Contents Lecture 5

- The divide and conquer algorithm design technique
- Analysing a divide and conquer algorithm: Mergesort
- Counting inversions
- Closest pair of points
- Convex hull

The divide and conquer algorithm design technique

- Suppose you have n items of input and the simplest technique to process it would be two nested for loops with a $\Theta(n^2)$ running time
- If n is small then this is fine
- With divide and conquer we instead aim at:
  - Divide in linear time the problem into two subproblems with $n/2$ items
  - Solve each subproblem
  - Combine the solutions to the subproblems in linear time into a solution for the n item problem
- The resulting running time becomes $\Theta(n \log n)$
- We will next study Mergesort

4 GHz modern CPU

n	n	$n \log n$	n^2	n^3	1.5^n	2^n	$n!$
10	2.5 ns	8.3 ns	25.0 ns	250.0 ns	14.4 ns	256.0 ns	907.2 μs
11	2.8 ns	9.5 ns	30.2 ns	332.8 ns	21.6 ns	512.0 ns	10.0 ms
12	3.0 ns	10.8 ns	36.0 ns	432.0 ns	32.4 ns	1.0 μs	119.8 ms
13	3.2 ns	12.0 ns	42.2 ns	549.2 ns	48.7 ns	2.0 μs	1.6 s
14	3.5 ns	13.3 ns	49.0 ns	686.0 ns	73.0 ns	4.1 μs	21.8 s
15	3.8 ns	14.7 ns	56.2 ns	843.8 ns	109.5 ns	8.2 μs	5 min
16	4.0 ns	16.0 ns	64.0 ns	1.0 μs	164.2 ns	16.4 μs	1 hour
17	4.2 ns	17.4 ns	72.2 ns	1.2 μs	246.3 ns	32.8 μs	1.0 days
18	4.5 ns	18.8 ns	81.0 ns	1.5 μs	369.5 ns	65.5 μs	18.5 days
19	4.8 ns	20.2 ns	90.2 ns	1.7 μs	554.2 ns	131.1 μs	352.0 days
20	5.0 ns	21.6 ns	100.0 ns	2.0 μs	831.3 ns	262.1 μs	19 years
30	7.5 ns	36.8 ns	225.0 ns	6.8 μs	47.9 μs	268.4 ms	10^{15} years
40	10.0 ns	53.2 ns	400.0 ns	16.0 μs	2.8 ms	5 min	10^{31} years
50	12.5 ns	70.5 ns	625.0 ns	31.2 μs	159.4 ms	3.3 days	10^{47} years
100	25.0 ns	166.1 ns	2.5 μs	250.0 μs	3 years	10^{13} years	10^{141} years
1000	250.0 ns	2.5 μs	250.0 μs	250.0 ms	10^{159} years	10^{284} years	huge
10^4	2.5 μs	33.2 μs	25.0 ms	4 min	huge	huge	huge
10^5	25.0 μs	415.2 μs	2.5 s	2.9 days	huge	huge	huge
10^6	250.0 μs	5.0 ms	4 min	8 years	huge	huge	huge
10^7	2.5 ms	58.1 ms	7 hour	10^4 years	huge	huge	huge
10^8	25.0 ms	664.4 ms	28.9 days	10^7 years	huge	huge	huge
10^9	250.0 ms	7.5 s	8 years	10^{10} years	huge	huge	huge

Mergesort

- Mergesort is a stable sort algorithm
- Running time $\Theta(n \log n)$
- See `mergesort.c` e.g. in the book

Recurrence relation

- Swedish differensekvation or rekursionsekvation
- A **recurrence relation** or just **recurrence** is a set of equalities or inequalities such as

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

- The value of $T(n)$ is expressed using smaller instances of itself and a boundary value.
- To analyze the running time of a divide and conquer algorithm, recurrences are very natural
- But we want to have an expression for $T(n)$ in **closed form**
- Closed form means an expression only involving functions and operations from a generally accepted set — i.e. ”common knowledge”.
- Closed form can also be called **explicit form**
- So our next goal is to rewrite $T(n)$ into closed form

Mergesort recurrence

- $T(n)$ = max comparisons to mergesort n items
- Mergesort recurrence:

$$T(n) \leq \begin{cases} 0, & n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & n > 1 \end{cases}$$

- This is a simplification as can be seen if compared with the source code, but it is sufficiently accurate.
- We ignore ceil and floor:

$$T(n) \leq \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

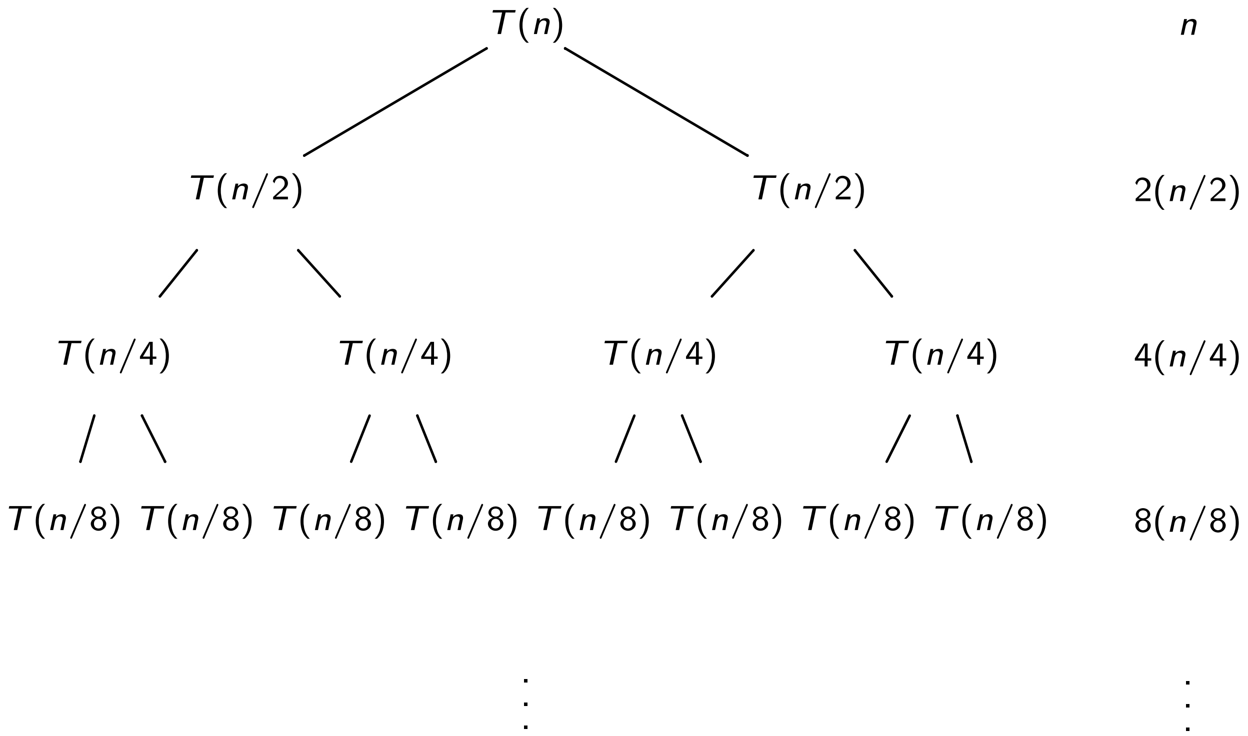
- We also assume n is a power of 2
- In the book it is shown that these simplifications do not affect our running time analysis

Rewriting a recurrence to closed form

- The easiest way to understand what the closed form is, may be to "expand" or "unroll" the recurrence and simply see what is happening
- Another way is to look at small inputs and try to guess the closed form
- When we have a guess which works for the small inputs, we then prove by induction that our guess is correct
- In both cases we prove our closed form by induction
- We will start with expanding $T(n)$

Expanding the recurrence and count

$$T(n) \leq \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$



- Assume n is power of 2
- $\log_2 n$ levels
- n comparisons per level
- In total $n \log n$ comparisons
- $T(n) = n \log n$

Proof by induction

Lemma

The recurrence

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

has the closed form $T(n) = n \log_2 n$*.*

Proof.

- Recall $\log ab = \log a + \log b$, so $\log_2 2n = \log_2 n + \log_2 2 = \log_2 n + 1$, and $\log_2 n = \log_2 2n - 1$
- Induction on n.
- Base case: $n = 1$: $T(1) = 1 \log_2 1 = 0$
- Induction hypothesis: assume $T(n) = n \log_2 n$
- $T(2n) = 2T(n) + 2n = 2n \log_2 n + 2n = 2n(\log_2 n + 1) = 2n(\log_2 2n - 1 + 1) = 2n \log 2n$

□

Remark about previous proof

- Normally we assume $S(i)$ is true and prove $S(i+1)$
- On previous slide we did not increment by one but rather doubled our variable
- We could have stated the lemma in terms of $S(i)$ and let $n = 2^i$
- Then we use induction on i and assume $S(i)$ and prove $S(i+1)$

Looking at small inputs

$$T(n) \leq \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

- Let us try out some small values:

$$\begin{array}{cccccccc} n & 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ T(n) & 0 & 2 & 8 & 24 & 64 & 160 & 384 \end{array}$$

- Can we identify a pattern?

$$\begin{array}{rrrrrrrrl} n & 1 & 2 & 4 & 8 & 16 & 32 & 64 & \\ T(n) & 0 & 2 & 8 & 24 & 64 & 160 & 384 & \\ T(n)/n & 0 & 1 & 2 & 3 & 4 & 5 & 6 & = \log_2(n) \end{array}$$

- So $T(n) = n \log_2 n$ is tempting to try to prove by induction, which we already know is true

The master theorem (MSc thesis by Dorothea Haken)

- There is a nice formula for finding $T(n)$ for many recursive algorithms:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= aT(n/b) + n^s. \end{aligned}$$

- There are three closed form solutions (for details, see the book):

$$T(n) = \begin{cases} O(n^s) & \text{if} \quad s > \log_b a \\ \\ O(n^s \log n) & \text{if} \quad s = \log_b a \\ \\ O(n^{\log_b a}) & \text{if} \quad s < \log_b a. \end{cases}$$

- $T(n) = 2T(n/2) + n$. With $a = b = 2$ and $s = 1$, we have $\log_b a = \log_2 2 = 1 = s$, so $T(n) = O(n \log n)$.
- $T(n) = 2T(n/2) + \sqrt{n}$. With $a = b = 2$ and $s = 0.5$, we have $\log_b a = \log_2 2 = 1 > s$, so $T(n) = O(n)$.
- $T(n) = 4T(n/3) + n^2$. We have $\log_b a = \log_3 4 = \frac{\log_{10} 4}{\log_{10} 3} = 1.26 < s = 2$, so $T(n) = O(n^2)$.

Finding people with similar tastes

- Consider a category such as text editor, programming language, preferred tab width, or the 22 Mozart operas
- To compare how similar tastes within a category three people have, they can rank a list of say 5 operas A-E

Tintin:	A	D	C	E	B
Captain Haddock:	A	C	B	D	E
Bianca Castafiolen:	A	B	D	C	E

- All agree opera A is best
- Who have most similar tastes?

Inversions

Tintin:	A	D	C	E	B
Captain Haddock:	A	C	B	D	E
Bianca Castafiolen:	A	B	D	C	E
- We have 5 positions in each list
- Start with Tintin's list and label each item $1, 2, \ldots, 5$:

Tintin:	A	D	C	E	B
Tintin:	1	2	3	4	5
- Then we put these labels according to Captain Haddock's ranking:

Captain Haddock:	1	3	5	2	4
	a_1	a_2	a_3	a_4	a_5
- i and j are **inverted** if $i < j$ and $a_i > a_j$
- Inversions: (3,2), (5,2), and (5,4)
- The fewer inversions, the more similar tastes

Counting inversions

```
for (c = i = 0; i < n; i += 1)
        for (j = i+1; j < n; j += 1)
                if (a[i] > a[j])
                        c += 1;

printf("%d inversions\n", c);
```

- Running time is $O(n^2)$
- How can we use divide and conquer to achieve $O(n \log n)$?
  1 3 | 5 2 4
- Count inversions in left part
- Count inversions in right part
- Somehow combine these parts and add number of inversions...???

What can we do to simplify the problem?

- 1 3 | 5 2 4
- Assume you know there are no inversions in the left part and two in the right part
- It is OK to "destroy" the array, such as sorting it, if that helps...
- If modifying the array is forbidden, we can always make a copy and work with the copy instead
- Copying the array is fine since that is faster than $O(n \log n)$
- Copying the array is $O(n)$ but memory allocation can be costly so don't do it too much
- For Mergesort, it is non-trivial to not use a second array

Sorting the array

- By subarray is meant the part our recursive subproblem is going to work with
- Sorting the subarray **after** counting the inversions may help
- $1 \quad 3 \mid 5 \quad 2 \quad 4$
- After having counted in the subarrays we have: $1 \quad 3 \mid 2 \quad 4 \quad 5$
- Combining two sorted parts can be done in linear time as in Mergesort
  $$\begin{array}{c|cccc||cc} 3 & 2 & 4 & 5 & 1 & \\ 3 & & 4 & 5 & 1 & 2 \end{array}$$
  The 2 was inverted with each remaining in left part — only the 3 in this example so one inversion is counted when the parts are combined
  $$\begin{array}{|cc||ccccc} 4 & 5 & 1 & 2 & 3 & & \\ & 5 & 1 & 2 & 3 & 4 & \\ & & 1 & 2 & 3 & 4 & 5 \end{array}$$
- In total 3 inversions

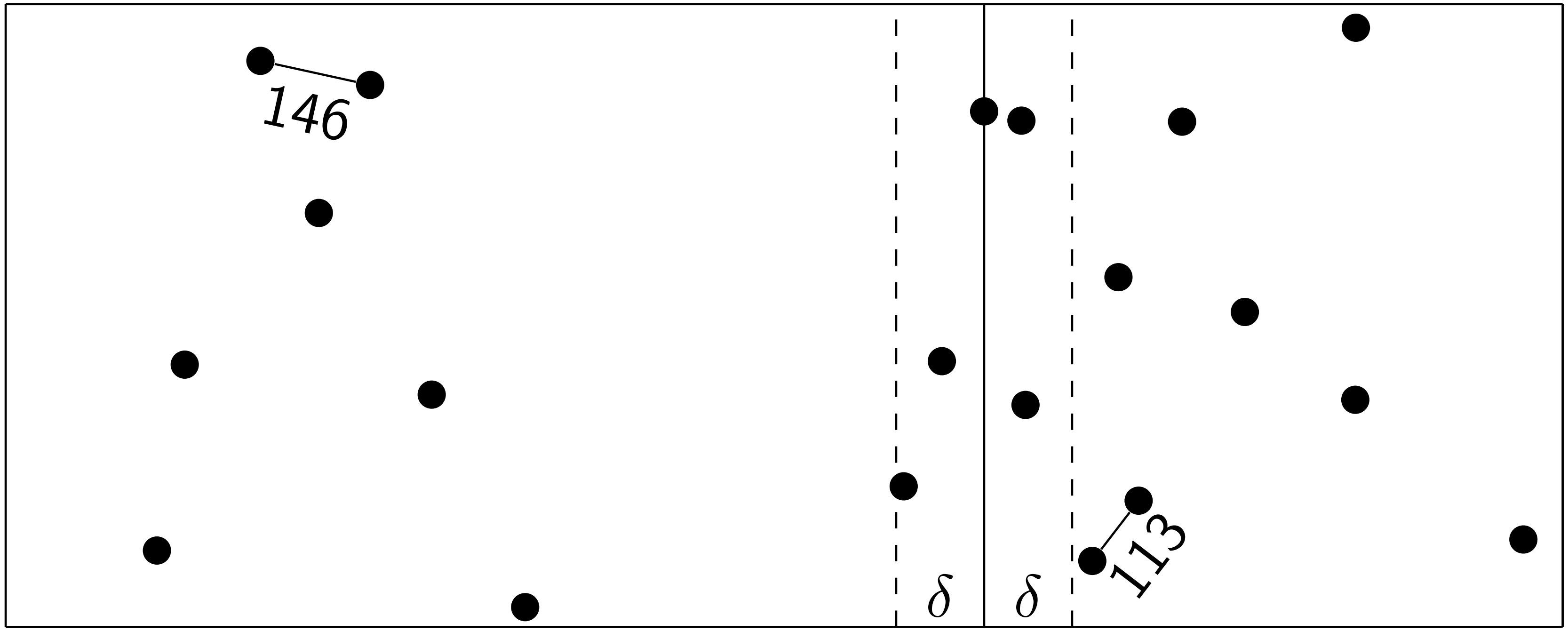
Implementing the $n \log n$ algorithm

- As always: first make a simple reference implementation that can be used to verify the correctness of a faster implementation
- In this case the n^2 algorithm is ideal if used with small inputs

Closest points in a plane

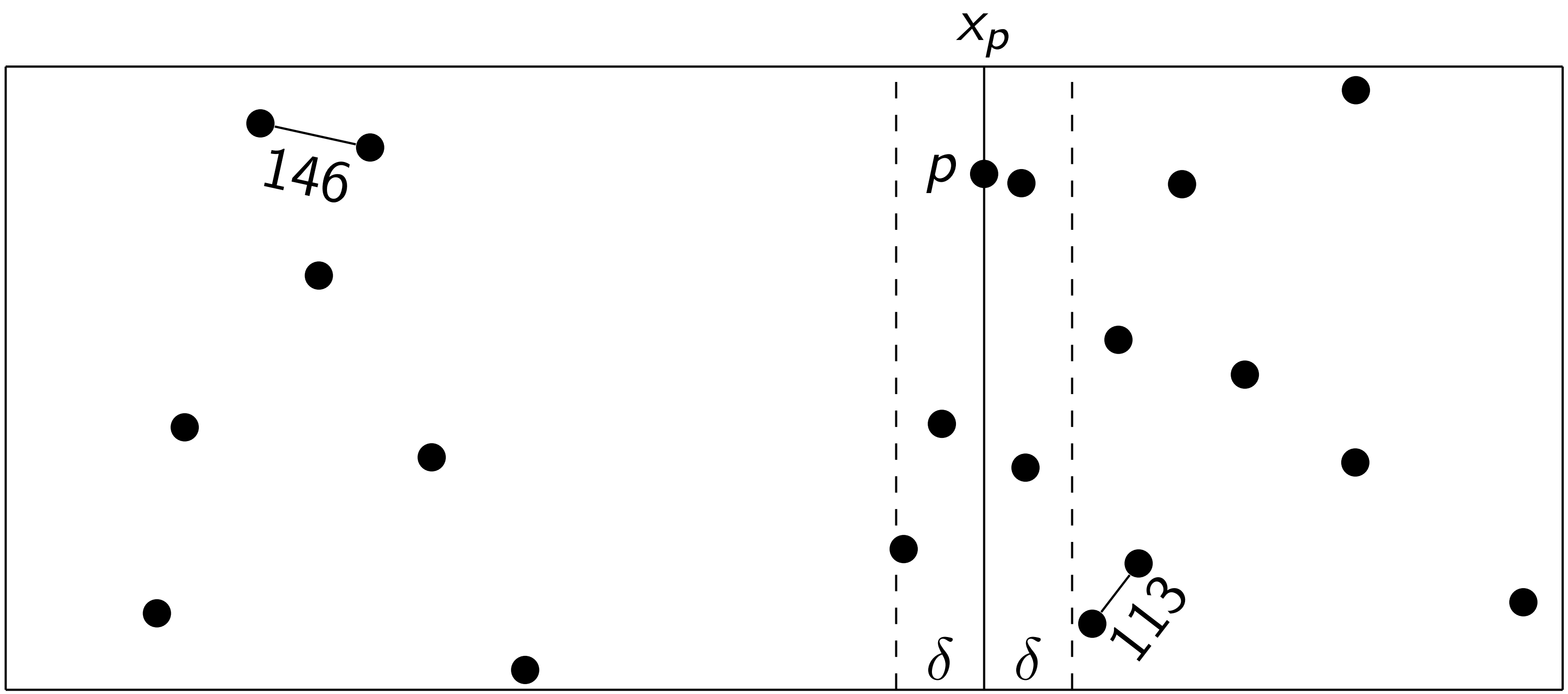
- This field is called **computational geometry**
- Consider n points (x_i, y_i) in a plane
- We want to find which points are closest
- Comparing all points with each other in an n^2 algorithm is simple
- But comparing points "obviously" far from each other is a waste
- How can divide and conquer be used to find an $n \log n$ algorithm?
- We cut the plane in two halves and find closest points in each half
- We have then three categories of point pairs which can be closest:
  1. Point pairs in the left half
  2. Point pairs in the right half
  3. Point pairs with one point in the left and the other in the right half
- Can we find close points from the last category in linear time???

An example



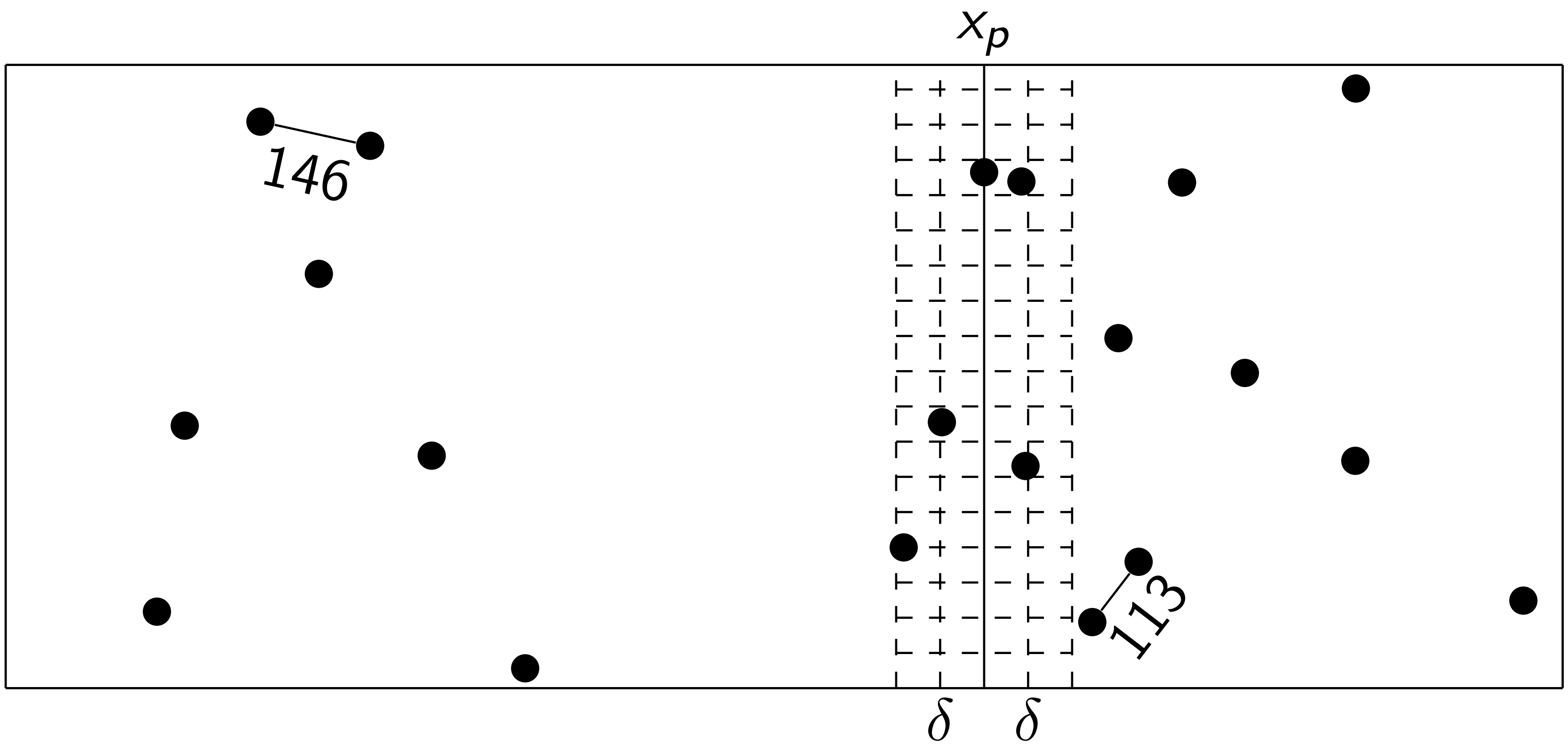
- We cut the plane in two halves with 10 points in each half
- We compute the nearest points in each half
- $\delta = \min(146, 113)$
- We only have to consider points within δ from the vertical line
- If there are none, then δ is the answer
- If there are, then they must be checked with points from the other side which also must be within δ from the vertical line, of course

Combining



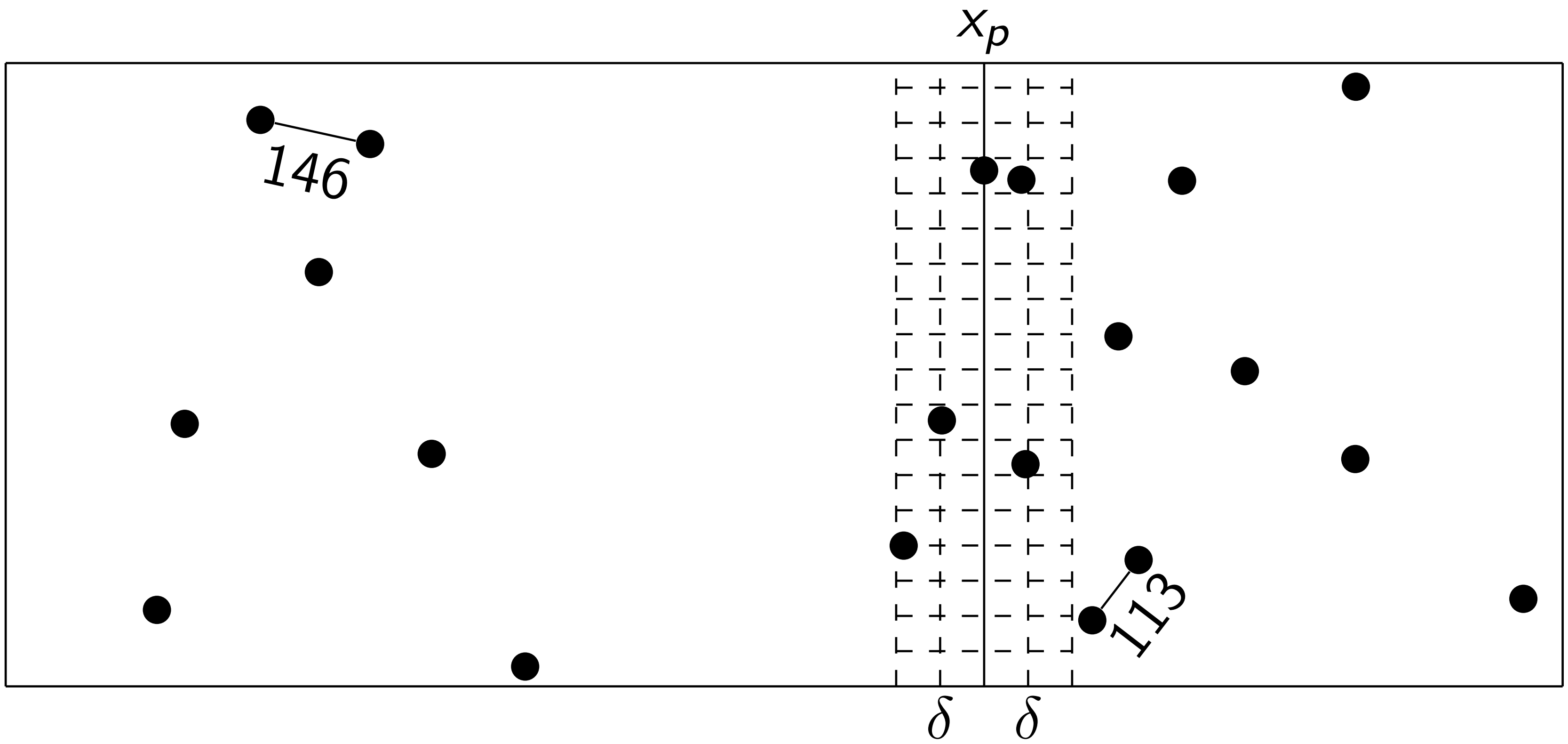
- The point p on the vertical line x_p belongs to the left half but there could also be points in the right half with the same x-coordinate
- Let the set S consist of all points with a distance within δ from the line x_p, (5 points here)
- Clearly it is sufficient to compare only points q and r from S such that p comes from the left half and q from the right part

Combining



- Each dashed box has a side of $\delta/2$
- How many points can each such box contain at most?
- The diagonal of a dashed box is $\sqrt{2} \times \delta/2 < \delta$
- With two points in a dashed box, their distance would be less than δ so at most one point

Combining

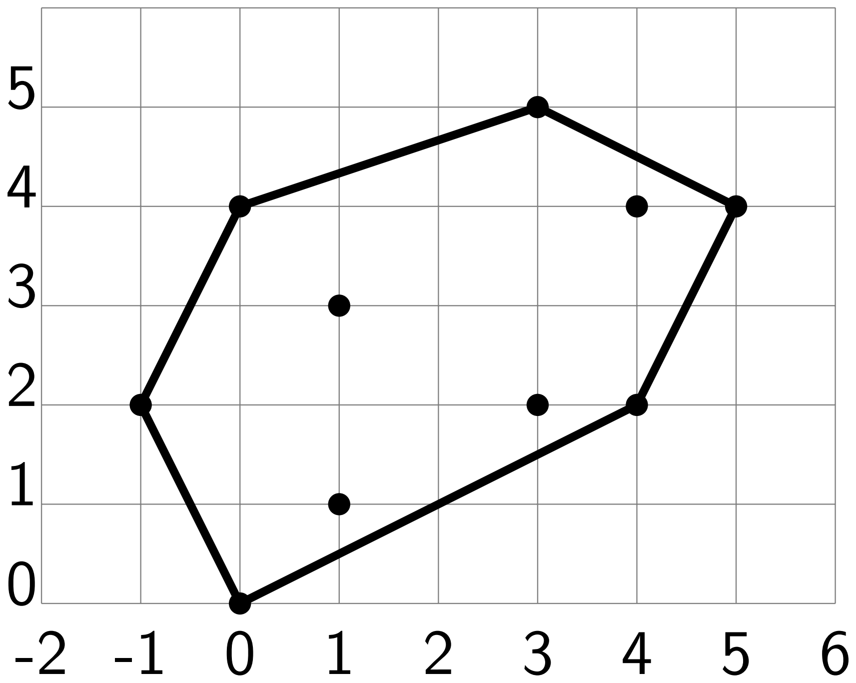


- With at most one point per dashed box, we can do as follows.
- Let S be sorted on y-coordinates
- Each point $p \in S$ is inspected at a time.
- The distances from p to each of the next six points on the other side in S (according to y-coordinates) are checked to see if it less than the shortest distance found so far

Algorithm outline

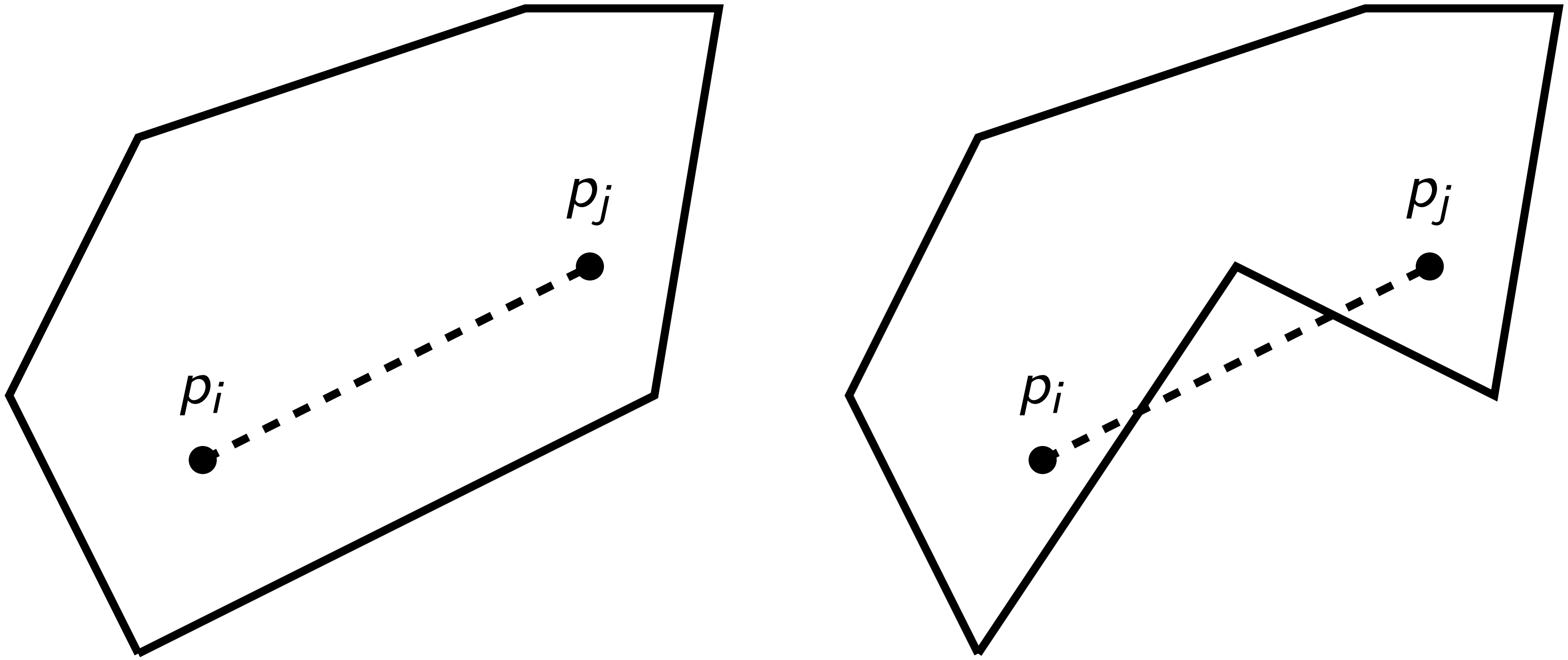
- What do we need for this?
- Input is a set of n points P
- We produce two sorted arrays P_x and P_y before starting our recursion
- We divide P_x into two arrays L_x and R_x (left and right)
- We divide P_y into two arrays L_y and R_y
- We solve the two subproblems $(L_x, L_y, n/2)$ and $(R_x, R_y, n/2)$
- Then we compute δ as the minimum from these subproblems
- Then we create the set S_y from P_y
- All dividing and combining can be done in linear time, so we solve this in $\Theta(n \log n)$ time

A set of points P and its convex hull $\mathrm{CH(P)}$

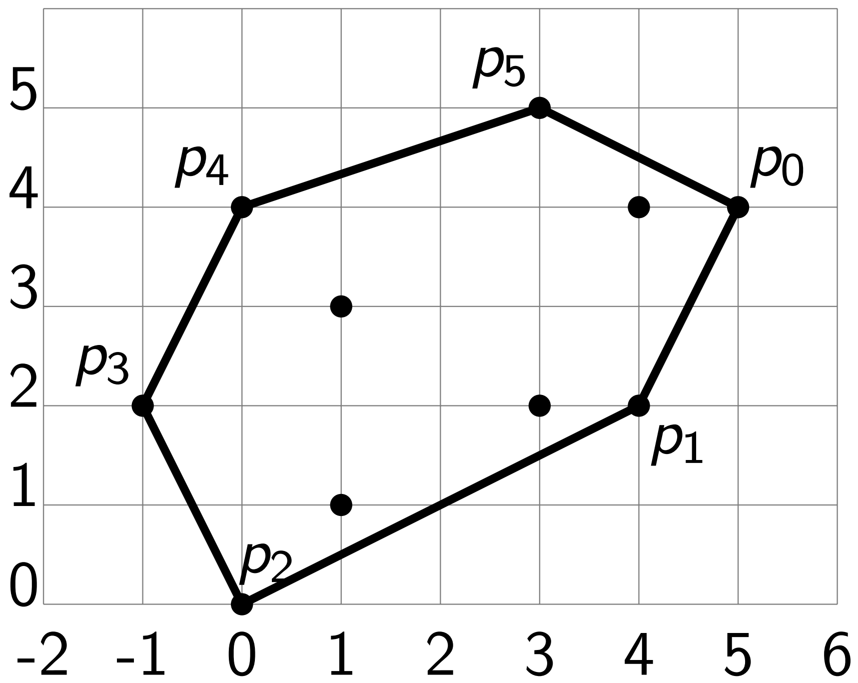


- The convex hull should have a minimal number of points.
- For example a point at $(2, 1)$ would not be in the convex hull.

A convex and a non-convex region



Clockwise order of $\mathrm{CH(P)}$

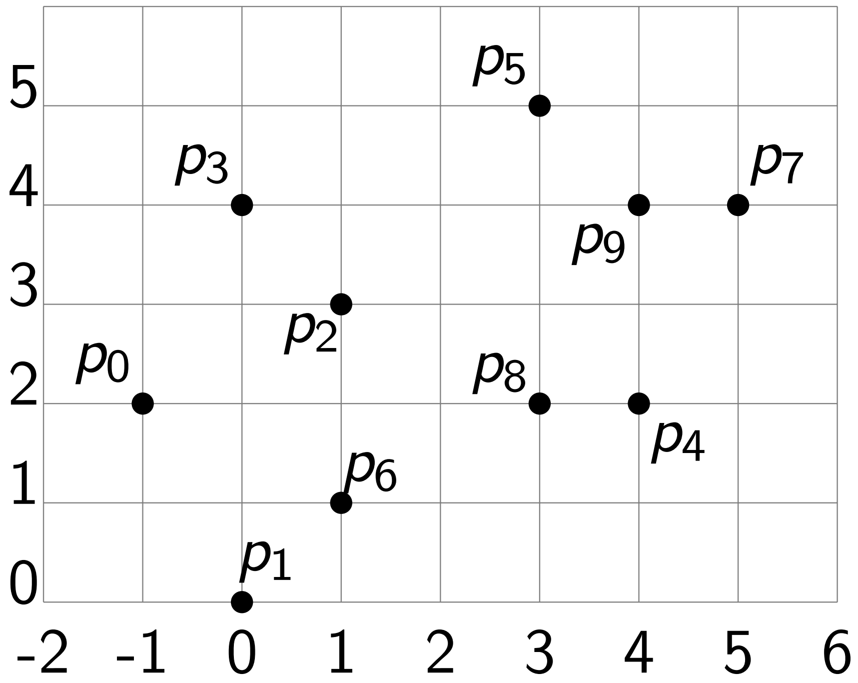


- Not necessary to select the rightmost point as p_0 in general
- In Lab 4 (2023) we did that however, since that is what the divide-and-conquer algorithm does.

Three algorithms for computing the convex hull

- Jarvis march, or gift wrapping
- Graham scan
- Preparata-Hong
- Always a good approach to implement a non-trivial algorithm: start with something simpler and use the simple (and hopefully correct) as a reference

Jarvis march

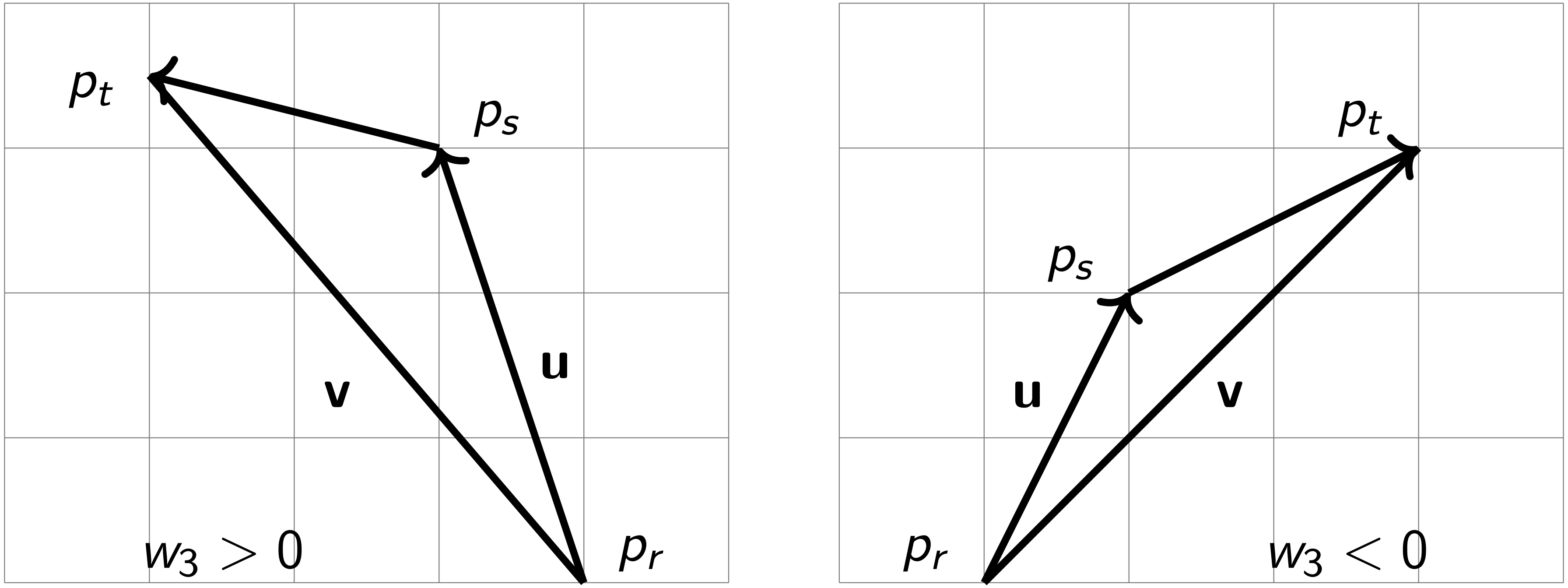


- Start at any point known to be in CH such as the leftmost: p_0, $i \leftarrow 0$
- Select the next point p_j as the one through which we only make left turns when going from p_i through p_j to p_k for every other point p_k
- Right turn with p_0, p_2, p_1 so don't take p_2 next
- Always left turn with p_0, p_1, p_k so take p_1 as next
- Continue until back at p_0

Review of vector products

- Also called cross product (vektorprodukt or kryssprodukt)
- Given two vectors in $\mathbb{R}^3$, $\mathbf{u}$ and $\mathbf{v}$, the vector product, $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ is a vector with the following properties:
  1. $\mathbf{w}$ is perpendicular to both $\mathbf{u}$ and $\mathbf{v}$, i.e., the dot products $\mathbf{w} \cdot \mathbf{u}$ and $\mathbf{w} \cdot \mathbf{v}$ are both zero,
  2. $|\mathbf{w}| = |\mathbf{u}||\mathbf{v}| \sin\theta$ where θ is the angle between $\mathbf{u}$ and $\mathbf{v}$,
  3. $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ are positively oriented, i.e. according to the right-hand rule.
- $\mathbf{u} = u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}$ and $\mathbf{v} = v_1\mathbf{e_1} + v_2\mathbf{e_2} + v_3\mathbf{e_3}$
- $\mathbf{w} = (u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}) \times (v_1\mathbf{e_1} + v_2\mathbf{e_2} + v_3\mathbf{e_3}) = (u_2 v_3 - u_3 v_2)\mathbf{e_1} + (u_3 v_1 - u_1 v_3)\mathbf{e_2} + (u_1 v_2 - u_2 v_1)\mathbf{e_3} = w_1\mathbf{e_1} + w_2\mathbf{e_2} + w_3\mathbf{e_3}$.
- Since our points are in $\mathbb{R}^2$, their $\mathbf{e_3}$ coordinates are zero and so w_1 and w_2 also become zero
- $\mathbf{w} = w_3\mathbf{e_3} = (u_1 v_2 - u_2 v_1)\mathbf{e_3}$.

Left or right direction at p_s determined with $\mathbf{w} = \mathbf{u} \times \mathbf{v}$



- To find the direction from p_r through p_s to p_t we let
- $\mathbf{u} = \overrightarrow{p_r p_s}$,
- $\mathbf{v} = \overrightarrow{p_r p_t}$, and
- $\mathbf{w} = \mathbf{u} \times \mathbf{v}$. If $w_3 > 0$ it is a left turn, if $w_3 < 0$ it is a right turn, and otherwise the three points are on the same line.

Jarvis march

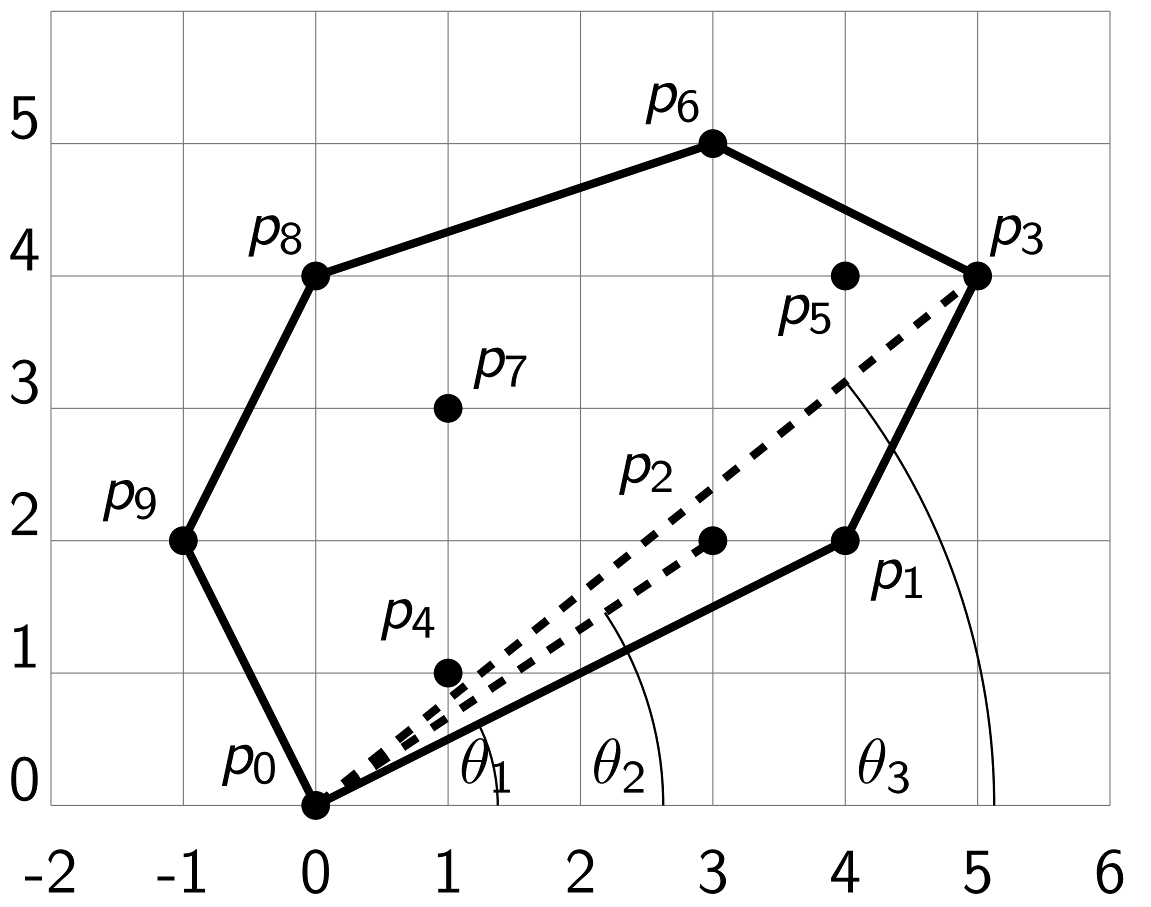
function *jarvis_march* (p)
begin
    $n \leftarrow |p|$
    $i \leftarrow$ index of point in p with minimum x coordinate
    swap p_0 and p_i
    $r \leftarrow 0$
    while (1) {
        $s \leftarrow (r + 1) \mod n$
        for $t \leftarrow 0$; $t < n$; $t \leftarrow t + 1$ {
            if $s = t$ **then**
                continue
            $\mathbf{u} = \overrightarrow{p_r p_s}$
            $\mathbf{v} = \overrightarrow{p_r p_t}$
            $\mathbf{w} = \mathbf{u} \times \mathbf{v}$
            $(w_1, w_2, w_3) \leftarrow \mathbf{w}$
            // right turn or p_s between p_r and p_t on a line?
            if $w_3 < 0$ **or** $w_3 = 0$ **and** $|\mathbf{v}|^2 > |\mathbf{u}|^2$ **then**
                $s \leftarrow t$
        }
        $r \leftarrow r + 1$
        if $s = 0$ **then**
            break
        swap p_s and p_r
    }
    return r // number of points in $CH(P)$
end

Time complexity of Jarvis march

- Time complexity is $O(n \cdot h)$ with h points in the convex hull.
- We increment r once for every point in the convex hull.
- Since some convex regions consist of all their points, the worst case is $O(n^2)$
- For example a regular polygon ("circle" but not exactly round...)
- Regelbunden polygon
- We will next look at Graham scan which is $O(n \log n)$ due to all points must be sorted first

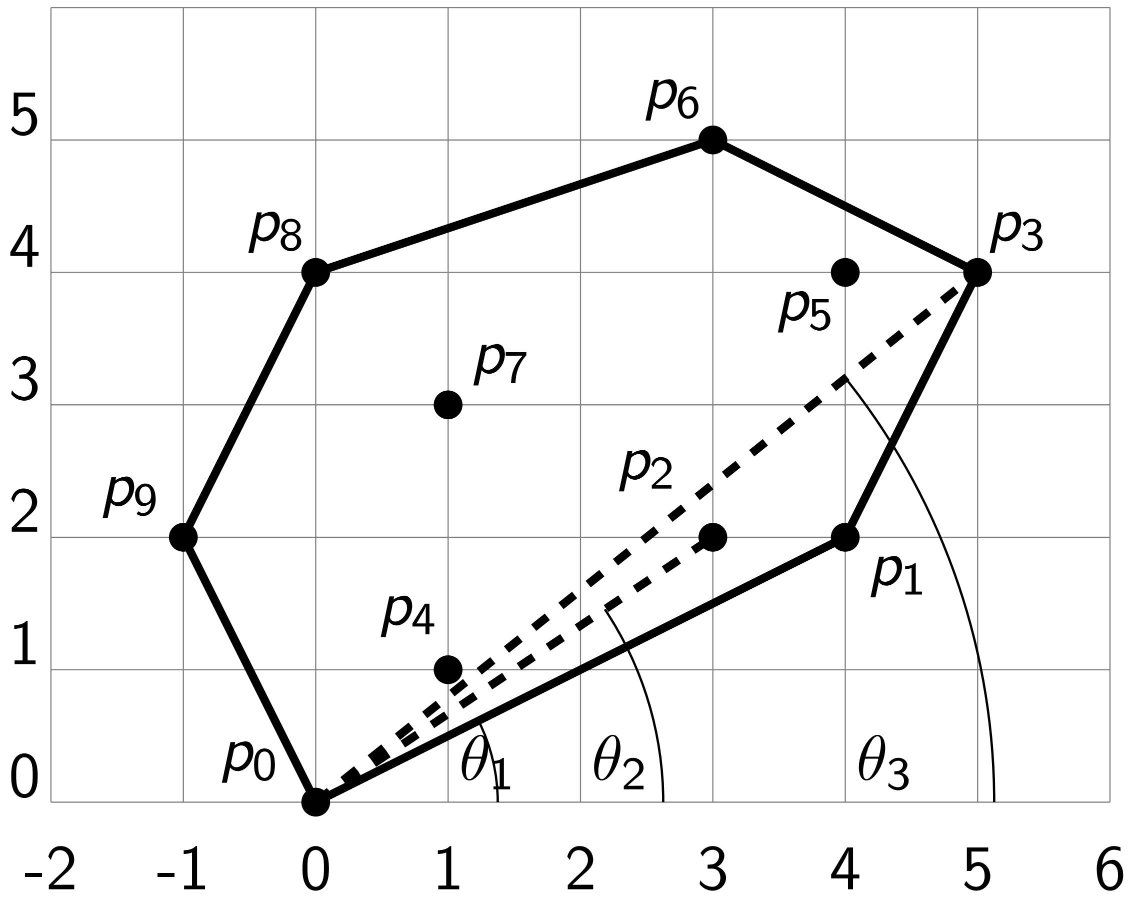
Graham scan

- First a point p_0 with minimal y-coordinate is made a new origo.
- One can make an angle between the x-axis, p_0, and every other point
- The points are sorted by these angles θ_i, $1 \leq i \leq p_{n-1}$



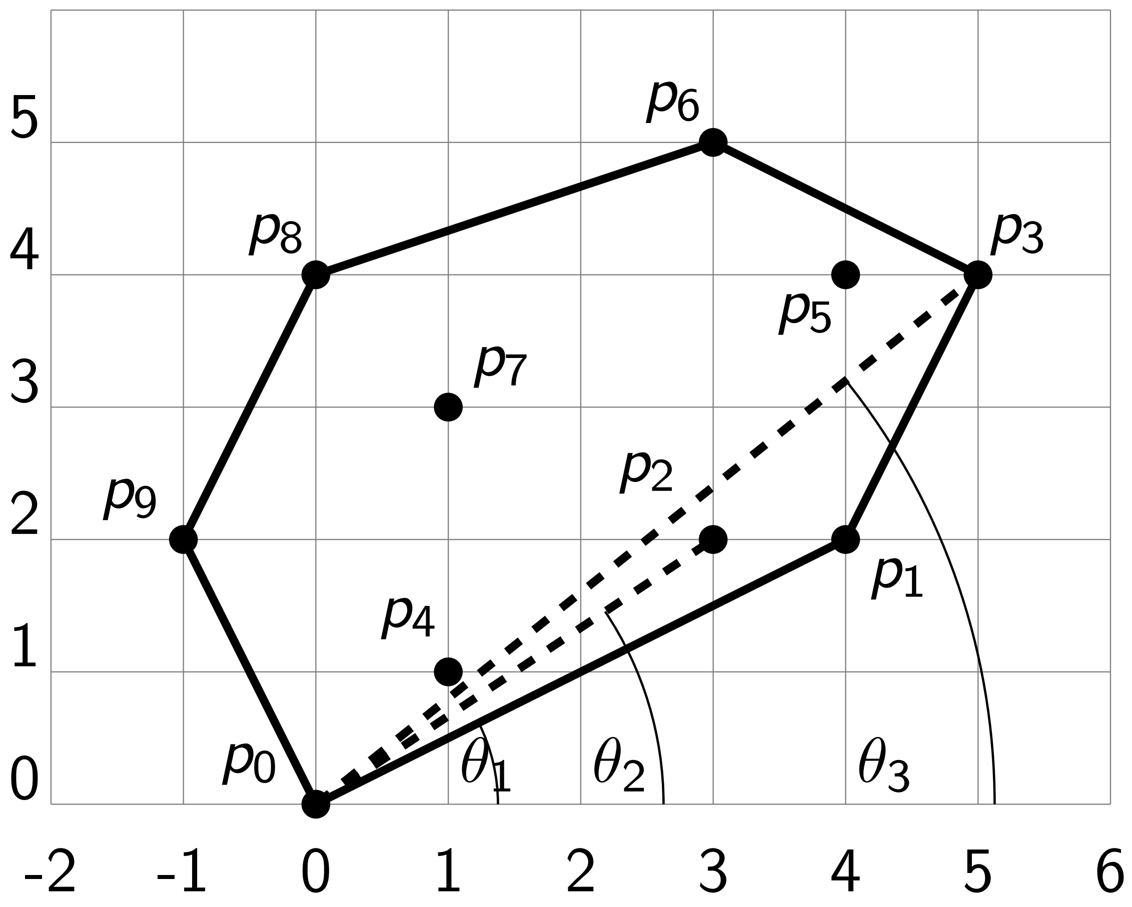
Including points in the convex hull

- The points p_0, p_1, p_2 are pushed to a stack with p_2 at the top
- Call the point at the top of the stack p_s (initially p_2)
- Call the point just below the top of the stack p_r (initially p_1)
- Call the "next point" p_t (initially p_3)



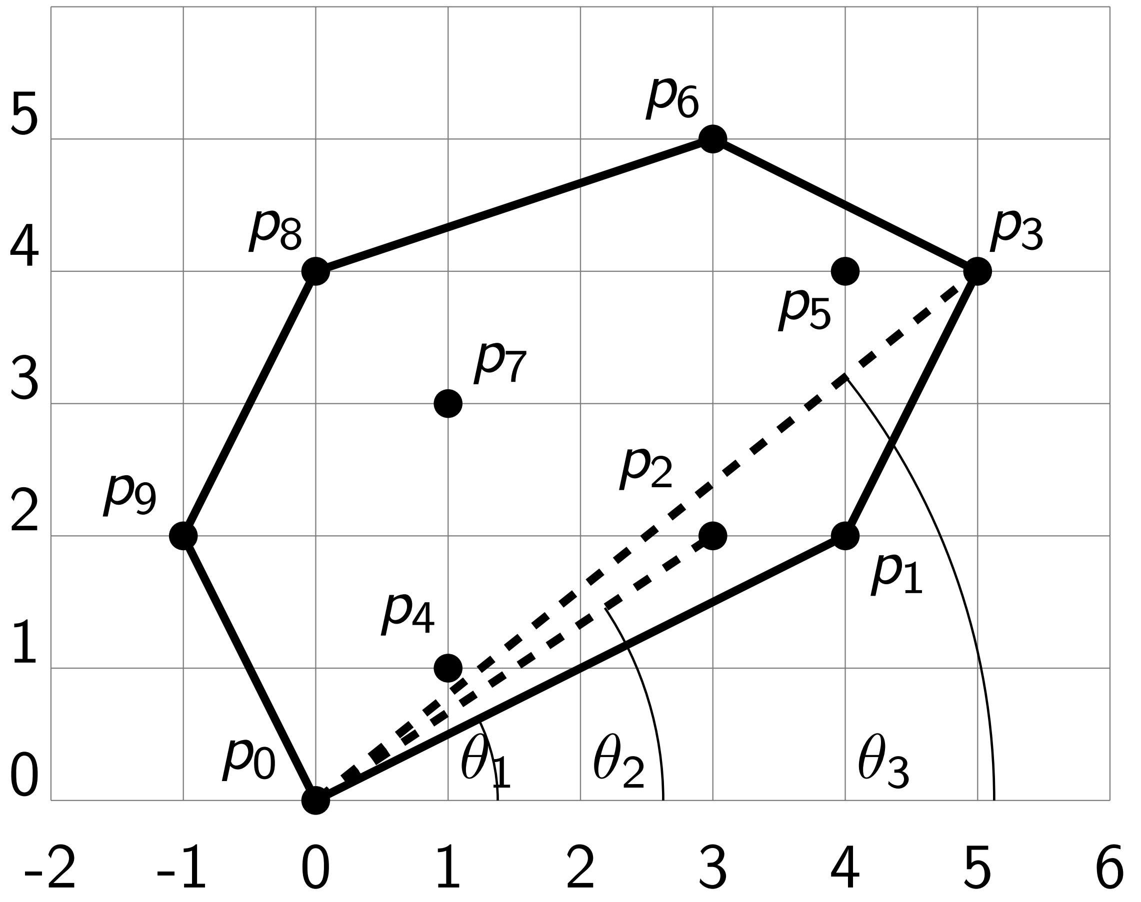
Excluding points from the convex hull

- Consider going from p_r, through p_s and to p_t initially p_1, p_2, p_3
- If the direction through p_s is straight or right, then p_s is not in CH
- In that case it is popped from the stack



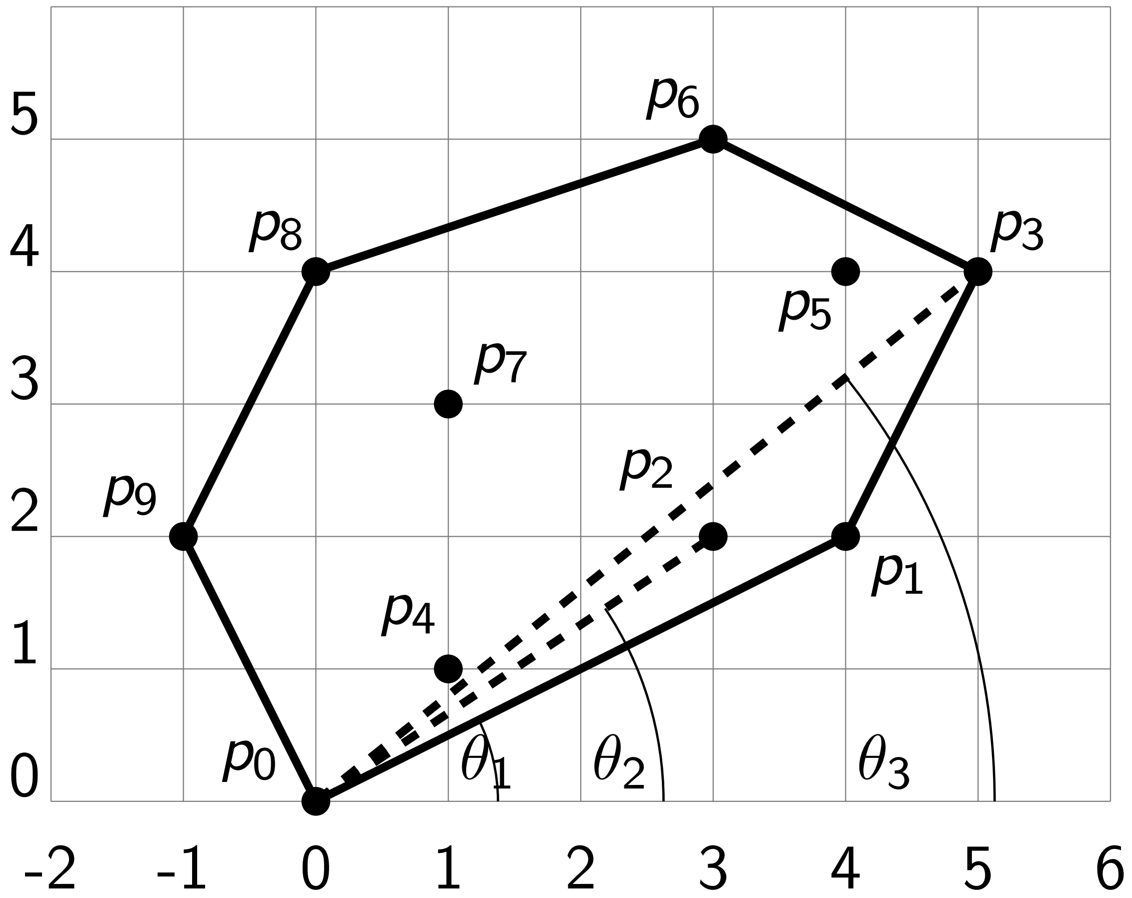
Excluding points from the convex hull

- Consider going from p_r, through p_s and to p_t
- After p_2 was popped, p_1 becomes new p_s and p_0 new p_r
- Any more non-left turns results in a pop
- Then p_t is pushed so $p_r = p_1$ and $p_s = p_3$



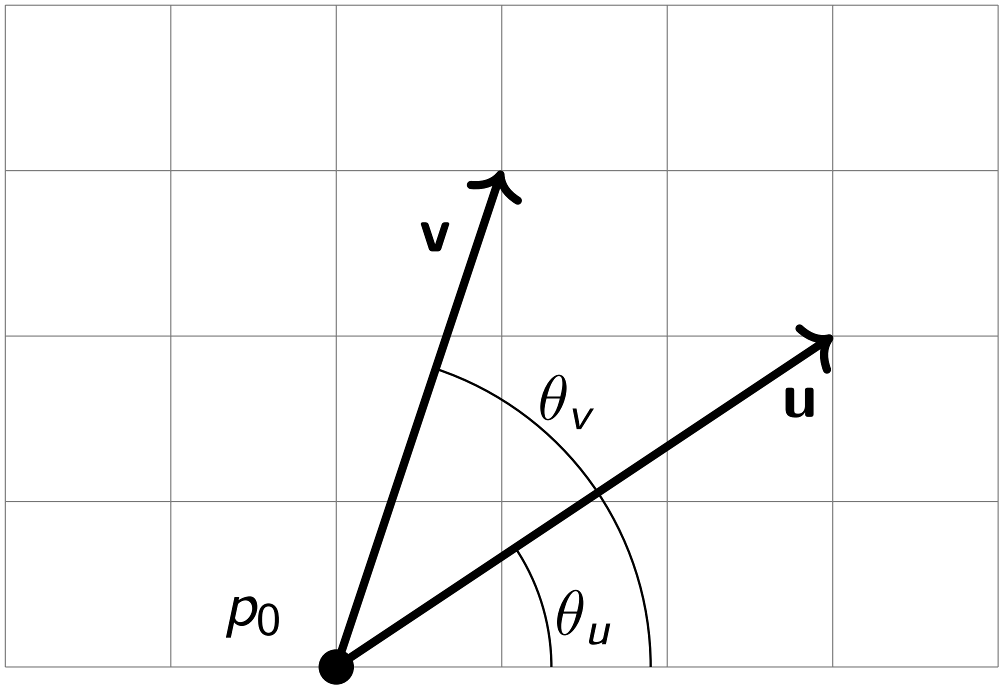
Excluding points from the convex hull

- $p_r = p_1$ and $p_s = p_3$
- $p_t = p_4$ with a left turn from p_r and p_s so p_4 is pushed
- After that p_5 will cause p_4 being popped
- In the end, all points remaining on the stack are the convex hull



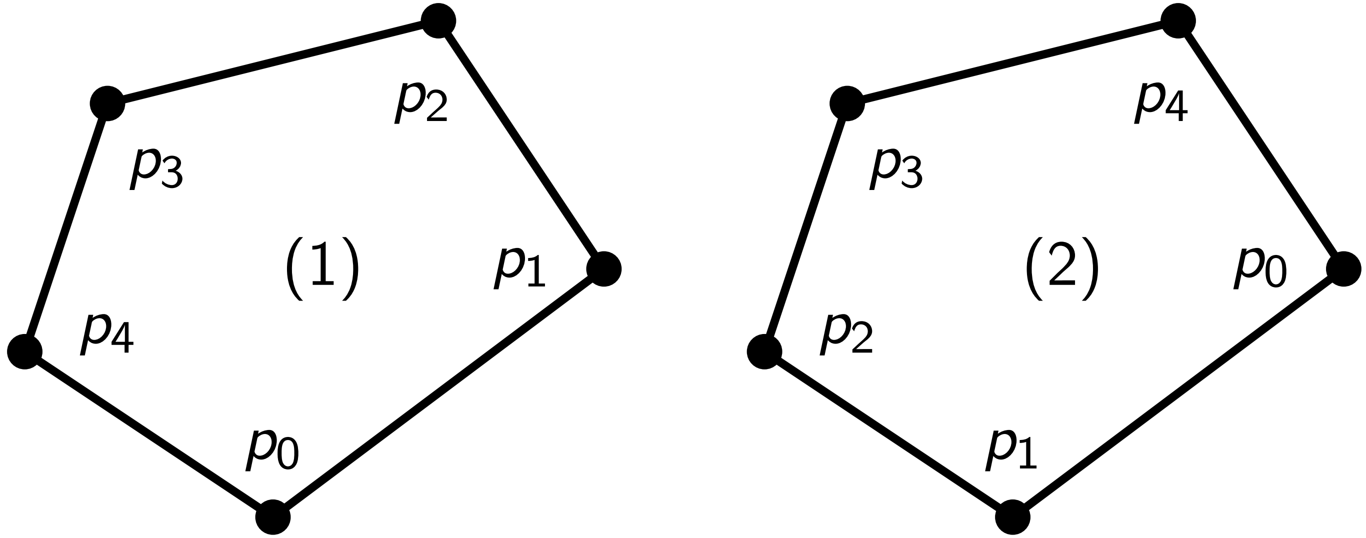
Relative sizes of θ is sufficient

- Compare angles with $\mathbf{u} \times \mathbf{v}$
- If $\theta_u = \theta_v$ then how should they be ordered?
- We want the point nearest origo on the stack first so the other can pop it



Output for Lab 4 2023

- The `check_solution.sh` script expects output as in (2)
- The reason is that Preparata-Hong produces that output.

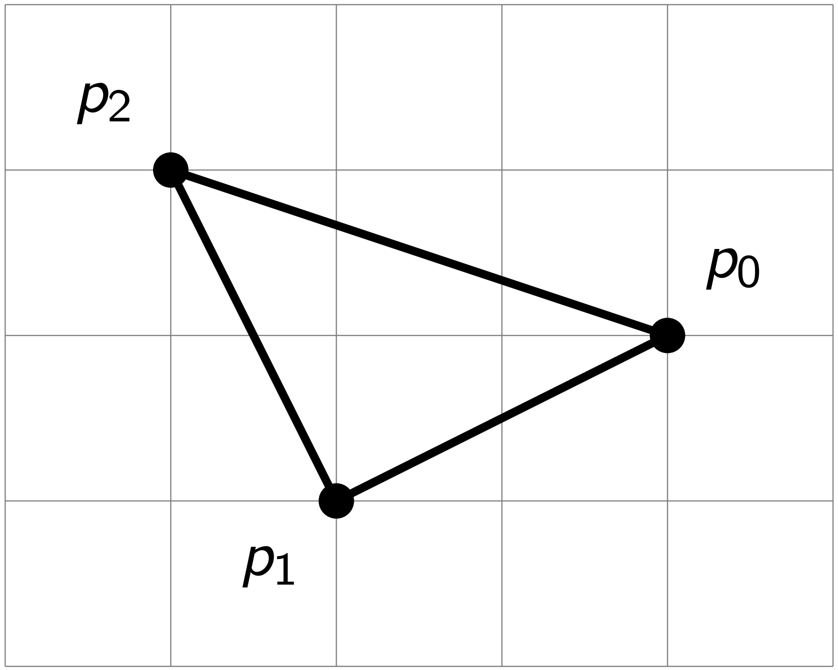


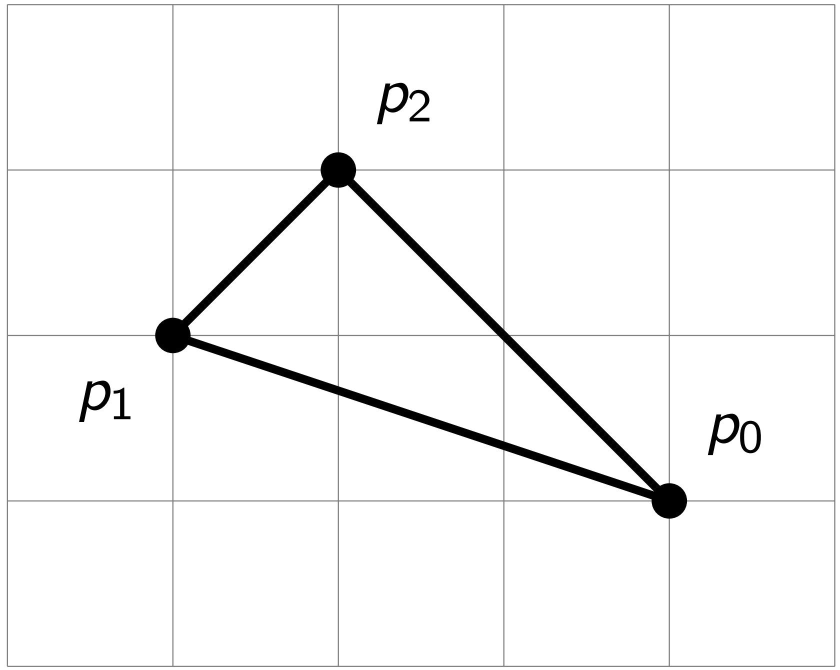
Graham scan

```
function graham_scan(p)
begin
    // p is an array with n points.
    n ← |p|
    i ← index of point in p with minimum y coordinate
    swap p_0 and p_i
    t = p_0
    subtract the coordinates of t from every point
    sort elements 1..n − 1 of p by θ_i
    h ← new stack
    push(h, p_0)
    push(h, p_1)
    push(h, p_2)
    for (k ← 3; k < n; k = k + 1) {
        // p_s is the top of h
        // p_r is below p_s on h
        p_t ← p_k
        while direction(next_top(h), top(h), p_t) is not left
            pop(h)
        push(h, p_t)
    }
    add the coordinates of t to every point
    n ← number of points on the stack
    copy the points in h to p, and deallocate h
    return n
end
```

Preparata-Hong output

- The sequence that is the convex hull
- The number of points in the convex hull
- The index of the leftmost point





Preparata-Hong algorithm

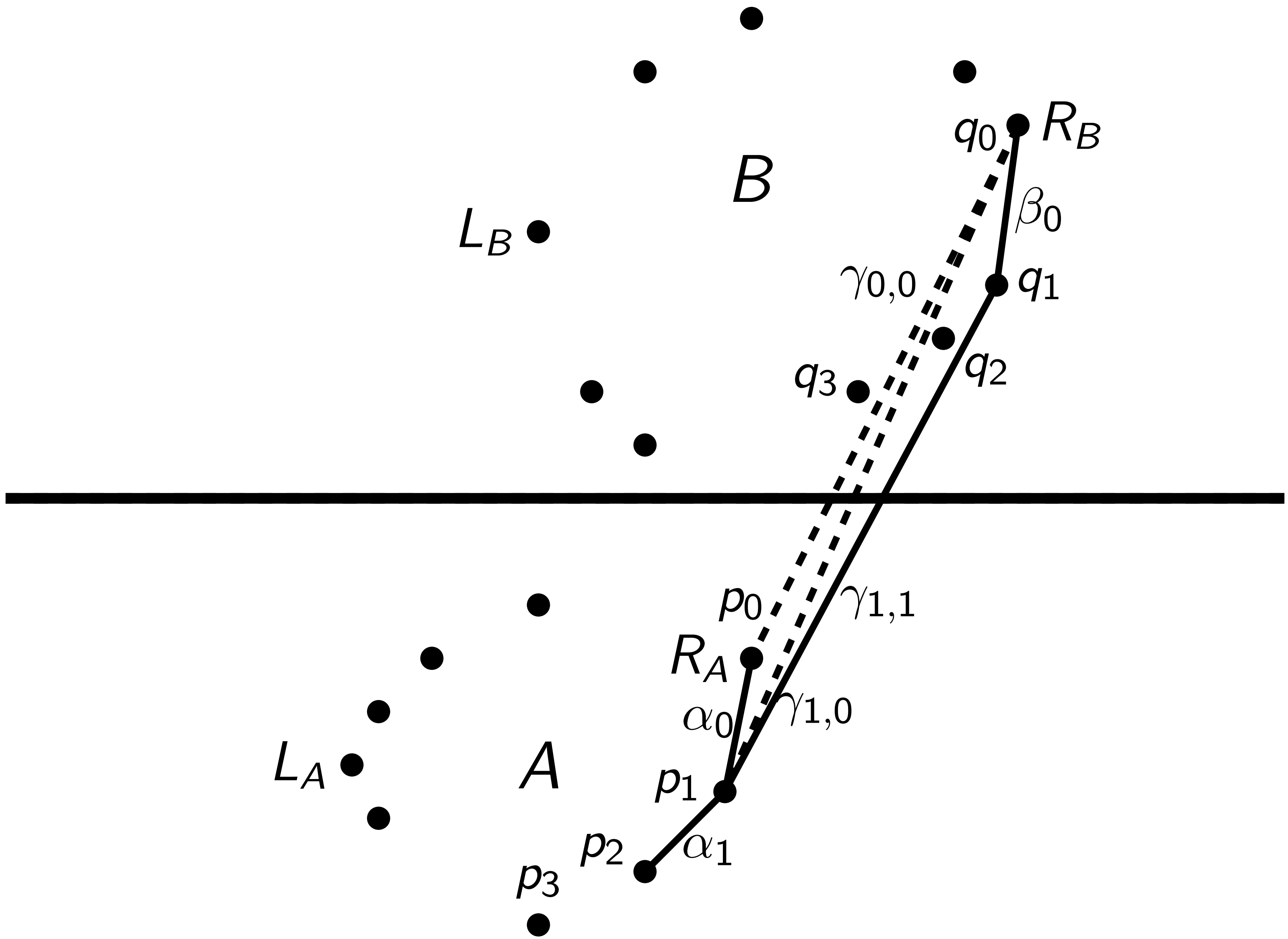
- It relies on lines expressed as $y = k \cdot x + m$
- Sort all points n in order of increasing y-coordinates
- With $n \leq 3$ solve directly and return
- Divide in two approximately equal parts A and B
- All points in A must have a y-coordinate lower than any in B
- Find $CH(A)$ and $CH(B)$
- Merge $CH(A)$ and $CH(B)$ which is simplified by knowing that they are in clockwise order

α and β

- n_a points in lower convex hull
- n_b points in upper convex hull
- The lower points are called $p_0..p_{n_a-1}$
- The upper points are called $q_0..q_{n_b-1}$
- The inner points from A and B are not needed for anything
- $y = k \cdot x + m$
- We need to compute the k-values from p_0 to p_1, from p_1 to p_2 etc
- The k value of the line segment from p_i to $p_{i+1 \mod n_a}$ is called α_i
- The k value of the line segment from q_i to $q_{i+1 \mod n_b}$ is called β_i
- For merging $CH(A)$ and $CH(B)$ we start with computing α and β

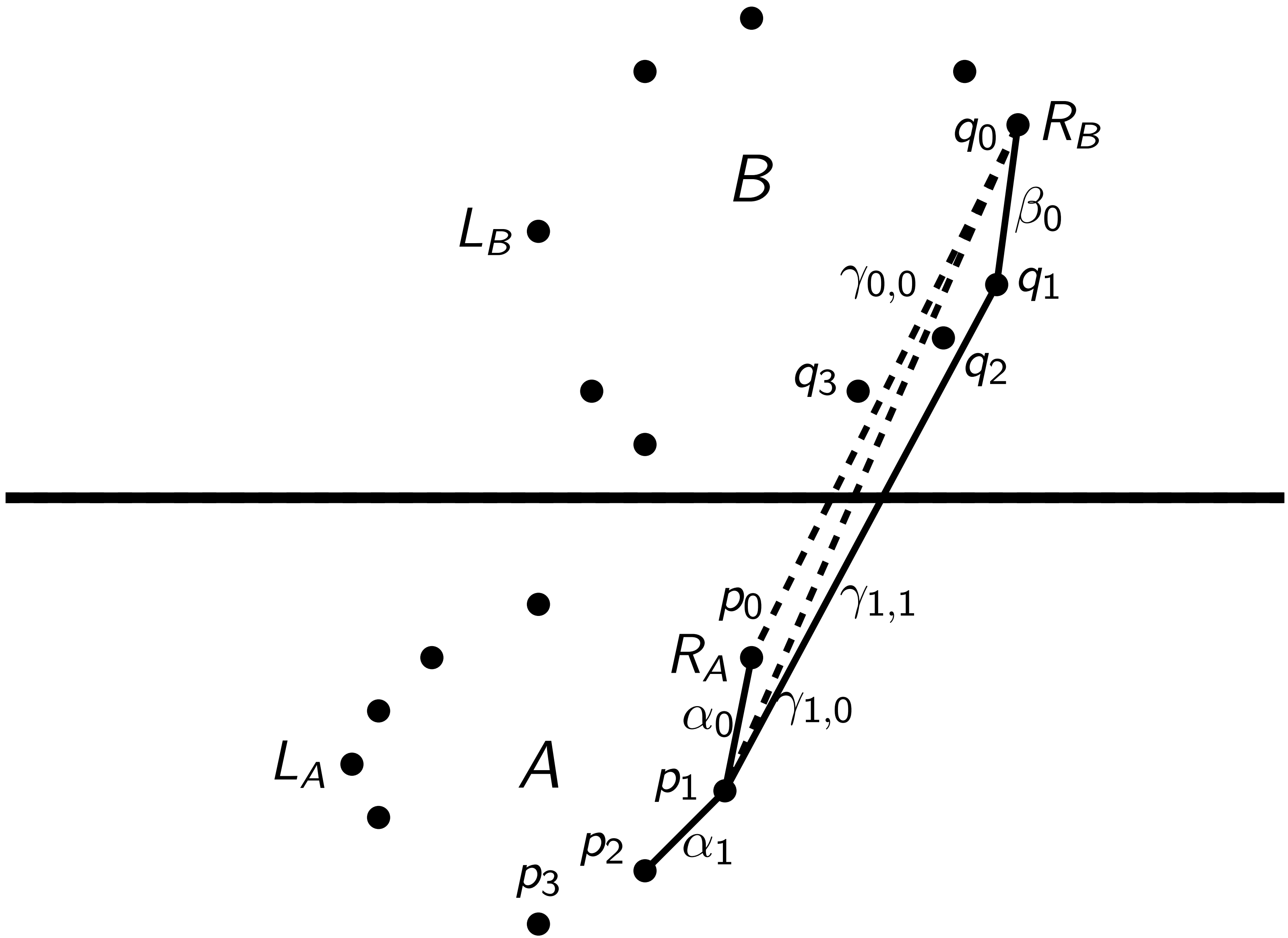
Selecting points on right side: q_0, q_1 and p_1, p_2, p_3, p_4

- We want to find first $i_R^* = p_1$ and last $j_R^* = q_1$ and skip $q_2, \ldots$ and p_0



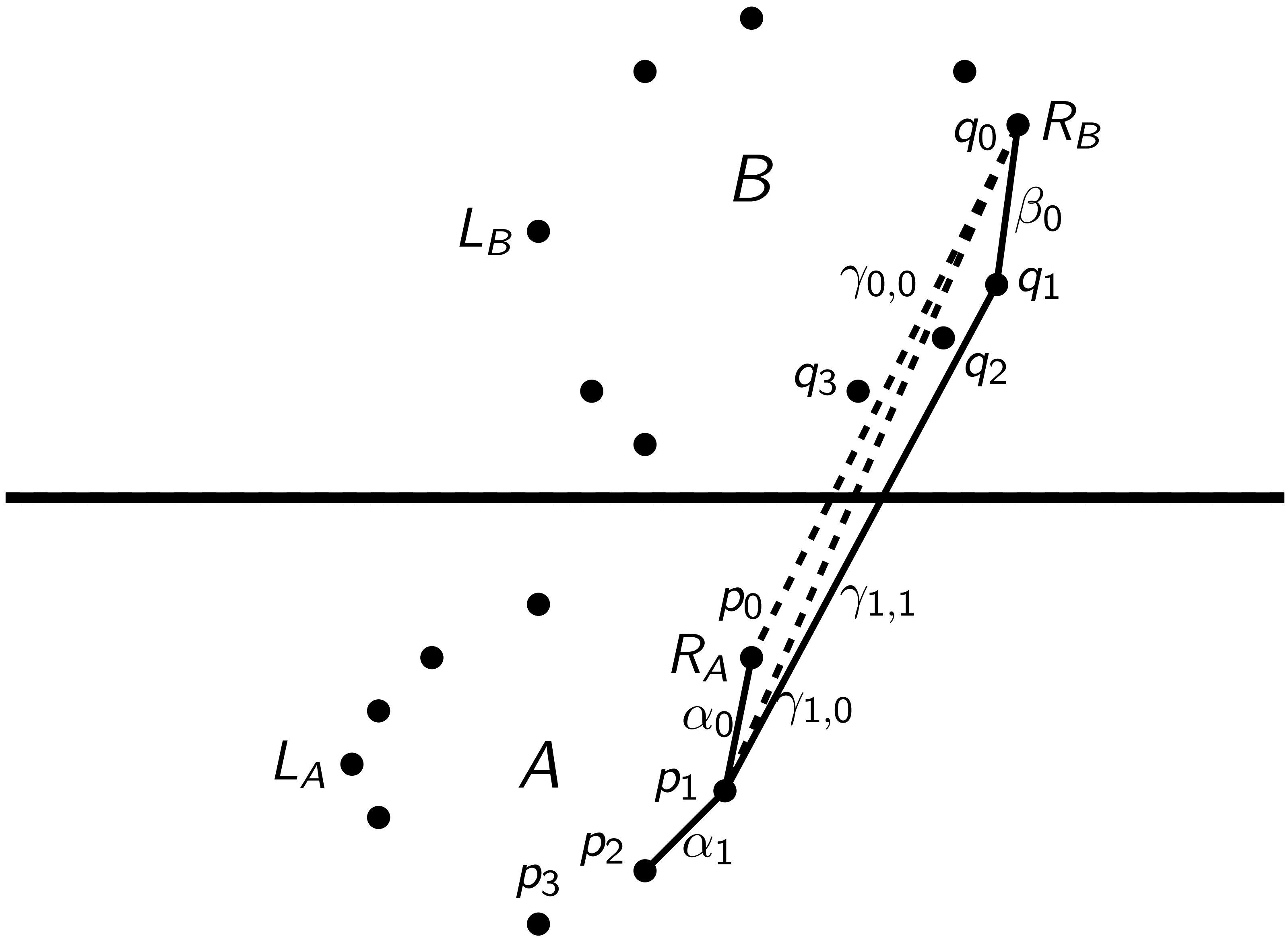
Exclude p_0

- First compare α_0 and $\gamma_{0,0}$. Since $\alpha_0 > \gamma_{0,0}$ we exclude p_0



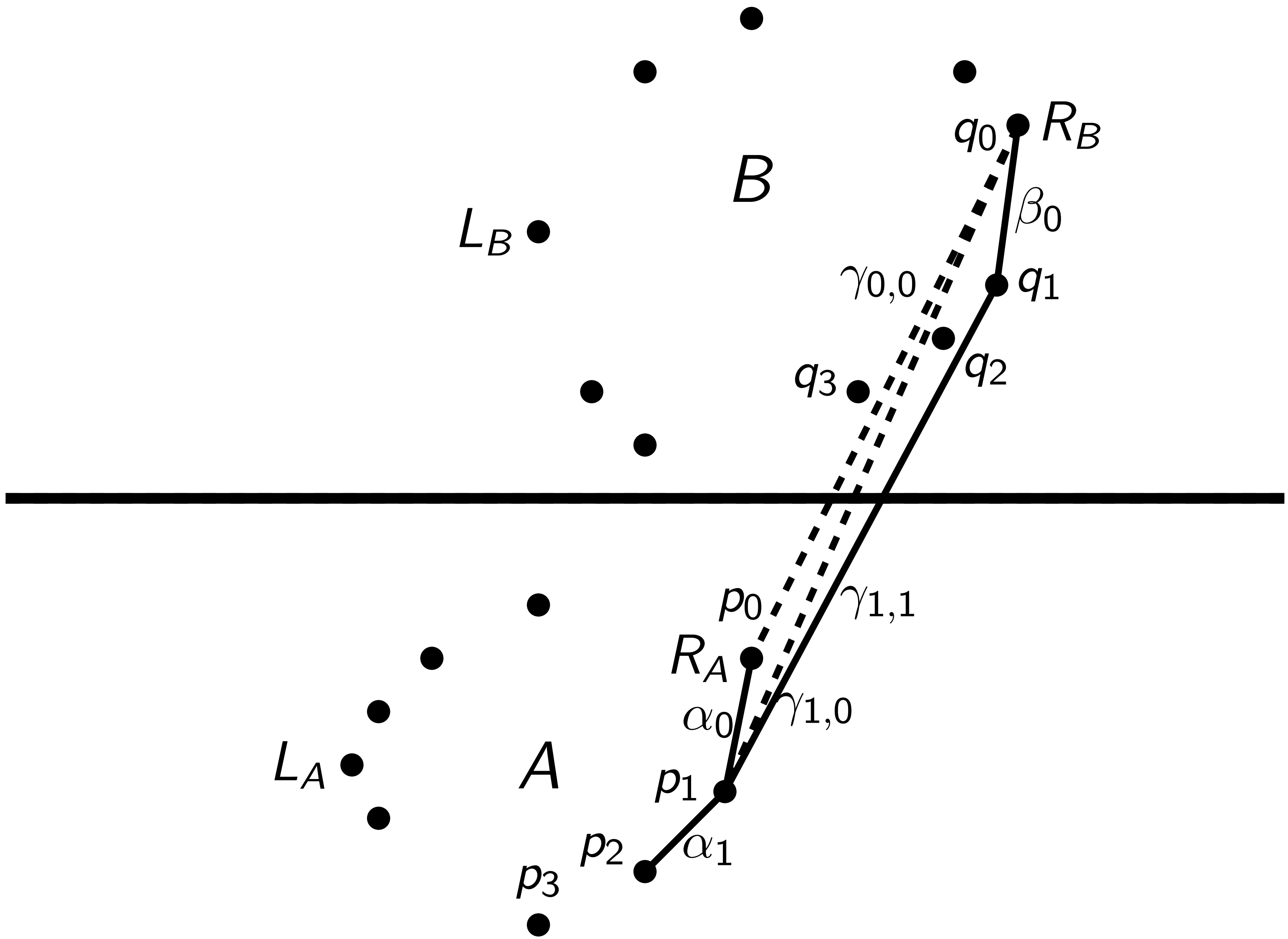
Include q_1

- Compare α_1 and $\gamma_{1,0}$. Since $\alpha_1 \leq \gamma_{1,0}$ we check β_0 and include q_1.



We are done at right side

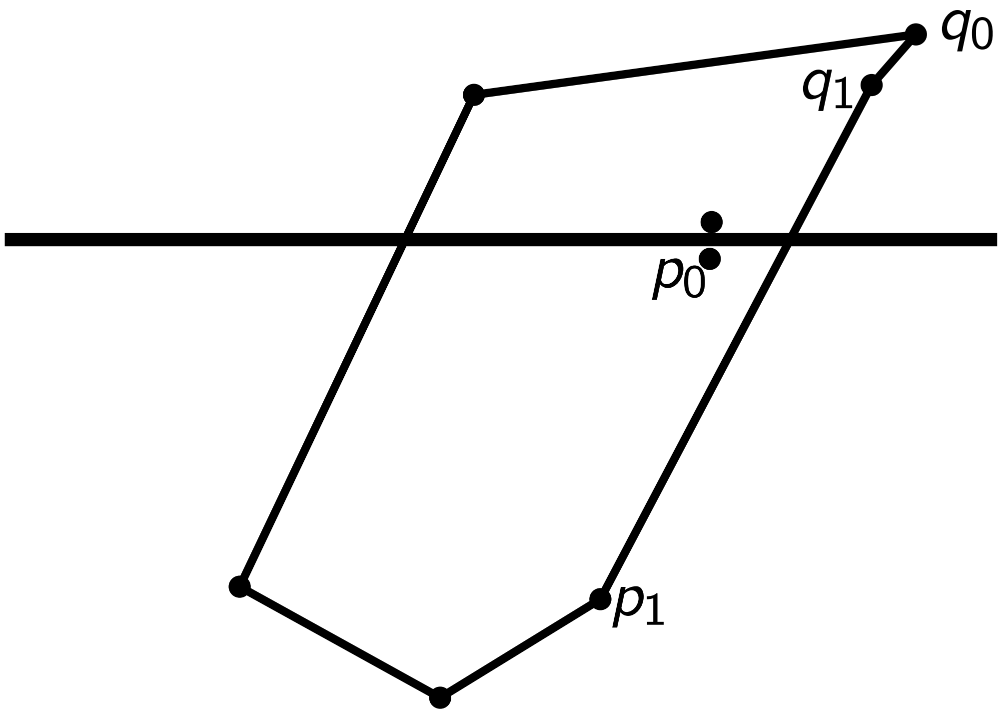
- Since both α_1 and β_1 are less than $\gamma_{1,1}$ we have found i_R^* and j_R^*



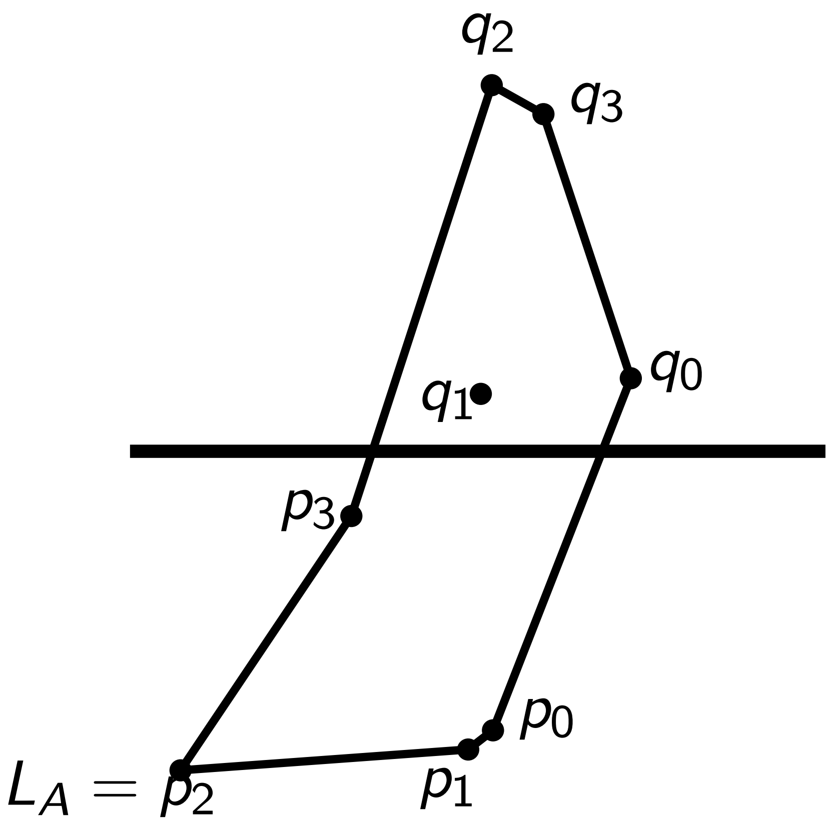
Next we do the same on the left side

- On the left side we want to find i_L^* and j_L^*
- Then we add lower points from i_R^* up to and including i_L^* to the output
- And add upper points from j_L^* up to and including j_R^* to the output
- Does it matter which of α_i and β_j we first compare with $\gamma_{i,j}$?

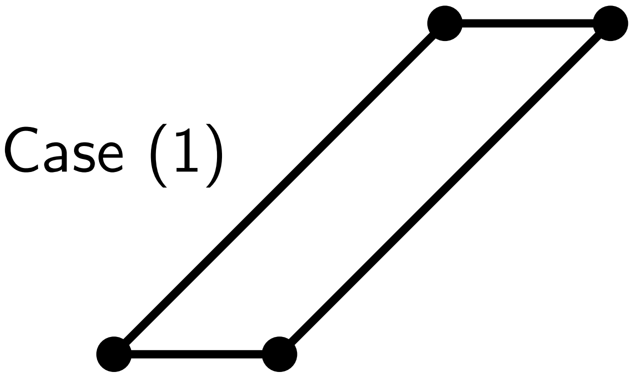
Wrong order on right side: comparing β_j before α_i



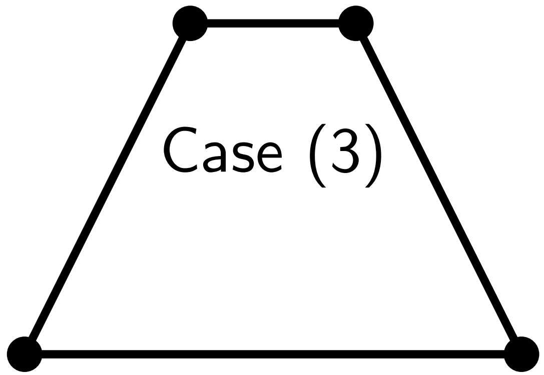
Wrong order on left side: comparing α_i before β_j



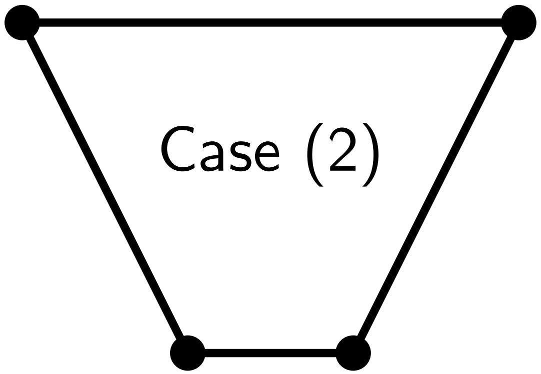
Four cases



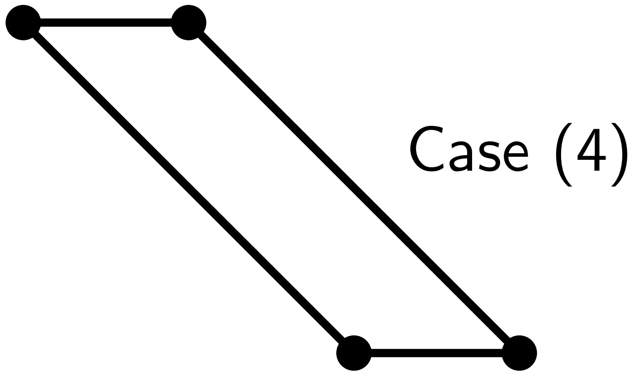
$L_A < L_B$ and $R_A < R_B$



$L_A < L_B$ and $R_A \geq R_B$



$L_A \geq L_B$ and $R_A < R_B$



$L_A \geq L_B$ and $R_A \geq R_B$

Some hints

```
function k(p, q)
begin
        return (q.y − p.y)/(q.x − p.x)
end

function compute_k(p)
begin
        n ← |p|
        α ←new double [n]
        for i = 0; i < n; i ← i + 1
                α[i] ← k(p[i], p[i + 1  mod n])
        return α
end

function add(k, from, to, q, p, n)
begin
        j ← from
        do {
                q[k] ← p[j]
                k ← k + 1
                i ← j
                j ← (j + 1)  mod n
        } while i ≠ to
        return k
end
```

Avoid middle point when three points are on a line

function *include_points*(j, q, p, n)
begin
    for $k \leftarrow 0$; $k < n$; $k \leftarrow k + 1$ {
        $u \leftarrow (n + k + j - 1) \mod n$ // point before v
        $v \leftarrow (n + k + j - 0) \mod n$ // point that may be excluded
        $w \leftarrow (n + k + j + 1) \mod n$ // point after v

        if not *line_segment*(q[v], q[u], q[w]) {
            $p[i] \leftarrow q[v]$
            $i \leftarrow i + 1$
        }
    }
    return i
end

Case 1

```
function case_1(a, n_a, b, n_b, α, β, i_L, j_L)
    i ← 0
    j ← 0
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        if (α_i > γ_{i,j}  or α_i == −∞)  and i < i_L then
            i ← i + 1
        else if (β_j > γ_{i,j}  or β_j == −∞)  and j < j_L then
            j ← j + 1
        else
            break
    }
    i*_R ← i
    j*_R ← j
    i ← i_L
    j ← j_L
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        if β_j > γ_{i,j}  and j ≠ 0 then
            j ← (j + 1)  mod n_b
        else if α_i > γ_{i,j}  and i ≠ 0 then
            i ← (i + 1)  mod n_a
        else
            break
    }
    i*_L ← i
    j*_L ← j
    return (i*_R, i*_L, j*_L, j*_R)
```

Case 2

```
function case_2(a, n_a, b, n_b, α, β, i_L, j_L)
    i ← 0
    j ← 0
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        if (α_i > γ_{i,j}  or α_i == −∞)  and i < i_L then
            i ← i + 1
        else if (β_j > γ_{i,j}  or β_j == −∞)  and j < j_L then
            j ← j + 1
        else
            break
    }
    i*_R ← i
    j*_R ← j
    i ← i_L
    j ← j_L
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        a_k ← (n_a + i − 1)  mod  n_a
        b_k ← (n_b + j − 1)  mod  n_b
        if isfinite(α_{a_k})  and α_{a_k} < γ_{i,j}  and i ≠ 0 then
            i ← a_k
        else if β_{b_k} < γ_{i,j}  and j ≠ 0 then
            j ← b_k
        else
            break
    }
    i*_L ← i
    j*_L ← j
    return (i*_R, i*_L, j*_L, j*_R)
```

Case 3

function *case_3*(a, n_a, b, n_b, α, β, i_L, j_L)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while (1) {
        $\gamma_{i,j} \leftarrow k(a_i, b_j)$
        $a_k \leftarrow (n_a + i - 1) \mod n_a$
        $b_k \leftarrow (n_b + j - 1) \mod n_b$
        if $\beta_{b_k} < \gamma_{i,j}$ **and** $j < j_L$ **then**
            $j \leftarrow b_k$
        else if $\alpha_{a_k} < \gamma_{i,j}$ **and** $i \neq i_L$ **then**
            $i \leftarrow a_k$
        else
            break
    }
    $i_R^* \leftarrow i$
    $j_R^* \leftarrow j$
    $i \leftarrow i_L$
    $j \leftarrow j_L$
    while (1) {
        $\gamma_{i,j} \leftarrow k(a_i, b_j)$
        if $\beta_j > \gamma_{i,j}$ **and** $j \neq 0$ **then**
            $j \leftarrow (j + 1) \mod n_b$
        else if $\alpha_i > \gamma_{i,j}$ **and** $i \neq 0$ **then**
            $i \leftarrow (i + 1) \mod n_a$
        else
            break
    }
    $i_L^* \leftarrow i$
    $j_L^* \leftarrow j$
    return $(i_R^*, i_L^*, j_L^*, j_R^*)$

Case 4

```
function case_4(a, n_a, b, n_b, α, β, i_L, j_L)
    i ← 0
    j ← 0
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        a_k ← (n_a + i − 1)  mod  n_a
        b_k ← (n_b + j − 1)  mod  n_b
        if β_{b_k} < γ_{i,j}  and j ≠ j_L then
            j ← b_k
        else if α_{a_k} < γ_{i,j}  and i ≠ i_L then
            i ← a_k
        else
            break
    }
    i*_R ← i
    j*_R ← j
    i ← i_L
    j ← j_L
    while (1) {
        γ_{i,j} ← k(a_i, b_j)
        a_k ← (n_a + i − 1)  mod  n_a
        b_k ← (n_b + j − 1)  mod  n_b
        if isfinite(α_{a_k})  and α_{a_k} < γ_{i,j}  and i ≠ 0 then
            i ← a_k
        else if isfinite(β_{b_k})  and β_{b_k} < γ_{i,j}  and j ≠ 0 then
            j ← b_k
        else
            break
    }
    i*_L ← i
    j*_L ← j
```

More hints

- You don't need to split the code into four cases if you don't want to (simpler though, in my opinion, and likely faster)
- If you fail a test case it is a good idea to print the points in Matlab or to a pdf-file
- Implementing the Preparata-Hong algorithm is harder than Graham scan so why would you want to do it?
- An advantage of divide-and-conquer algorithms is that they are easier to parallelize
- Compute $CH(A)$ and $CH(B)$ by different threads at the same time
- If you have e.g. 80 hardware threads, you can easily let 64 work in parallel
- The machine `power.cs.lth.se` has just 80 hardware threads

Is p on a line segment between q and r in a plane?

- We want to know if p is between two points on a line in which case p should not be in the convex hull
- $\mathbf{u} = \overline{qp}$
- $\mathbf{v} = \overline{qr}$
- $\mathbf{w} = \mathbf{u} \times \mathbf{v}$.
- $\mathbf{w} = w_3\mathbf{e_3} = (u_1 v_2 - u_2 v_1)\mathbf{e_3}$ due to a plane.
- If $w_3 \neq 0$ then p is not on the line through q and r.
- Assume instead $w_3 = 0$ so the points are colinear.
- Is p between q and r?
- Compute the dot products $\mathbf{v} \cdot \mathbf{v}$ and $\mathbf{u} \cdot \mathbf{v}$
- If $\mathbf{u} \cdot \mathbf{v} < 0$ then $\mathbf{u}$ and $\mathbf{v}$ have opposite directions so p is not between
- Otherwise if $\mathbf{u} \cdot \mathbf{v} > \mathbf{v} \cdot \mathbf{v}$ then p is also not between them