - in robust way.*

Output-Sensitive Algorithms.

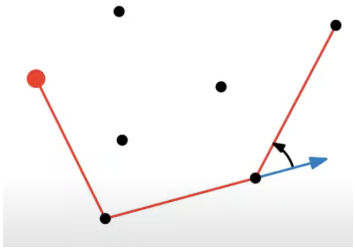
Jarvis gift wrapping algorithm (aka Jarvis' march).



We first take vertical ray and start rotating it untill we get to the next vertex with the smallst angle obtained with the vertical ray, from this vertex we choose again smallest angle possible with the other vertices,

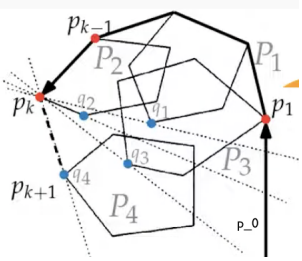


So we will get the follow,



note that we check angles for each vertex we get in the convex hull i.e., for each vertex in the convex hull we compare the angle it generates with other n vertices hence the complexity is $O(nh)$ where $h = |CH(S)| =$ size of the output. This is optimal when h is pretty small. A more optimal approach is called Chan's exponential-search algorithm with complexity of $O(nlogh)$ which is intricate, however, we will present roughly the idea.

Chan algorithm code.



Algorithm $Hull2D(P)$, where $P \subset E^2$:

(1) for $t = 1, 2, ...$ do
(2) $L \leftarrow Hull2D(P, m, H)$ where $m = h = min\left\{2^{2^t}, n\right\}$
(3) if $L \neq$ incomplete then return L

Algorithm 3: Hull2D(P)

Algorithm $Hull2D(P, m, H)$, where $P \subset E^2, 3 \leq m \leq n$ and $|H| \geq 1$:

(1) partition P into usbets $P_1, ..., P_{\lceil \frac{n}{m} \rceil}$ each of size at most m. (In figure above we took $m = 5$)
(2) for $i = 1, ..., \lceil \frac{n}{m} \rceil$ do
(3) computer $conv\,(P_i)$ (convex hull of P_i) by Graham scan [in $O(mlogm)$ time] and store its vertices in an array in ccw (clockwise) order
(4) $p_0 \leftarrow (0, -\infty)$
(5) $p_1 \leftarrow$ the rightmost point of P
(6) for $k = 1, .., H$ do
(7) for $i = 1, ..., \lceil \frac{n}{m} \rceil$ do
(8) compute the point $q_i \in P_i$ the maximizes $< p_{k-1}p_kq_i$ $(q_i \neq p_k)$ by performin a binary search on the vertices if $conv(P_i)$
(9) $p_{k+1} \leftarrow$ the point q from $\left\{q_1, ..., q_{\lceil \frac{n}{m} \rceil}\right\}$ the maximizes $< p_{k-1}p_kq$

Remark. Jarvis march algorithm is what we do in line 8-9.

(1) if $p_{k+1} = p_1$ then retun the list $< p_1, ..., p_k >$
(2) return incomplete

Algorithm 4: Chan's Algorithm

Chan's approach and analysis. each group, and so the number of groups is $k = ceiling(n/m)$. For each group, we compute its hull using Graham's scan, which takes $O(mlogm)$ time per group, for a total time of $O(kmlogm) = O(nlogm)$. Thus, after this step, we have k subhulls. Note that since the points in a set were selected arbitrarily, these subhulls may intersect with each other, one may completely lie within another, or they may be disjoint. Next, we use the Jarvis march algorithm (to compute convex hull of a 2D point set) on the groups. Here we take advantage of the fact that you can compute the tangent between a point and a convex m-gon in O(log m) time. The idea is to find the next vertex of the convex hull, p_{i+1} by knowing a current vertex, p_i. We select the leftmost point among all the vertices of all the subhulls. This point is guaranteed to be in the convex hull of P. We name this point as $p_0, i = 0$. Then, we find the tangents from pi to each of the subhulls. The tangent which makes the lowest angle with p_ip_{i-1}, its point of contact with the subhull will surely be the next vertex of the convex hull of P.

Remark. Note Finding the tangent from an external point, q to a convex hull: Without loss of generality, we can assume that the vertices of the convex hull are ordered in clockwise direction. We perform a binary search on the list of vertices, choosing sides by calculating the angles made by the edge qp_i with the two edges which contain the point, p_i. where p_i is the point of tangency if the two edges containingp_i lie on the same side of qp_i. This new line segment is the corresponding tangent.

We repeat until the newly found hull vertex coincides with p_0. So, as before there are h steps of Jarvis's march, but because we are applying it to k convex hulls each step takes $O(klogm)$ time. Each iteration consists of finding a tangent using binary search $O(logm)$ and then finding the tangent which makes the least angle with the current edge of the convex hull $(O(k))$. Thus, the cost of each iteration $= O(klogm)$ and total running time $= O(hklogm) = O((hn/m)logm)$. Combining

the two parts, we get a total of $O((n+hn/m)logm)$ running time. We can observe that if we set $m = h$, we can get $O(nlogh)$ running time. There is only one small problem here. We do not know what h is in advance, and therefore we do not know what m should be when running the algorithm. We will see how to take care of this later. The algorithm works correctly as long as $m \geq h$.

Choosing the value of m. The remaining thing is choosing the value of m correctly. We can choose m to be 1, 2, 3 and so on, till we get $m \geq h$. But this will increase the time complexity. Binary search will definitely be more efficient than this but if we use binary search, we may guess too large value of m (for example, $m = n/2 = O(n)$), which will in turn increase the time to $O(nlogn)$ So, Chan uses the trick that we start with a small value of m and increase it rapidly. Since the dependence on m is only in the logarithmic term, as long as our value of m is within a polynomial of h, that is, $m = ch$ for some constant c, (in other words $m = O(h)$) then the running time will still be $O(nlogh)$. So, our approach will be to guess successively larger values of m, each time squaring the previous value, until the algorithm returns a successful result. This technique is often called doubling search (because the unknown parameter is successively doubled). The only difference is that in Chan's algorithm, we will be squaring the old value rather than doubling.

Remark. We should note that 2^{2^t} has the effect of squaring the previous value of m.

Time and space complexity. For the t-th iteration, running time $= O(nlog(m_t)) = O(nlog2^{2^t}) = O(n2^t)$. We know that it will stop as soon as $2^{2^t} > h$, that is if $t = ceiling(lg(lgn))$. So, the total running time is

$$\sum_{t=1}^{lglgh} n2^t = n \sum_{t=1}^{lglgh} 2^t \leq n \cdot 2^{(1+lglgh)} = 2nlgh = O(nlgh)$$

which is what we wanted to achieve.

Link: https://personal.utdallas.edu/~daescu/convexhull.pdf.

Sweep-Line Algorithm.

Assuming we have a map and some of the features we want to highlight for e.g., cities, rivers, streets

most of the time we want to combine them to a single map which is called map overlay in Geographic Informatics Systems (GIS),

For example if we have all the roads and rivers we can find instantly all the bridges because they are the intersection, how could we do it quickly? i.e., scan the map and find all the bridges.

Definition. assuming we have to segments and the end of one of them lies in the middle of the other do we cound it or not? this depends on the how we define. In this book we are going to consider it as intersection between two segments.

Problem. Given a set S of n closed non-overlapping line segments in the plane, compute

- all points where at least two segments intersect and
- for each suck point report all segments the contain it

check if two segments intersect.

Definition. Two segments ab and cd intersect if and only if:

- the endpoints a and b are on opposite sides of the line cd, and
- the endpoints c and d are on opposite sides of the line ab.

To test whether two points are on opposite sides of a line through two other points, we use the same counterclockwise test that we used for building convex hulls. Specifically, a and b are on opposite sides of line cd if and only if exactly one of the two triples a, c, d and b, c, d is in counterclockwise order. So we have the following simple algorithm.

```
INTERSECT(a, b, c, d):
  if CCW(a, c, d) = CCW(b, c, d)
      return FALSE
  else if CCW(a, b, c) = CCW(a, b, d)
      return FALSE
  else
      return TRUE
```

(i.e., https://www.geeksforgeeks.org/orientation-3-ordered-points/ (Orientation of 3 ordered points) this could be done with the terminology of orientation) Or even simpler:

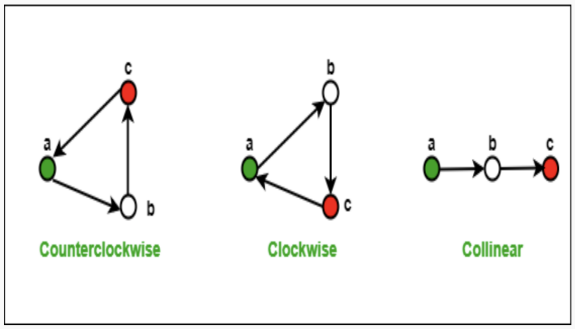
$$\underline{\textsc{Intersect}(a,b,c,d)\text{:}}$$
$$\text{return } \Big[\text{CCW}(a,c,d) \neq \text{CCW}(b,c,d)\Big] \wedge \Big[\text{CCW}(a,b,c) \neq \text{CCW}(a,b,d)\Big]$$

we can see that in case $CCW(a,c,d) = CCW(b,c,d)$ accordin to the following figure,

Orientation of an ordered triplet of points in the plane can be

- *counterclockwise*
- *clockwise*
- *collinear*

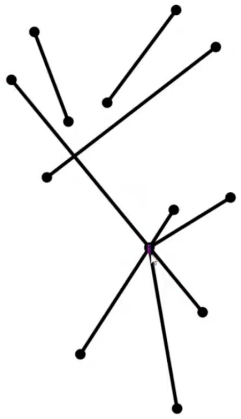
The following diagram shows different possible orientations of (a,b,c)



If orientation of (p1, p2, p3) is collinear, then orientation of (p3, p2, p1) is also collinear.
If orientation of (p1, p2, p3) is clockwise, then orientation of (p3, p2, p1) is counterclockwise and vice versa is also true.

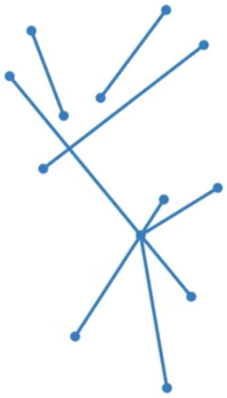
we can see that the defintion of intersection is celarly not satisfied because we need exactly one of the two triples a, c, d and b, c, d is in counterclockwise however we got both are counter clockwise which mean that no such intersection. (because then both do the same rotation i.e., a, b are on the same half-plane so no intersection).

Example. Assuming our set is the following set,

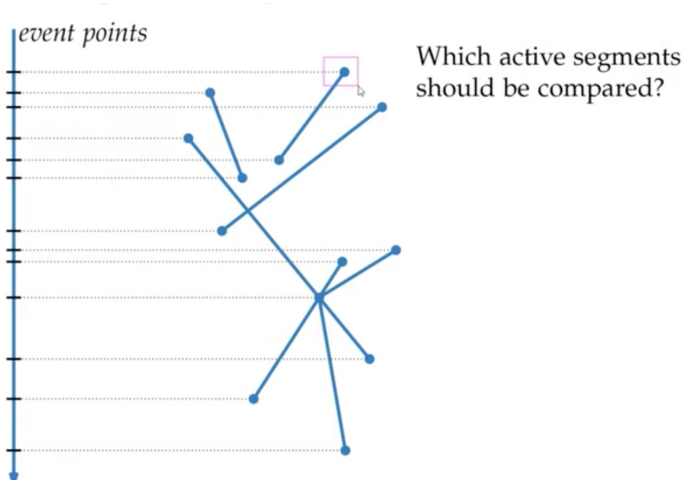


One way to doit is brute force by taking each two segement and check whether intersect we will have complexity of $O(n^2)$, but we can do it in $O(nlogn)$ how? the idea is proccessing segments top-to-bottom using a "Sweep line".

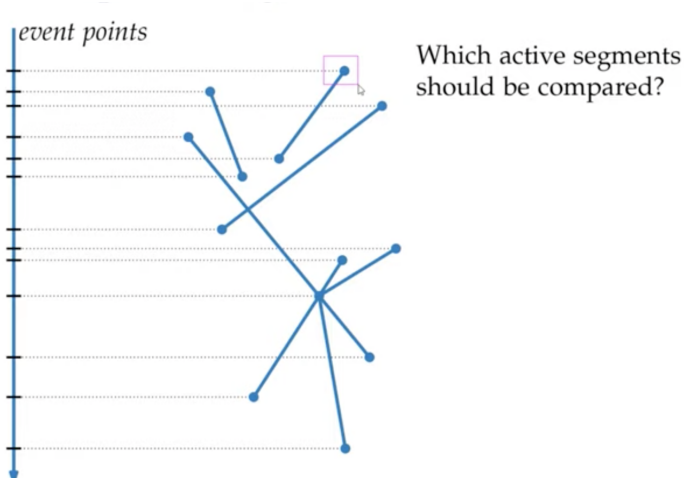
Sweep line algorithm.



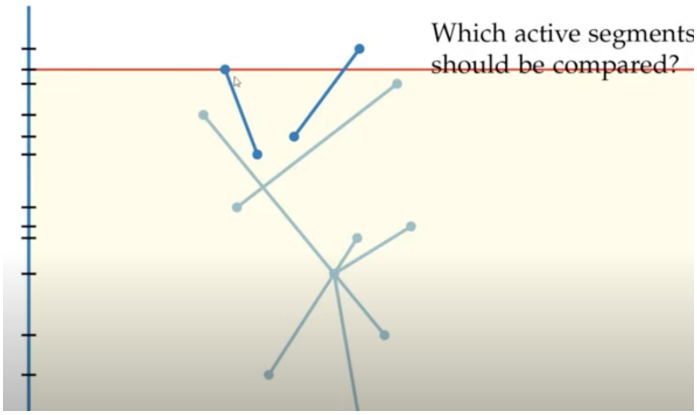
first we take some horizontal line that move downwards in the very beginning before it hits any segment it doesn't have to do anything but as soon it hits the first segment we have to start to process somehhing and whenever its hits any other something we have to do something and when it reach end point it have to do something so we get even point as follow,



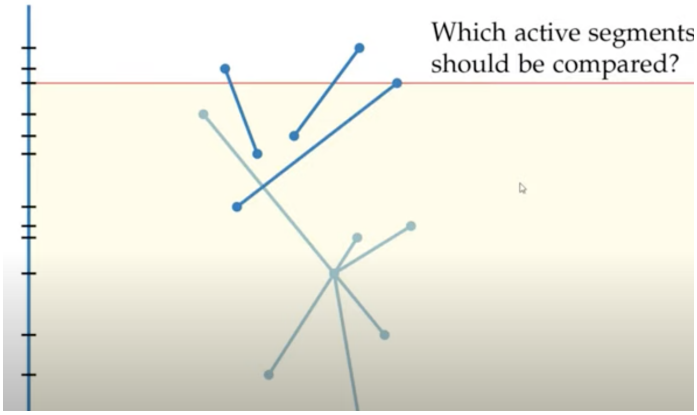
we move the horizonal line till we get our first active segment is the the one marked in a emphasized blue,



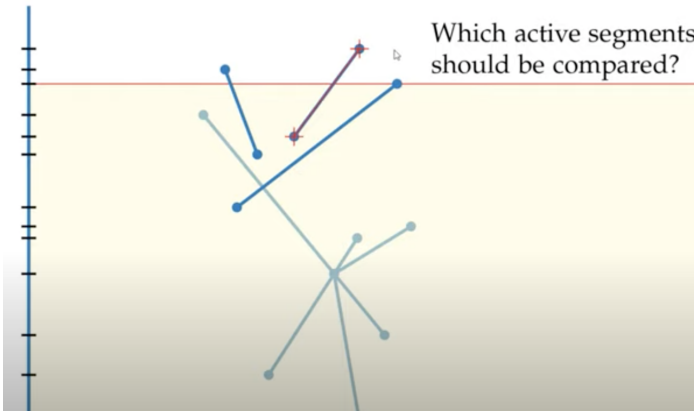
we keep moving and get the following segment which is also avtive



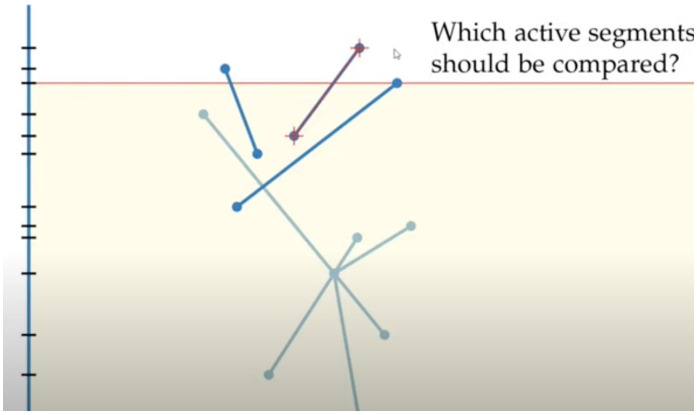
now we compare this two active segments and check if there is such intersection in this case there are not so we continue till we get the third active segment,



we can compare the segment with the other two and check whether intersecion occures but do we really have tom campare the new active segment with the old active segments? no, we only compare it to its neighbots active segment which is the red segment in the following figure,

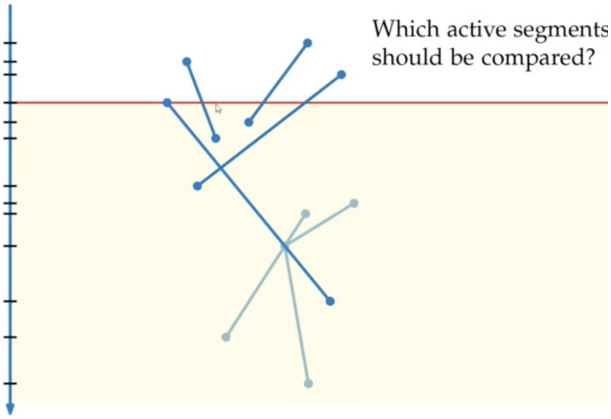


this red segment lies between the two other active segments so the two other active segment can not intersect with each other untill at least one of them has intersection with the red line i.e., one of them must overtake the red segment before it can intersect the other one so untill that doesn't happen we don't have to compare the red segement so they connot intersect till the end of the red segments for sure



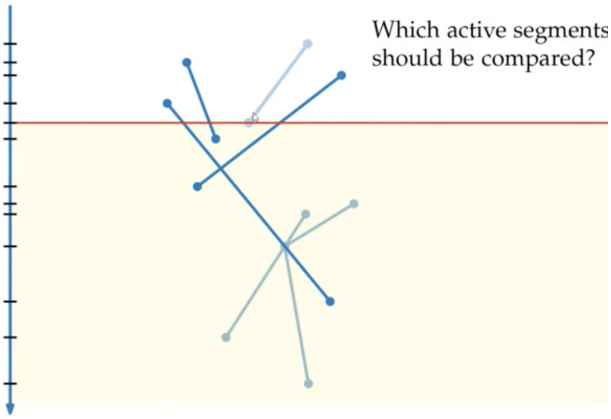
we keep downwaird till we reach the start of new segment,

Sweep-Line Algorithm



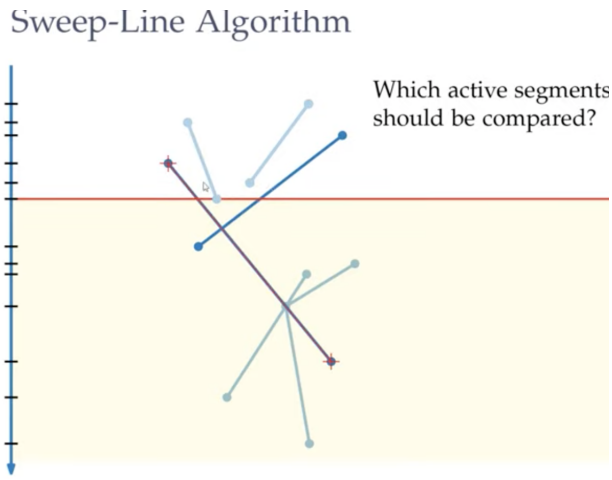
now we keep moving,

Sweep-Line Algorithm

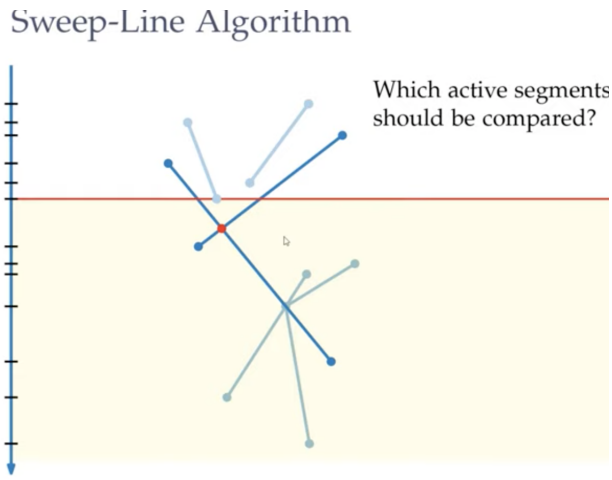


we get to the end of the red segment so we remove it and now we have to compare between the two neighbor segments of the red segment because the red

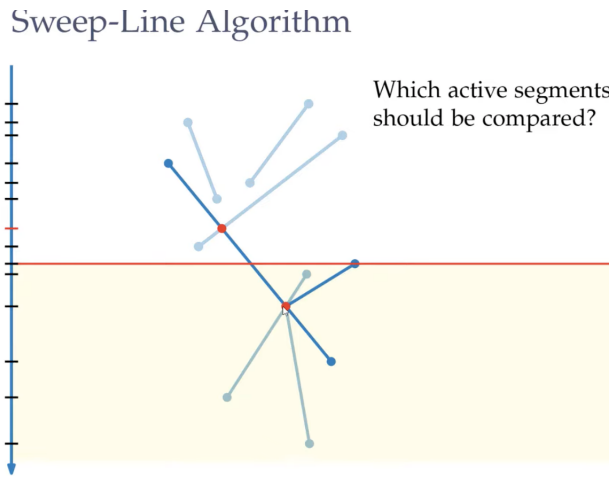
segment deesn't lie between them anymore so such intersection possible, we keep moving,



we reach the end of a segment, and now tbe two active segment are neighbots on the sweepline and we have to compare then we get intersection in the red point,



now we continue till we reach new active segment and get new intersection point,



now we continue till we get new active segment and fine that it intersect with one of the active segments in the point we already found and we move fowards the sweep line till we are done.

What are the data structures? first, we have all the event points and the intersection points that we have to insert to the data sructure that contain all end events so we need easy data structure that can find easily put new points and easily find the topmost point also we have to know in which order we have the segments currently on the sweep line so we have some segments and we want to order them from left to right by their intersection of the sweepline so we need other data structure where we can save a bunch of segments ordered where we can add new segments, remove, and switch the order between the two neighbors.

Data structure.

(1) Even points queue
(2) Sweep line statues T

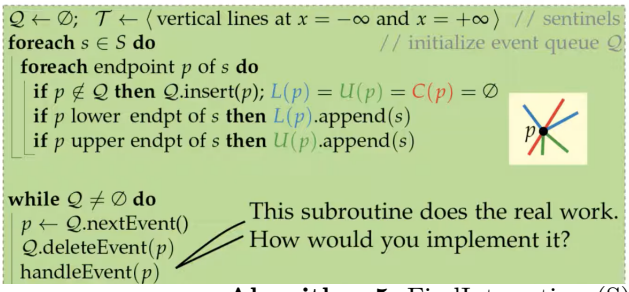
Definition. A point $p \prec q \iff y_p > y_q$ or if $y_p = y_q$ and $x_p < x_q$ (because we want to go from left which is our priority in case y values not equal) .

Our first data structure in Binary search tree acc. to $\prec \Rightarrow$ we can find next event, delete event, insert in $log\,(|Q|)$. The second data structure will store the segments intersected by l the sweep-line in left to right order. How? In a balanced binary search tree, so each time the sweepline go down we add new event and new segments intersected with it and we can detect intersection, and we sweepline reach end of segment we delete the segment from both data structures because its not relevant anymore.

Psuedo-Code. Given the function findIntersections(S) where our inpus is set S of n non-overlapping closed line segments, andour output is set I of inttersection points.

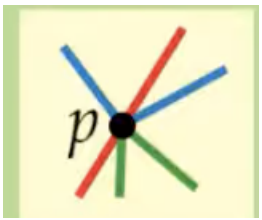
Sweep-Line algorithm code.

FindIntersetions(S):



Algorithm 5: FindIntersetions(S)

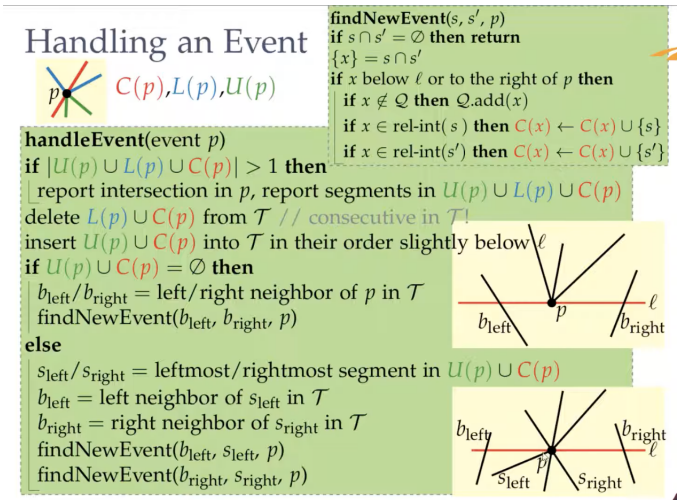
Code-Explain (roughly). In line 1 we initialize empty priority queue and two vertical lines, now we initialize our event queue, so we walk on all input segments and for each end point of segment we add it to the Q if its not added already. For example, assumine we have the following point p



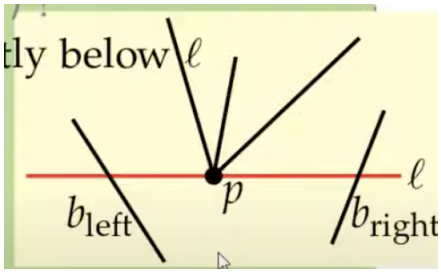
so we can see that p could be one of the three following set:

- $L(p)$: segments where p is the lower point of the segment (end point e.g., in the picture above p is end point of the blue segement)
- $U(p)$: segments where p is the upper point of the segment (e.g., in the picture above p is end point of the green segement)
- $C(p)$: segments where p lies inside the segment (e.g., in the picture above p is lies on the red segement we add it by checking with such active segment intersect and then we can detect it, we know if they intersect by seeing the the order of the segment change or in other word one start segment is in left of other segment and the end of it is in the right)

first all of the three set are empty, then we check which type of point p is respect to the segment include it then we add it to the set accordingly, so for now we intialized our data structure, and here come the step to process the data in the while loop (line-7), we check as long Q not empty we take the next point, delete it and hendle the event where the the handle event function is the core of the algorithm. How its implemented?



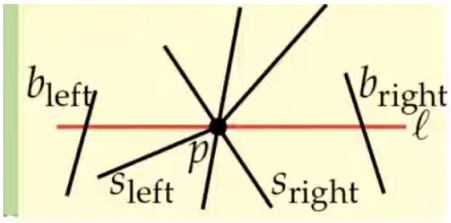
Remember that in the initialization we find all the events so when the sweepline reaches a point p and we check which segment that end of it they will be in the set $L(p)$ or start with it in $U(p)$ or segment where p lies on it in $C(p)$ so we can know how many segments hence, we can see that there is intersection between two segments $\iff |C(p) \cup L(p) \cup U(p)| > 1$ i.e. more than one segments contain p so we can report intersection. Now, since the order change in the tree we we delete $C(p) \cup L(p)$ and insert $U(p) \cup C(p)$ into T in their order slightly below l now after changing the order we can have new neighbors which wasn't before so for all of them we need to check if there is new intersection, if there is no segment in the bottom i.e., $U(p) \cup C(p) = \emptyset$ then there can be two neibors which wasn't before and became now, for example,



below l we can see that now b_{left}, b_{right} can intersect so we have to compare them and check whether there is intersection between them with $findNewEvent(b_{left}, b_{right})$ where $findNewEvent$ implemented as follow,

findNewEvent(s, s', p)
if $s \cap s' = \emptyset$ **then return**
$\{x\} = s \cap s'$
if x below ℓ or to the right of p **then**
  if $x \notin \mathcal{Q}$ **then** $\mathcal{Q}$.add(x)
  if $x \in$ rel-int(s) **then** $C(x) \leftarrow C(x) \cup \{s\}$
  if $x \in$ rel-int(s') **then** $C(x) \leftarrow C(x) \cup \{s'\}$

now examine some case, if indeed intersect and the intersection point x is left of p this means that this intersection is already treated and we continue, if its right of p or below p when we add it to Q and update $C(x)$. Now the certain case, where we have something as follows,



now that now we do have segment in the bottom and after reporting intersection the order change we we already mentioned how to update the trees structure accoringly, now s_{left} is the bottm part of the right segment below l and s_{riht} is the bottom part of left segment below l now, s_{left}, s_{right} may intersection with left neighbor and right neighbors so we need to check if there is new event and update the strcuture and then we are finished.

Correctness.

Lemma. *FindIntersection(S) correctly compute intersection points and the08 segments contain them.*

Proof. In induction, let p be an intersection pt assume by induction that,

- Every interior point $p \prec q$ has been computed correcctly.
- T contains all segment interseting l in left to right order

Case 1. p is not an interior point of segment $\Rightarrow$ p has been inserted in Q in the beginning. Segement in $U(p)$ and $L(p)$ are store with p in the beginning. When p is proccessed, we outpit all segment in $U(p) \cup L(p)$ $\Rightarrow$ all segment that contain p are reported. (Sweep line status is still correct).

Case 2. p is an interior point of segment, i.e., $C(p) \neq \emptyset$ if p not endpoint, need that p is inserted into Q before l reaches p. Let $s, s' \in C(p)$ be neighbors in the circular ordering of $C(p) \cup \{l\}$ around p, Imagine moving l slight back in time, Then s, s' were neighbors in the left-to-right order on l (in T). At the beginning of the algo, they weren't neighbots in T, so this mean that p must be intersection point

and there was some moment when they become neighbors! This is when $\{p\} = s \cap s'$ *was inserted to* Q*.*

Remark. In case 2 - we also need that every segment with p as an interior points in added to $C(p)$ so we can check all the neighbots in the circular order and apply the same argument.

□

Running time. Observe that FindIntersetions(S) in the beginning it look at all the segment and endpoint in them and for each end point we insert to the poriority queue and insert the segment into some set of the points which not take much times. Inserting into the PQ take $log(|Q|)$ where we have n segment and n points so in total $nlogn$, but later in the $WhileQ! = \emptyset$ we have no clue about number of points we will have in Q because in handlEvent some intersection points may be added and we don't know much there is, however, we may find upper bound on the number of intersection could be obtained, let analyze the time complexity of the loop first in $p \leftarrow Q.nextEvent$ which is $log(|Q|)$ and remove it in $log(|Q|)$ then handling it, how much cost the handling? in the handle event we first check the condition $|C(p) \cup L(p) \cup U(p)| > 1$ i.e., whehter p is intersection point if yes then we report it then since there is order change we delete and insert $2log(|C(p)|) + log(|U(p)|) + log(|L(p)|)$, then we make check if $C(p) \cup U(p) = \emptyset$ then we find left/right neighbor and apply FindNewEvent function on left/right neighbor with point p.

Observation. The maximum number of event point in Q can be $2n + n(n-1)/2 = O(n^2)$ event points where $2n$ is the start and end point of each segment and $n(n-1)$ is number of maximal intersection for each segment each of the sgement can intersect with the $n-1$ other segments so $|Q| = O(n^2)$ therefore, the loop can at most do each time is $O(logn^2) = 2O(logn)$. Now, all of the operation are $O(logn)$ however, the interesting part is to find how much time will take to delete and insert $U(p) \cup C(p), L(p) \cup C(p)$ respectively. This is output sensitive i.e., depends on the number of intersection points we want to report i.e., we can state the following lema.

Lemma. *FindIntersetions(S) finds* I *intersection points among* n *non-overlapping line segments in* $O((n+I)\,logn)$ *(i.e., depend on the number of intersection points).*

Proof. Let p be an event point,

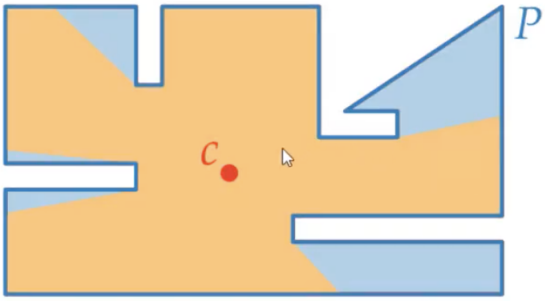
$$m(p) = |U(p) \cup C(p)| + |L(p) \cup C(p)|$$

and $m = \sum_p m(p)$ then its clear that the runtime is $O((m+n)\,logn)$. We want to show that $m \in O(n+I)$, from graph theory we can define Geometric graphss $G = (V, E)$ with $V = \{$end points,intersection points $\}$ number of vertices is $|V| \leq 2n + I$. For any $p \in V : m(p) = deg(p)$ hence,

$$m = \sum_p deg(p) = 2|E|$$

$\leq_{Euler-planar\ graph} 2\,(3|V| - 6) \in O(n+I)$ □

Guarding Art Galleries or Triangulating polygons. Given a simple polygon P (i.e., no holes, no self intersection)....

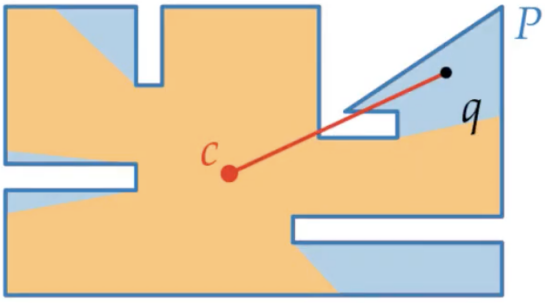


assuming we place a guard at point c, the orange space denote all the points that are visible by the guard in point c. This is done by rotating around c.

Observation. Guard c "sees" a start shaped region.

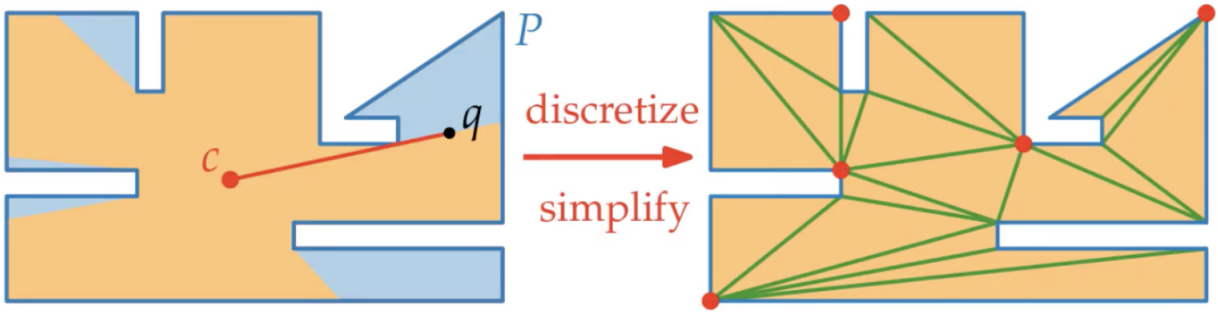
Definition. A point $q \in P$ is visible from $c \in P$ if $qc \subseteq P$.

Example. For some q in P as follow



examine that the line qc is not in P so q is not visible to c.

Aim. use few guards! but minimizing them is NP-hard. Given the polygon above we can paritition it into traigles (triangles are convex) and can do it as follows,



so 5 guards are needed.

Theorem. *The following statments,*

(1) *Every simple polygon can be triangulated.*

(2) *Any triangulation of a simple polygon with n vertices consits of $n-2$ triagnle.*

Proof. We will prove both statment with induction, our base is $n = 3$ where get triangle $n - 2 = 1$. Assuming we show it for all polygons with $n - 1$ vertices its true, we can pick one polygon corner v with angle < 180 (exists since sum of angles in polygon is $180(n-2)$) we cak take neigbors of v on the polygon boundary recall u, w and we connect u, w then we get two scenarios □

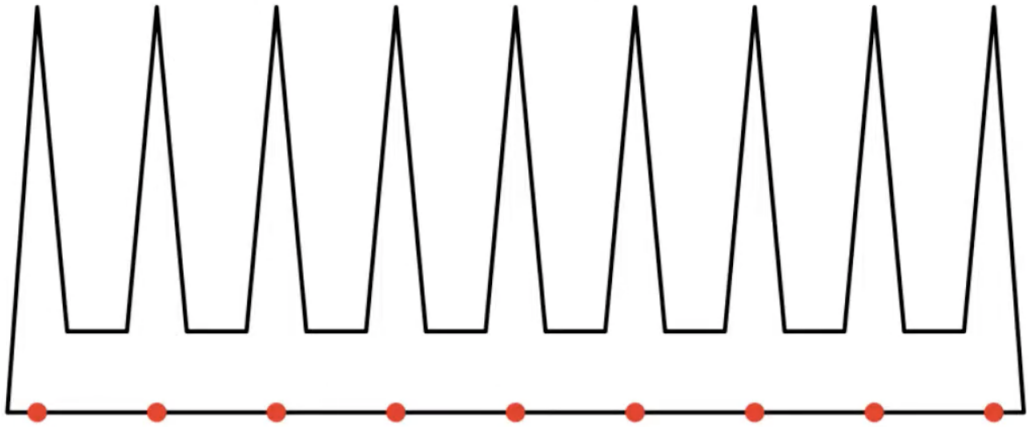
if w, u line in in the polygon then we will have a part of $n - 1$ vertices and one with 3 vertices so by assumption we will have $(n - 1) - 2 + 1 = n - 2$ triangles. In the other case, the line $wu \notin P$ then then must be a corner x so we get two polygons one with m vertices (green) and the other $n - m$ so together by induction we will get $n - 2$ triagnles.

The Are Gallery Theorem.

Theorem. *For surveilling a simple polygon with n vertices, $\lfloor \frac{n}{3} \rfloor$ guards are sometimes necessary and always suffcient.*

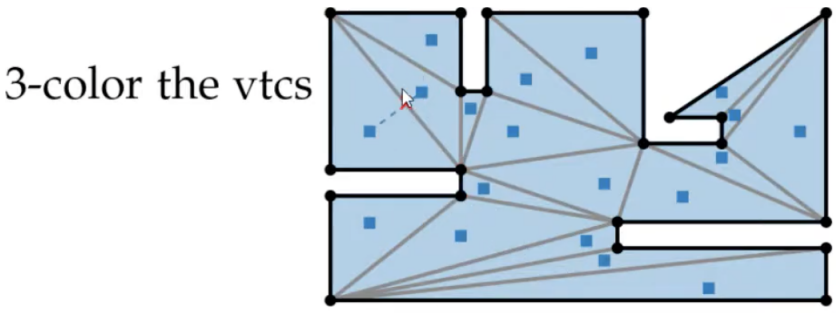
Exercise. Find for arbitrarily large n, a polygon with n vertices where $\approx n/3$ guards are necessary.

Consider the following polygon,

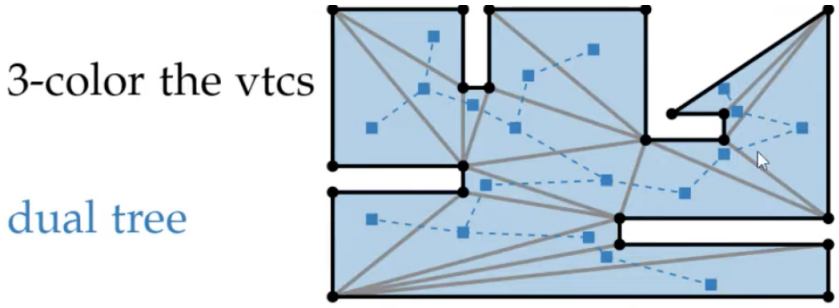


note that there is n spikes in each there is 3 points and we need $\approx n/3$ guards for surveiling.

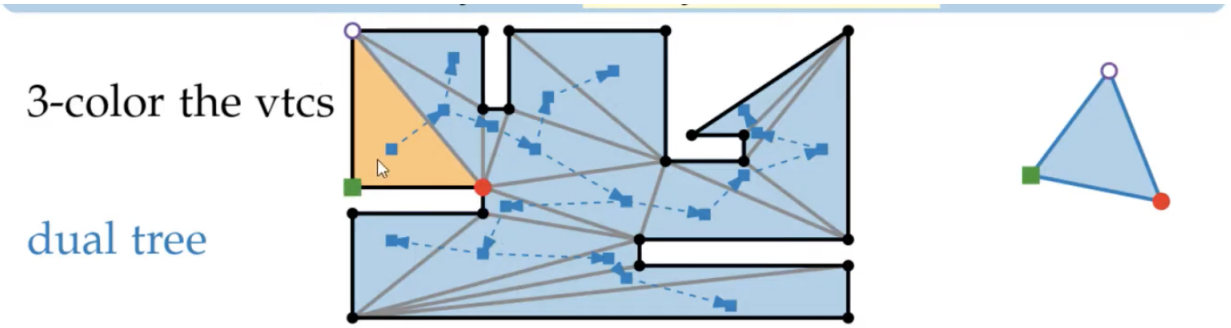
Motivation for proof. We want to use the fact the every simple polygon we can tringulate into $n-2$ triangles, we will 3-color the vertices i.e., no two vertices which are neighbors have the same color. Here we use the fact that triangulations of polygons 3-colourable which is always true for outerplane graph. To obtain 3 coloring we will use the dual graph by placing a vertex in every face (i.e., in each triangle)



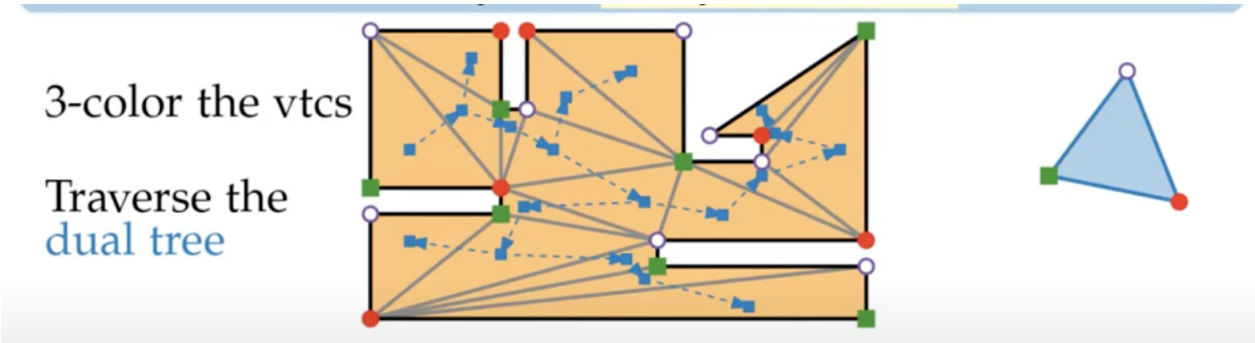
where each vertices are connected if the corresponding face share an edge so we will get the following graph (in dots) which obtain tree



we choose a leaf in the dual tree which gives us order in wchih we c an process all the traingles



we give the first triagnle three color then the next triangle is forced to be with the same colors on shared vertices with the first triangle and forth...



so every triangle has vertex of every colors as we examine above, what we do with this colors? Assuming we pick a color e.g., green and we place camera on each green vertices and so we cover all the triangles i.e., each of this colors clasess will always give us art gallery surveilliance, and by picking the class with smallest numbe of vertices which will have at most $n/3$ so we showed the art gallery theorem, however, how can we tringulate problem? for this we need algorithm, we presenet a brute force approach which follow existsance proof using recursion with running time of $O(n^2)$, a faster tringulation is obtained in two two steps, first given $n - vtx$ polygon $\underset{O(nlogn)}{\rightarrow}$ "nice" pieces, n' vertices $\underset{O(n')}{\rightarrow}$ n'' triangles. However, what are this

nice pieces what they look like? we are intereseted in cunvex pieces which is not easy so we want to introcude a formal definition for it.

Definition. (y monotone pieces) a polygon P is y monotine if, for any horizontal line $l, l \cap P$ is connected.

What the definition want to achienve? remember that in each convex polygon each line that connect two ertices lies on the polygon however, in P polygon which is y montone the property satsifed in case two vertices has the same $y-$ coordinate or in other word, each horizontal line intersect with our polygon.

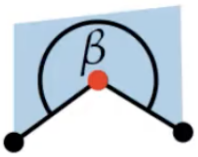
Example.

The green is convex and also y monoton where the point is only y monoton becaause each line we connect for two vertices with the same y cooardinate lies in the polygon and in the green this is true for each pair of vertices not neccessarilty with the same y cooardinate where in the pink not...

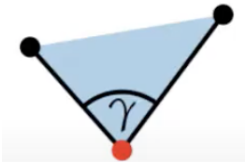
Idea: Classify vertices of given simple polygon P.

- Turn vertices
  - Vertical component of walking direction changes
- Start vertex: if $\alpha < 180$.

  -
- Split vertex: if $\beta > 180$.
  -



- End vertex: if $\gamma < 180$.
  -

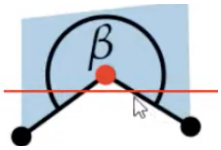


- Merge vertex: if $\delta > 180$.

–

note that when we have split vertex then P is not y monotonic because if we take two vertices with the same y and connect them as follows,

•



we can see that the line not lies in P (blue area) but in the white area so this type of vertices we want to avoid, also for merge vertices we can say that we get no y monotonic polygon hencefore, we can state a Lemma. In order to destroy this vertices we add diagonals for split and merge vertices so how?

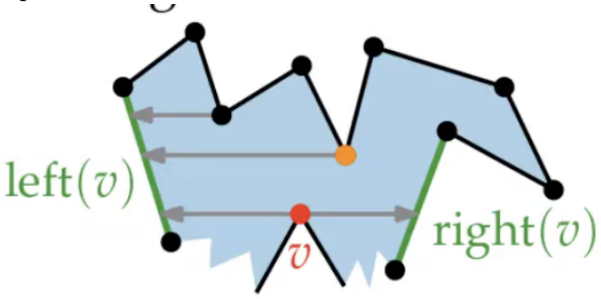
Lemma. *Let P be a simple polygon. Then P is y- monoton $\iff$ P has neither split vertices nor merge vertices.*

Idea: Add diagonals to "destroy" split and merge vertices.

Problem. Diagonals must not cross - each other - edges of P.

(1) Treating split vertices

•



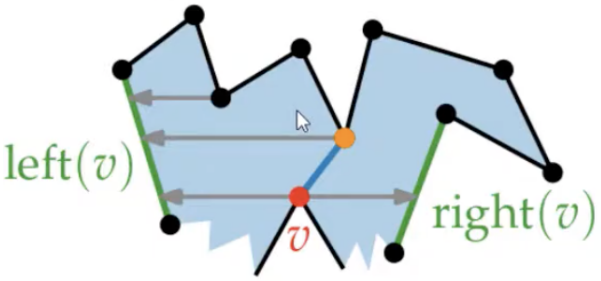
- first, we look to the left and to the right with horizontl line on v as on picture above, we can see two edges left, right emphasized in green, many other vertices may have the same left edge as v for example the orange vertices so from all vertices with left edge equal to $left(v)$ we choose the minimal y coordinate vertice that share $left(v)$ i.e.,

$$k = min\,\{\vec{x} : left(\vec{x}) = left(\vec{v}) \text{ and } y \text{ cooardinate of } x \text{ is minimal}\}$$

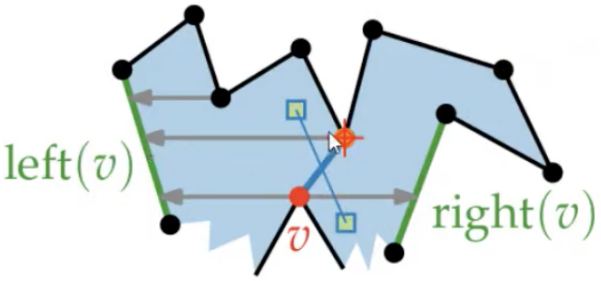
. note that k is unique because otherwise k is not the lowest candidate in the set

$$G = \{\vec{x} : left(\vec{x}) = left(\vec{v}) \text{ and } y \text{ cooardinate of } x \text{ is minimal}\}$$

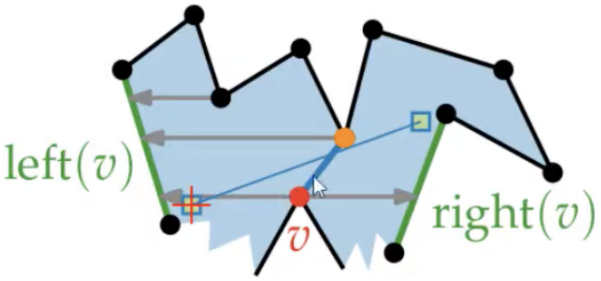
- Connect v to vertex w^* having minimum $y-$ coordinate among all vertices w above v (orange vertice) and with $left(w) = left(v)$.



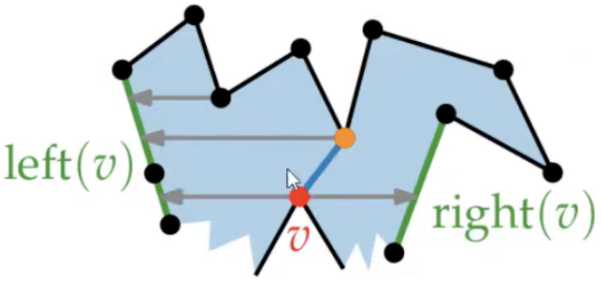
- now we can see that the edge connect them is the blue edge, however, why it can't be intersected with other diagonal? this is not trivial so the only way it can intersect something is a case where two vertices will be in different regions as follows,



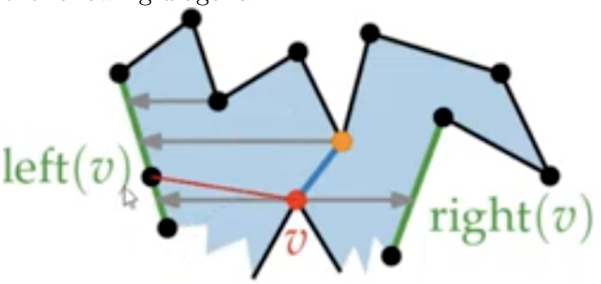
- this is not possible i.e., can't occur because now the edge in which the orange vertices w^* sees is the edge conntecting those two points and not $left(v)$ so w^* is not minimal in the set we defined which gives contradiction to the give that its minimal. (Simial case could occur if two green vertices placed as follow



- Now, what if $G = \emptyset$ is it possible? for example if we place a vertex on $left(v)$ as follow,



- observe that w^* doesn't see $left(v)$ anymore so the set $G = \emptyset$? not exactly, in fact its a qustion of defintion we can claim that the $G = \{\text{new vertex on } left(v)\}$ i.e., here we define that a vertex m lies on a boundary of polygon its left edge $left(m)$ is the edge going downwards and right is upwards, so in this new polygon the diagonal will be edge connects v and the vertices on the boundary i.e., on left v i.e., we get the following diagonal



- now he we find $left(v)$ in effcient way i.e., in $O(nlogn)$ complexity, A trivial approach is to walk on all the polygon vertices and check which vertices x have $left(v) = left(x)$ and take the minimal y cooadrdiante of them but for each of the vertices we will have linear time i.e., in total $O(n^2)$. A better way, is two use the sweep-line in each step we will have a active segment then we always have left edge active