Data Structures and Algorithms – W14

Huffman's codes

Knapsack problems

Convex hull

Contents

- Data compression
  - Prefix codes
  - Huffman codes, creation algorytm
- Knapsack problem:
  - fractional knapsack problem and the solution
  - 0-1 knapsack problem and the solution
- Convex hull:
  - Definitions
  - Convex hull problem
  - Grahama's algorithm – „sweeping" technique

Data compression

- Suppose we have a 100,000-character data file that we wish to store compactly.
- The characters in the file occur with the frequencies given by below table.
- In the file only 6 different characters appear.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Fixed-length code needs:
3*100,000 = 300,000 bits

Variable-length codeword needs:
$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits
It means about 25% less bits

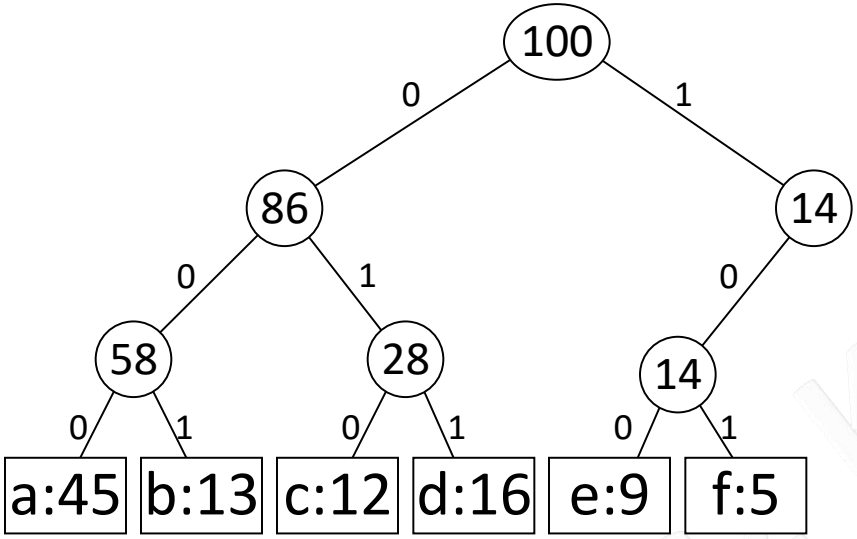
Prefix codes

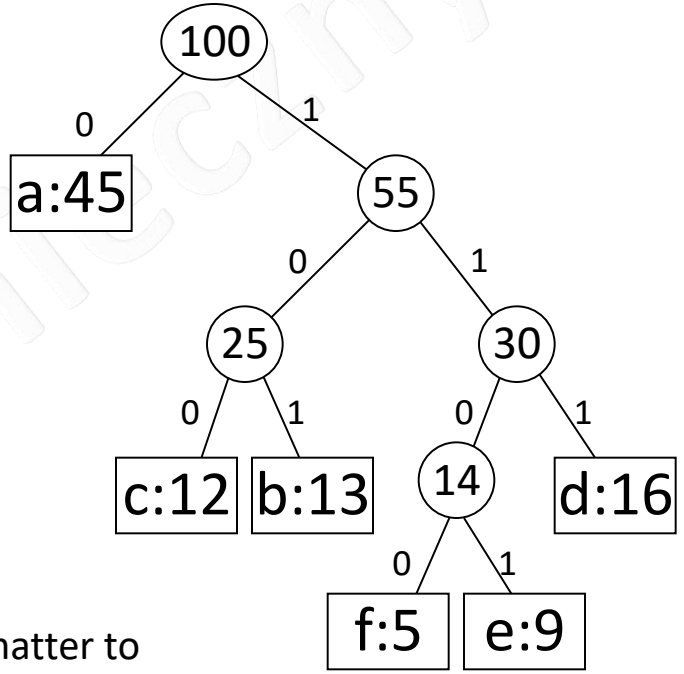
- No codeword is also a prefix of some other codeword. Such codes are called **prefix codes**.
- **Encoding** is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of the table, we code the 3-character file „abc” as 0·101·100 = 0101100, where we use ‘·’ to denote concatenation.
- Prefix codes are desirable because they simplify **decoding**. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as 0 · 0 · 101 · 1101, which decodes to „aabe”.
- In practice, there will be a header in the file showing the coding used.

	a	b	c	d	e	f
prefix codes	0	101	100	111	1101	1100

Coding tree

- The leaves represent coded characters.
- The path from the root to the leaf represents the coding method of the symbol, where the path to the left child is 0, and the path to the right is 1.





Given a tree *T* corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character *c* in the alphabet *C*, let *f*(*c*) denote the frequency of *c* in the file and let $d_T(c)$ denote the depth of *c*'s leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character *c*. The number of bits required to encode a file is thus *B(T)* which we define as the cost of the tree *T*

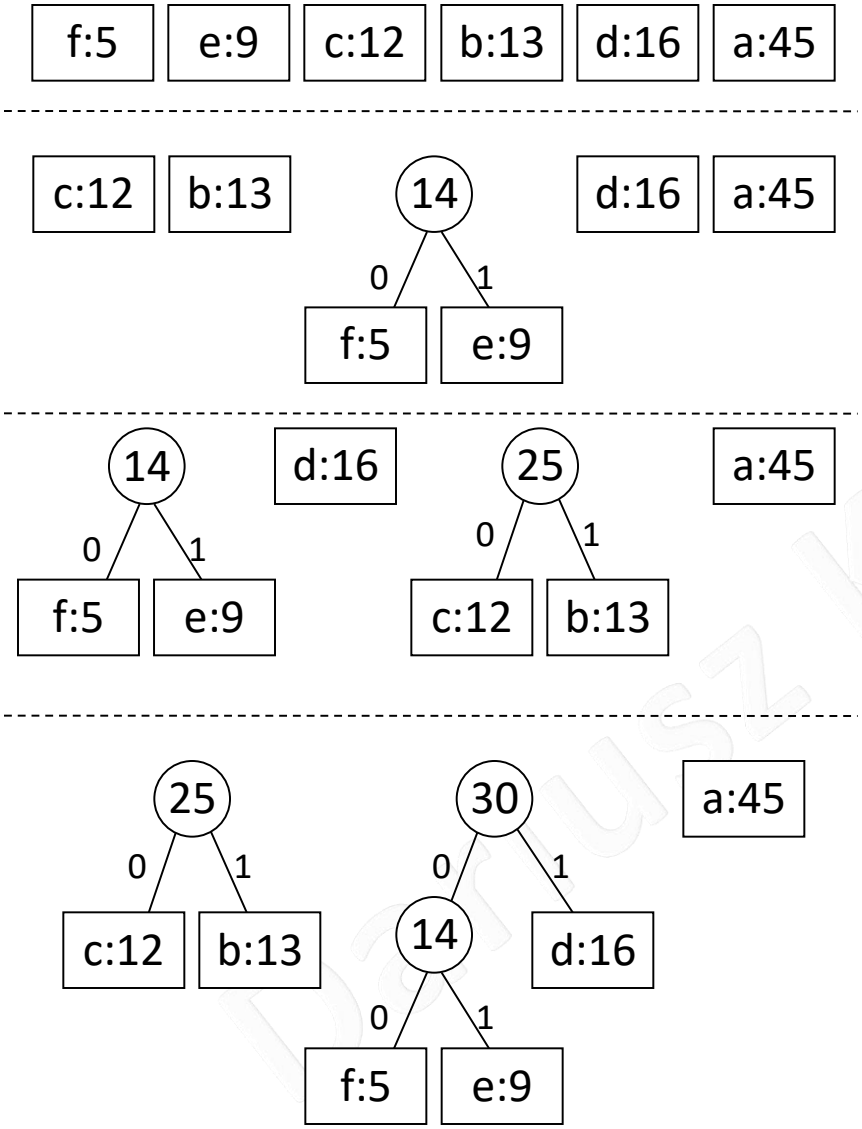
$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Huffman codes - Code

- C is a set of n characters.
- Each character $c \in C$ is an object with a defined frequency $f[c]$.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree.
- A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together.
- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged

```
{ 1}  Huffman(C)
{ 2}  n := |C|
{ 3}  Q := C
{ 4}  for i:=1 to n-1 do
{ 5}    z := Allocate_Node()
{ 6}    x := left[z] := Extract_Min(Q)
{ 7}    y := right[z] := Extract_Min(Q)
{ 8}    f[z] := f[x] + f[y]
{ 9}    Insert(Q,z)
{10}  return Extract_Min(Q)
```

Huffman codes - Example



Huffman codes - complexity

- *Q* is implemented as a binary minheap.
- For a set *C* of *n* characters, the initialization of *Q* in line 2 can be performed in *O*(*n*) time using the BUILD-MIN-HEAP procedure.
- The **for** loop in lines 3-8 is executed exactly *n*-1 times
  - each heap operation requires time *O* (lg *n)*
  - loop contributes *O* (*n* lg *n*) to the running time.
- the total running time of HUFFMAN on a set of *n* characters is *O* (*n* lg *n*).

Letters and (interesting) frequency generating very unbalanced tree:
a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21
a:1 b:1 c:2 d:4 e:8 f:16 g:32 h:64

Knapsack problems

- Knapsack problems :
  - The ***0-1 knapsack problem:***
    - **Input**: we have a knapsack with a maximum capacity *W* and a set of *n* items { x_1, x_2, ..., x_j, ..., x_n } with each element having a specific value v_j and size w_j (from natural numbers)
    - **Objective** : select a subset of items whose total size does not exceed the capacity of *W* and their total value is the largest possible.
    - **Assumption**: You can only choose the entire item.
  - **Fractional knapsack problem**, the same input and objective, but:
    - **Assumption**: you can choose any piece of item (any fractional part).

Solution for fractional knapsack

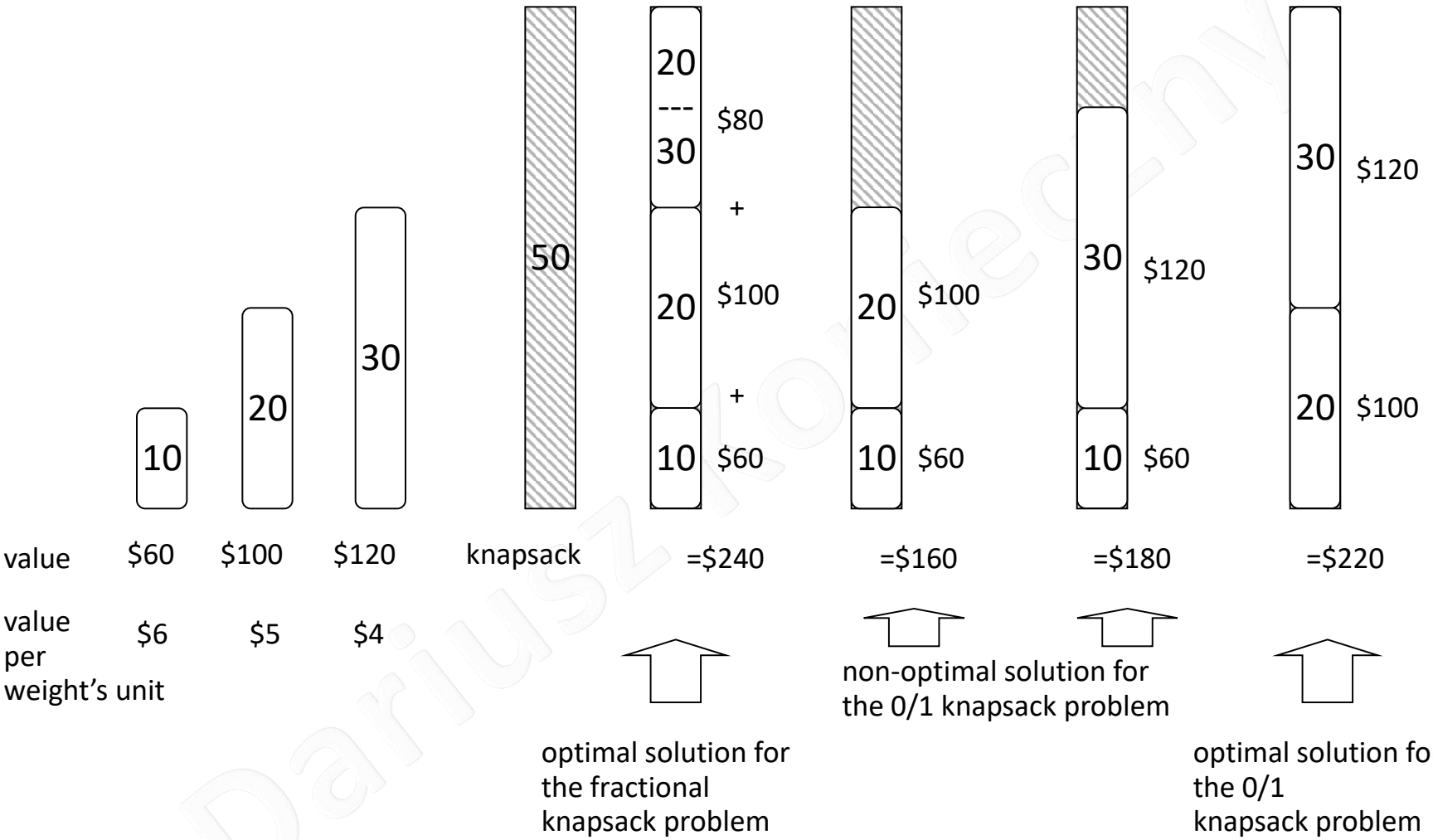
fractional knapsack problem – solution: (*greedy algorithm):*

- compute the value per weight's unit v_i/w_i for each item
- take as much as possible of the item with the greatest value per weight's unit . If the supply of that item is exhausted and you can still carry more, take as much as possible of the item with the next greatest value per weight unit , and so forth until you can't carry any more

- because of sorting the items by value per weight's unit, the greedy algorithm runs in O(n lg n) time

- for ***0-1 knapsack problem*** this greedy strategy does not work!

Example 1



0/1 knapsack problem - solution

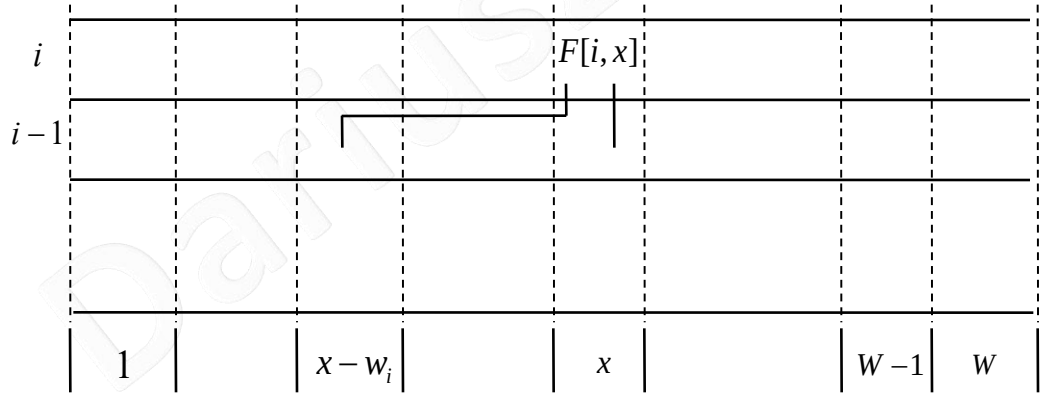
- **0/1 knapsack problem** – solution: (*dynamic programming*)
- Knowing the best solutions for items from a set $\{v_1,v_2,\ldots,v_{i-1}\}$ for a knapsack with capacity from 1 to W, we can find a formula to compute the best solutions for items from a set $\{v_1,v_2,\ldots,v_{i-1},v_i\}$

$$\sum_{i=1}^{n} w_i t_i \le W \qquad t_i \in \{0,1\}$$

$$\sum_{i=1}^{n} v_i t_i \to \max$$

$$F[i,x] = \begin{cases} 0 & x \ge 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1,x], (F[i-1,x-w_i]+v_i)\} & 1 \le i \le n \end{cases}$$

Complexity: O(nW)



Example 2 – 1/4

	1	2	3	4	5	6	7
weight, w_i	3	10	8	6	2	9	
value, v_i	5	12	10	7	1	11	

Capacity W=20

7																			
6																			
5																			
4																			
3																			
2	0	0	5	5	5	5	5	5	5	12	12								
1	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Example 2 – 2/4

	1	2	3	4	5	6	7
weight, w_i	3	10	8	6	2	9	
value, v_i	5	12	10	7	1	11	

Capacity W=20

7																		
6	0	1	5	5	6	7	7	10	12	12	15	16	17	17	18	19	22	23	24	26
5	0	1	5	5	6	7	7	10	12	12	15	15	17	17	18	19	22	22	24	24
4	0	0	5	5	5	7	7	10	12	12	15	15	17	17	17	19	22	22	24	24
3	0	0	5	5	5	5	5	10	10	12	15	15	17	17	17	17	17	22	22	22
2	0	0	5	5	5	5	5	5	5	12	12	12	17	17	17	17	17	17	17	17
1	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Example 2 – 3/4

	1	2	3	4	5	6	7
weight, w_i	3	10	8	6	2	9	
value, v_i	5	12	10	7	1	11	

Capacity W=20

7																		
6	0	1	5	5	6	7	7	10	12	12	15	16	17	17	18	19	22	23	24	26
5	0	1	5	5	6	7	7	10	12	12	15	15	17	17	18	19	22	22	24	24
4	0	0	5	5	5	7	7	10	12	12	15	15	17	17	17	19	22	22	24	24
3	0	0	5	5	5	5	5	10	10	12	15	15	17	17	17	17	17	22	22	22
2	0	0	5	5	5	5	5	5	5	12	12	12	17	17	17	17	17	17	17	17
1	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Example 2 – 4/4

	1	2	3	4	5	6	7
weight, w_i	3	10	8	6	2	9	
value, v_i	5	12	10	7	1	11	

Capacity W=14

7													
6	0	1	5	5	6	7	7	10	12	12	15	16	17	17
5	0	1	5	5	6	7	7	10	12	12	15	15	17	17
4	0	0	5	5	5	7	7	10	12	12	15	15	17	17
3	0	0	5	5	5	5	5	10	10	12	15	15	17	17
2	0	0	5	5	5	5	5	5	5	12	12	12	17	17
1	0	0	5	5	5	5	5	5	5	5	5	5	5	5
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Computational geometry - concepts and problems

- Basic geometric objects :
  - **Point** p, represented by a pair of coordinates (x_p, y_p) in the XOY system
  - **Segment** *p*-*q*, represented by a pair of *p* and *q* points,
  - **Vector** beginning at *p* and end at *q*, denoted as $p \rightarrow q$
  - **Straight**, represented by any pair of different points lying on it.
- Example problems:
  - relative location of points,
    - on which side of the $p \rightarrow q$ vector lies the *r* point
    - whether the points *x* and *y* lie on the same side of the straight line *p*-*q*
    - whether the point *r* belongs to the segment *p*-*q*
    - whether the segments *p*-*q* and *r*-*s* intersect
  - is the *p* point lying inside the polygon *W*,
  - **finding a convex hull of a set of points**
  - determining whether any pair of segments intersects
  - finding the closest pair of points.
  - …

Relative location of points 1/2

- Acceptable arithmetic operations are **addition**, **subtraction** and **multiplication**. The remaining ones cause rounding and inaccurate calculations!
  - Determining the slope factor of a straight line - NO!
- The atomic operation used in algorithms is the operation of determining the relative position of three points :

  $p = (x_p, y_p)$, $q = (x_q, y_q)$, $r = (x_r, y_r)$
- We build a determinant :

$$\det(p,q,r) = \det \begin{bmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{bmatrix}$$

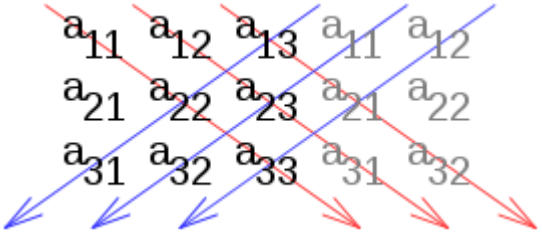
- The determinant sign is equal to the sine of the angle between the $p \rightarrow r$ vector and $p \rightarrow q$ vector.

Determinant - the Sarrus method

- Calculation of the determinant of a 3-degree matrix (Sarrus rule):
  To calculate the determinant:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

- the first two columns are added to the right:

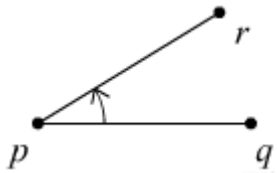


- and then the sum of the products along the red arrows is calculated and subtracted the sum of the products along the blue arrows. The general formula has the following form:

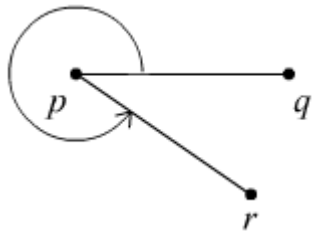
$$(a_{11}\cdot a_{22}\cdot a_{33}+a_{12}\cdot a_{23}\cdot a_{31}+a_{13}\cdot a_{21}\cdot a_{32})-(a_{13}\cdot a_{22}\cdot a_{31}+a_{11}\cdot a_{23}\cdot a_{32}+a_{12}\cdot a_{21}\cdot a_{33})$$

Relative location of points 2/2

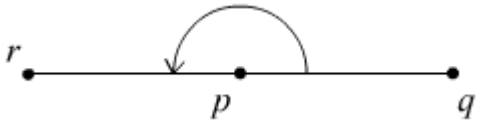
- The point *r* lies on the left side of the $p \rightarrow q$ vector, if $\det(p,q,r) > 0$.



- The point *r* lies on the right side of the $p \rightarrow q$ vector, if $\det(p,q,r) < 0$.



- If $\det(p,q,r) = 0$ then we say that the points are collinear.

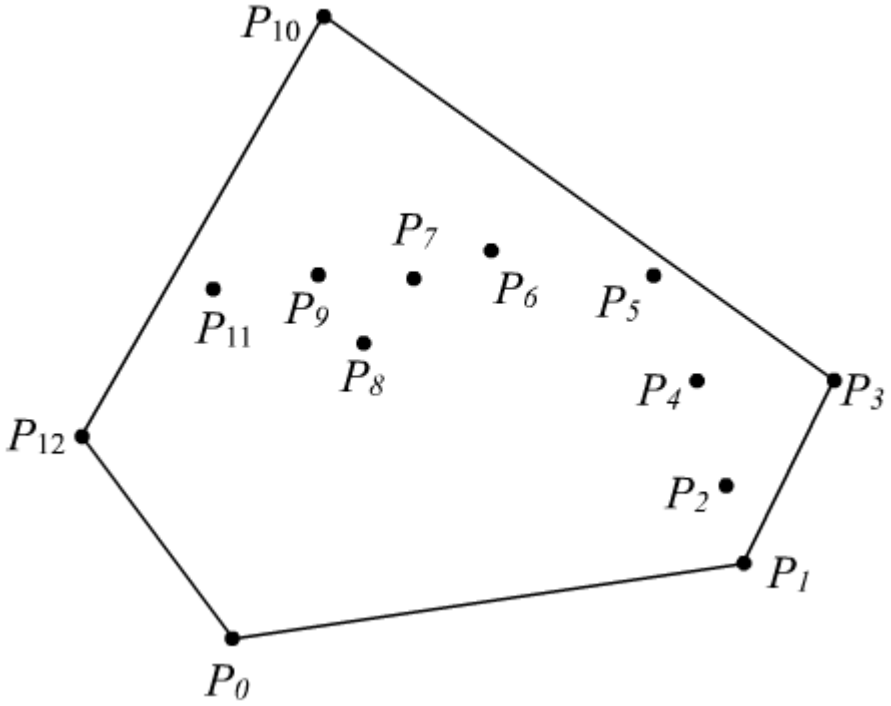


Sweep line algorithm

- In addition to the previously known problem-solving techniques, in the case of geometrical problems, there is one more basic technique: sweep method.
- **Plane sweep**: We start by sorting the points according to one of their coordinates, eg the coordinate x. Next, we review the points by moving the vertical straight (so-called **broom**) from left to right. In the broom, we remember the information about the intersecting objects.
  - Example: construction of an algorithm to check whether in a set of rectangles with sides parallel to the axis of the system any two rectangles overlap
- **Polar sweep**: First, select one of the points and organize the rest of the objects according to their polar coordinate (angular) relative to this point. Then we browse the points according to their order.
  - Example: construction of an algorithm for finding a convex hull of a set of points.

Convex hull problem

- The **convex hull** CH(S) of the finite set of points S is the smallest convex polygon P containing the points of the set S.
- In the convex hull problem we have a given set of points and we want to determine the convex hull vertices in the order of their occurrence on its circumference.



$S = \{P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}\}$

$CH(S) = \{P_0, P_1, P_3, P_{10}, P_{12}\}$

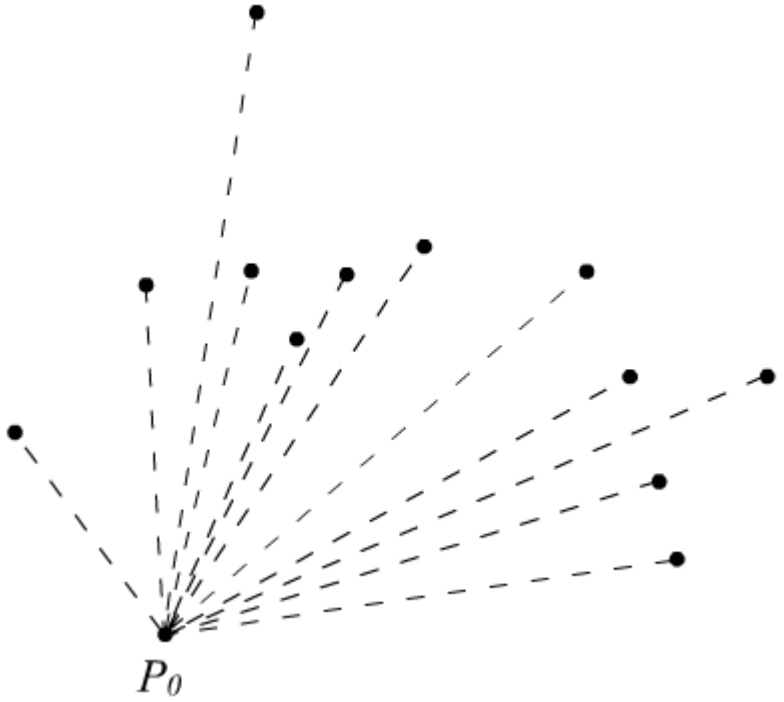
Graham's algorithm - idea

- **Idea of the algorithm**: Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of CH(Q) are eventually popped from the stack. When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.
- Input data for the GRAHAM procedure is at least a 3-element set of *Q* points.
- This procedure uses the function:
  - `top(S)` that returns the top of the stack S,
  - `nextToTop(S)` that returns the second vertex on the stack,
  - `pop(S)` removing the element at the top of the stack from the stack,
  - `push(p, S)` putting the vertex p at the top of the stack.
- The algorithm first requires selecting a starting point from the Q set and sorting the remaining points with respect to their polar coordinate.

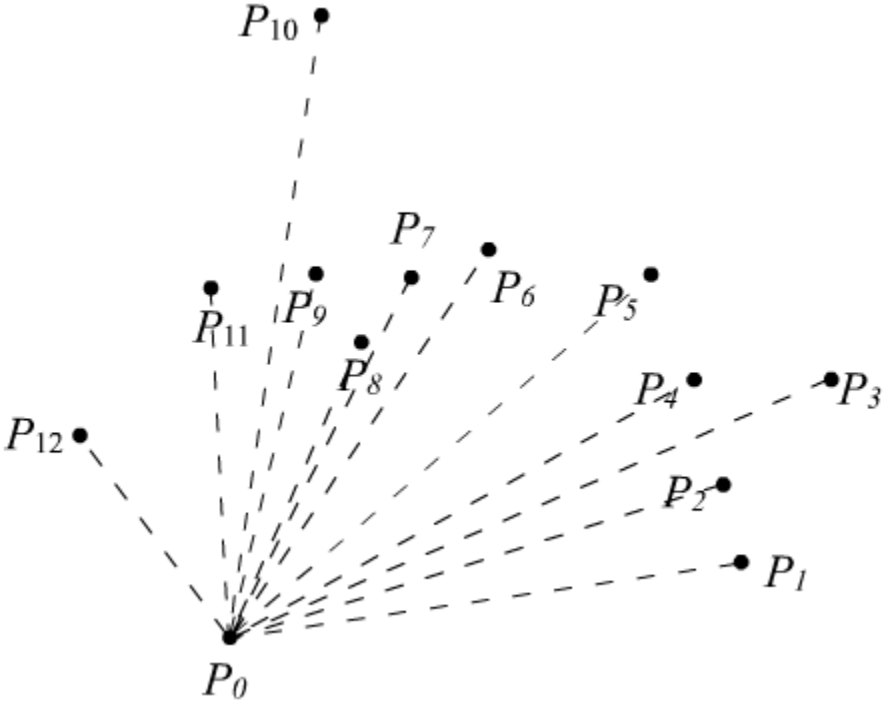
Graham's algorithm - code

```
      Graham(Q)
{ 1}  Let p0 be the point in Q with the minimum y-coordinate,
            or the leftmost such point in case of a tie.
{ 2}  Let {p1, p2, … , pm} be the remaining points in Q,
            sorted by polar angle in counterclockwise order around p0
            (if more than one point has the same angle, remove all but
            the one that is farthest from p0)
{ 3}  Push(p0,S)
{ 4}  Push(p1,S)
{ 5}  Push(p2,S)
{ 6}  for i=3 to m do
{ 7}    while the angle formed by points NEXT-TO-TOP(S), TOP(S), and pi
{ 8}          makes a nonleft turn do
{ 9}      Pop(S)
{10}    Push(S,pi)
{11}  return S
```

Example 1/6

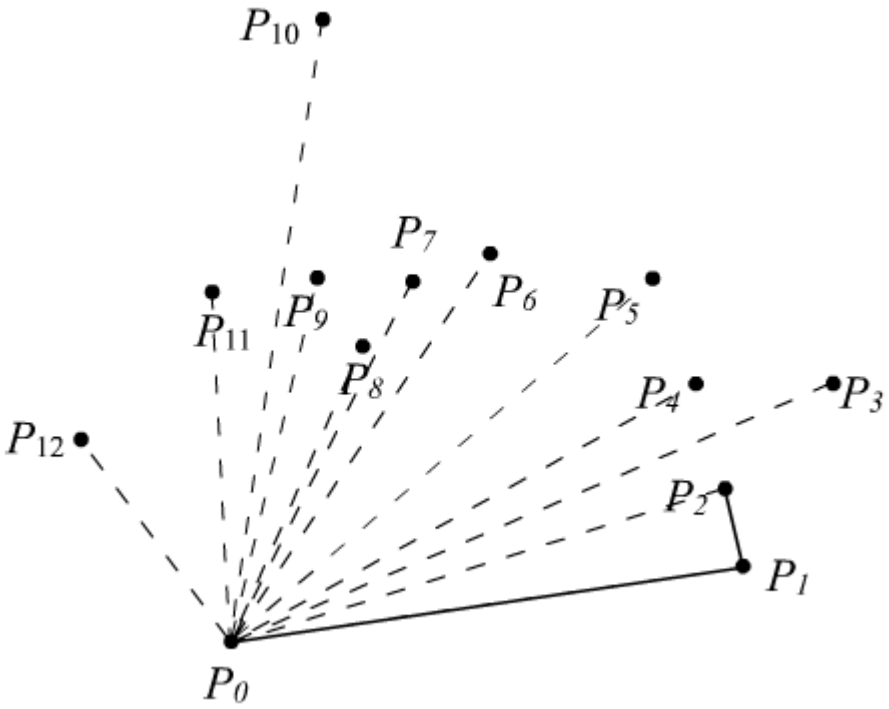


We choose point P_0

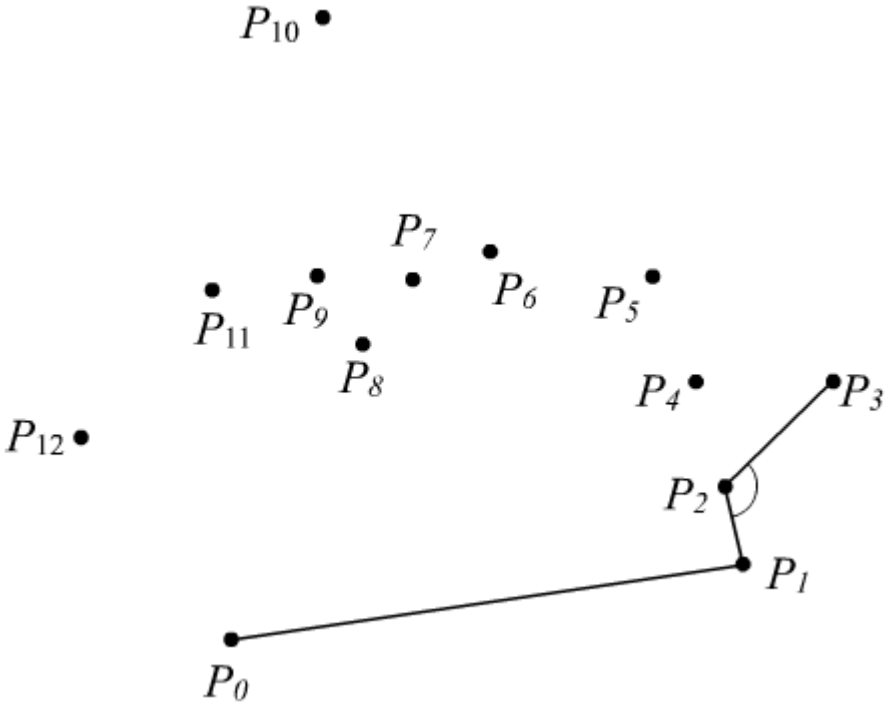


We sort the points in the increasing order of the polar coordinate (in which they will be processed).

Example 2/6

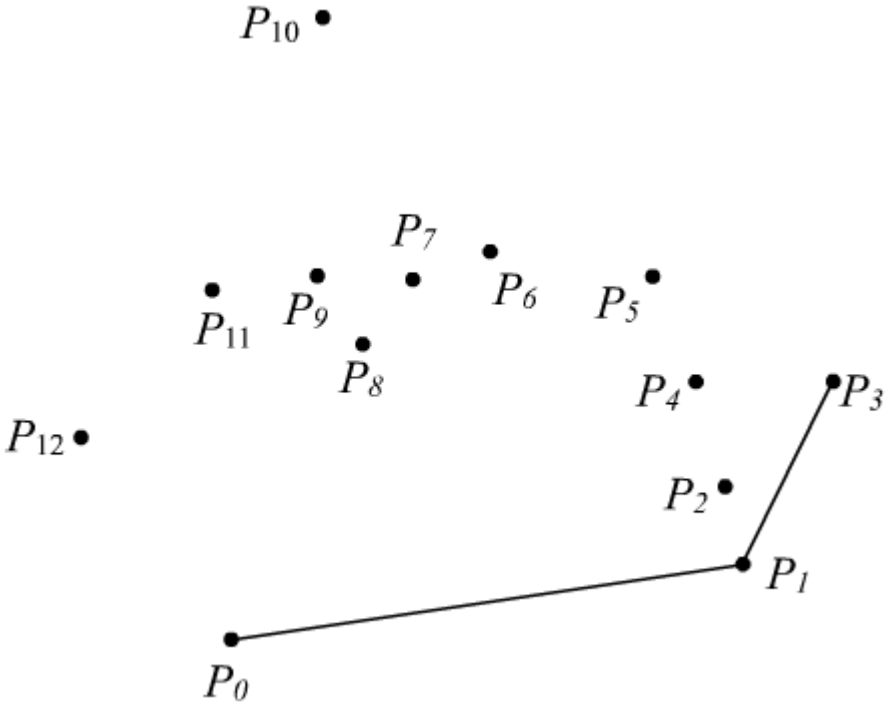


We construct a convex hull composed of the first three points: {P_0, P_1, P_2}. In the next steps, we will construct a convex hull for an increasing number of points.

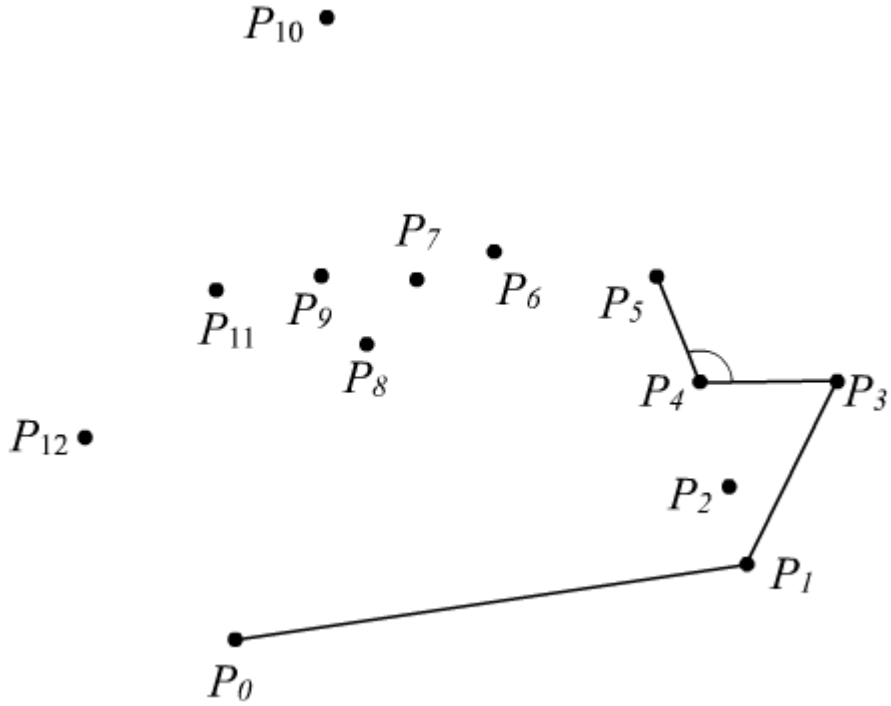


We add a convex hull connection with the next point (P_3). The convex hull {P_0, P_1, P_2, P_3} has ceased to be convex (point P_3 lies to the left of the vector $P_2 \rightarrow P_1$.

Example 3/6

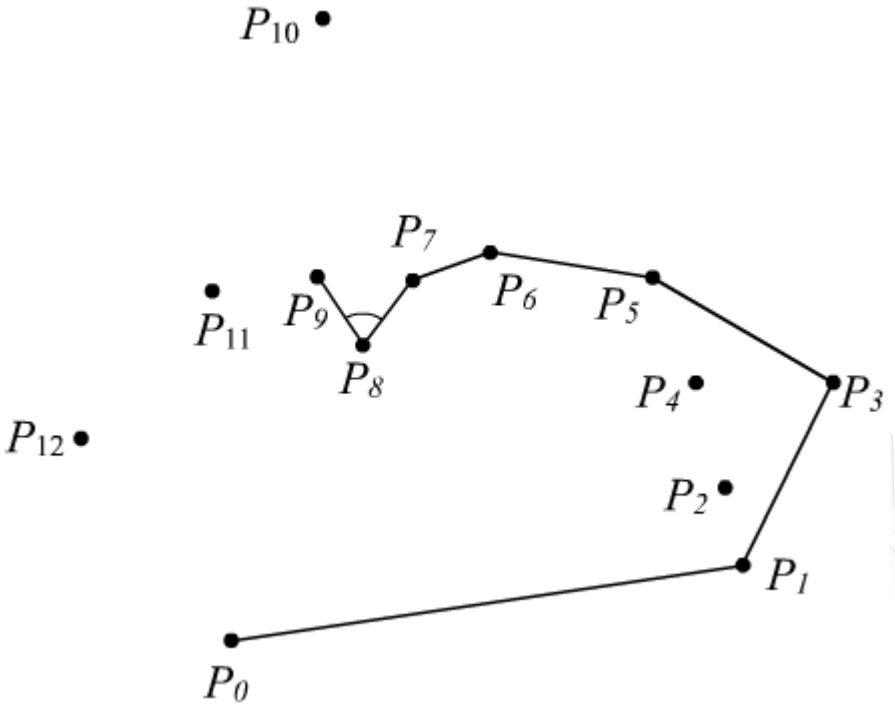


Removed P_2 point from the convex hull (replace the concave angle with segment P_1–P_3).

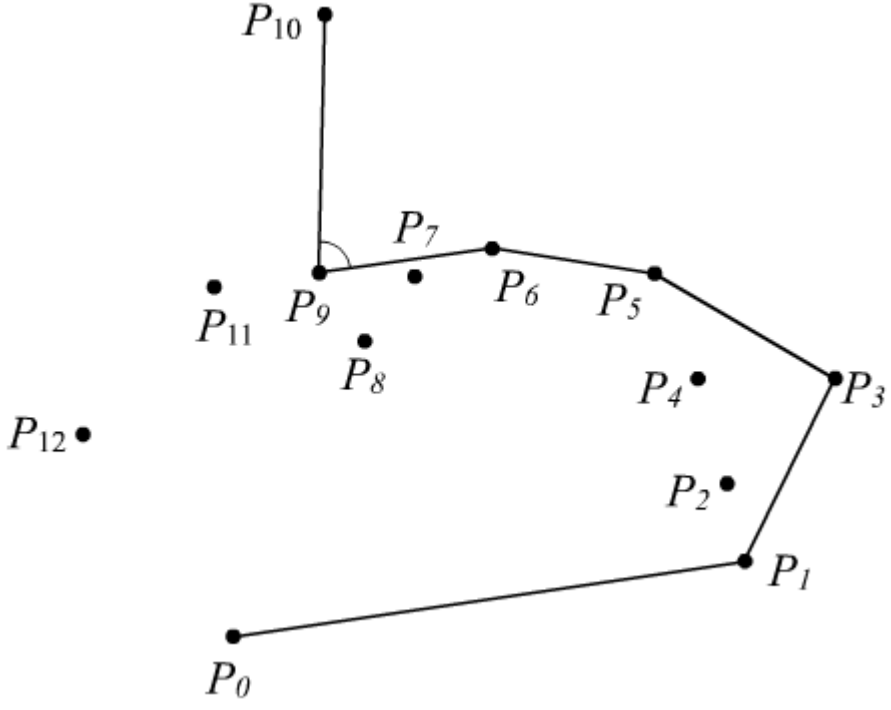


After the next steps ...

Example 4/6

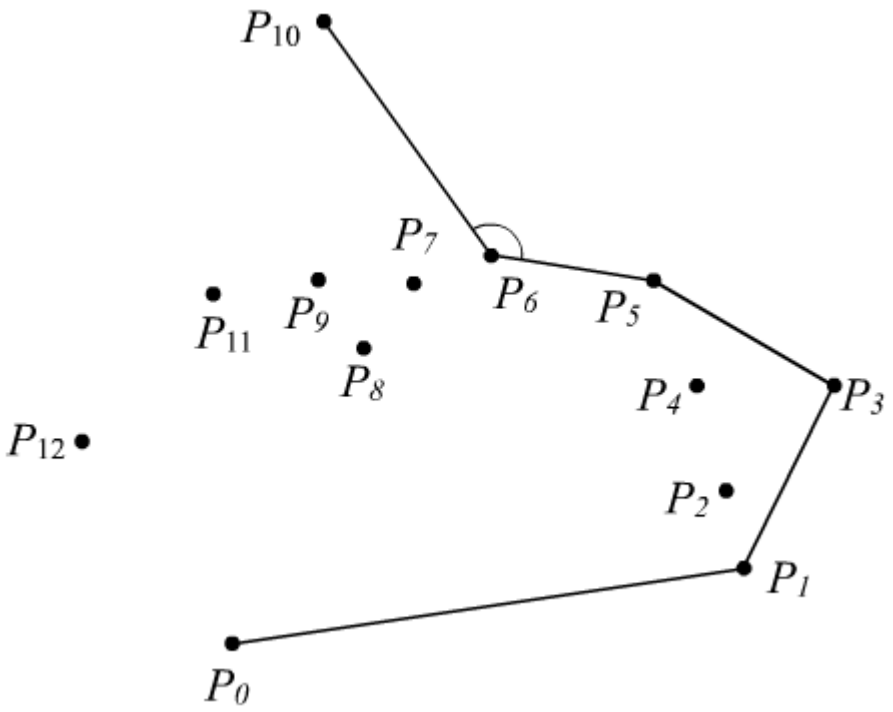


After the next steps ...

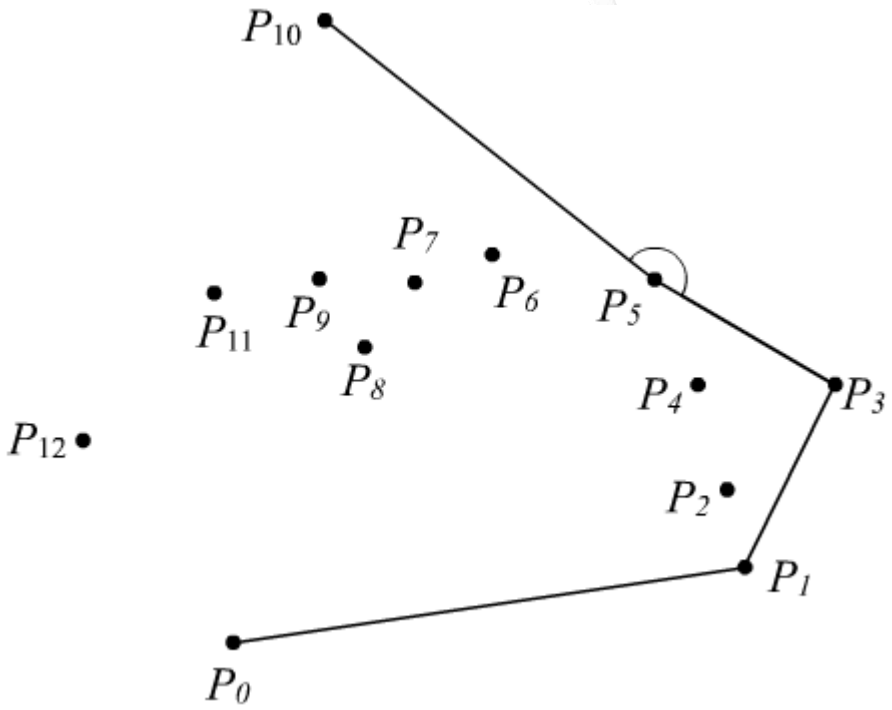


After the next steps ...

Example 5/6

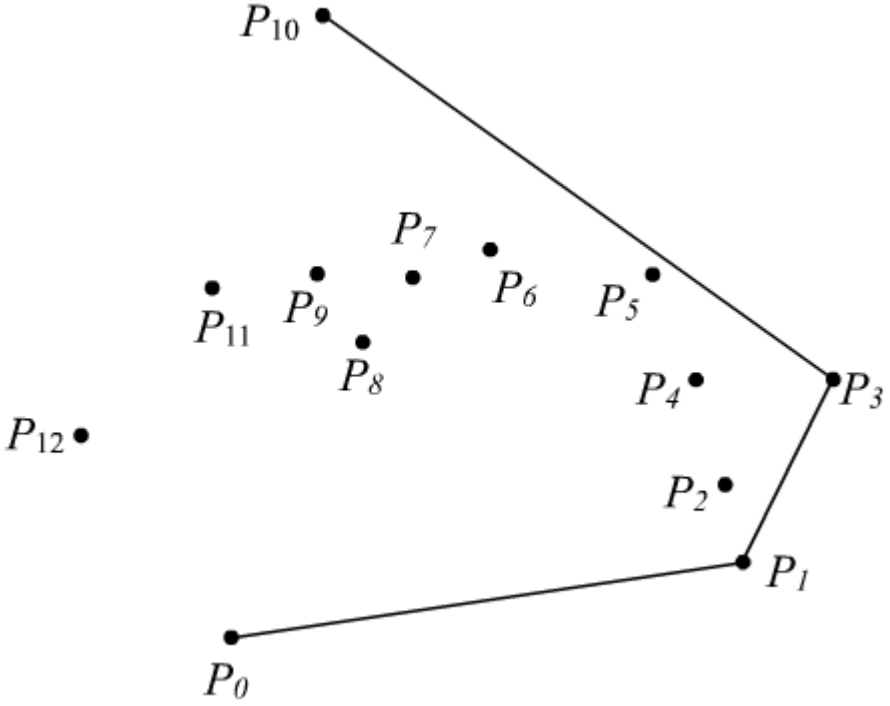


After the next steps ...

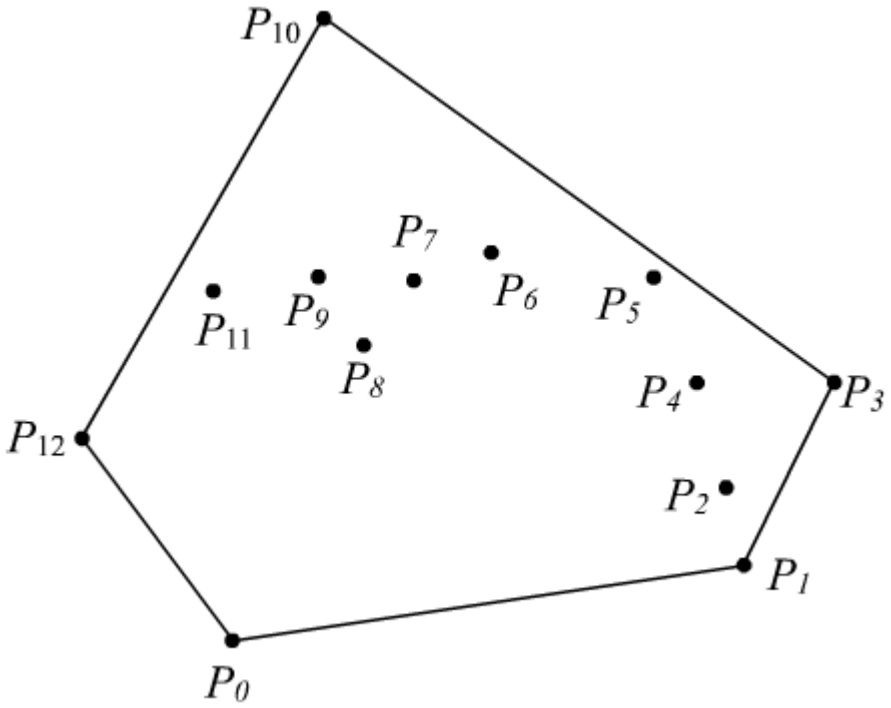


After the next steps ...
CH(S) = {P_0, P_1, P_3, P_5}
Checking the P_5 point which will be rejected, while P_{10} will be added

Example 6/6



After the previous step:
CH(S) = {P_0, P_1, P_3, P_{10}}



After a few more steps, finally:
CH(S) = {P_0, P_1, P_3, P_{10},P_{12}}

Graham's algorithm - complexity

- The complexity of step 2 in which the elements are sorted is O(n log n).
- The number of performances of the **for** loop equals n.
- The total number of executions of the **while** loop can not exceed O(n), because you can remove the element from the stack after inserting it in the stack.
  - It can be proved that there will always be at least two first elements on the stack (P_0 and P_1)
- In one execution of the **for** loop we will perform one Push operation.
- **Conclusion**: The total complexity of Graham's algorithm is O(n log n).