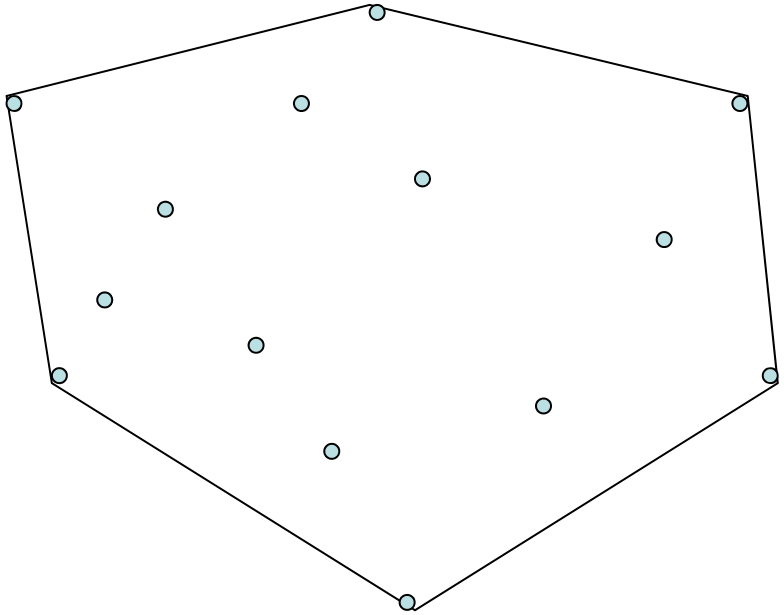
テーマ3：凸包

凸包の定義，構成アルゴリズム

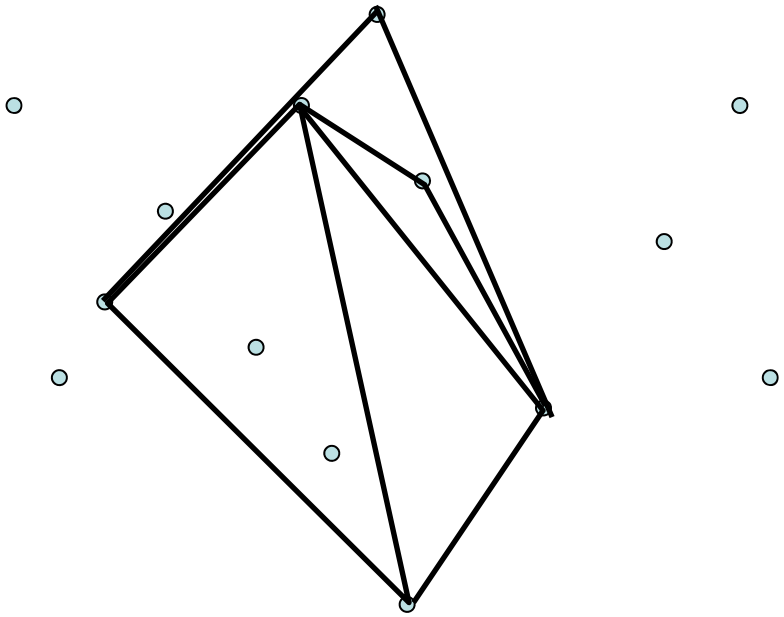
凸包の定義

凸包(convex hull)とは,
与えられた点をすべて包含する最小の凸多角形（凸多面体）のこと.



凸包の定義

凸包(convex hull)とは,
与えられた点をすべて包含する最小の凸多角形(凸多面体)のこと.

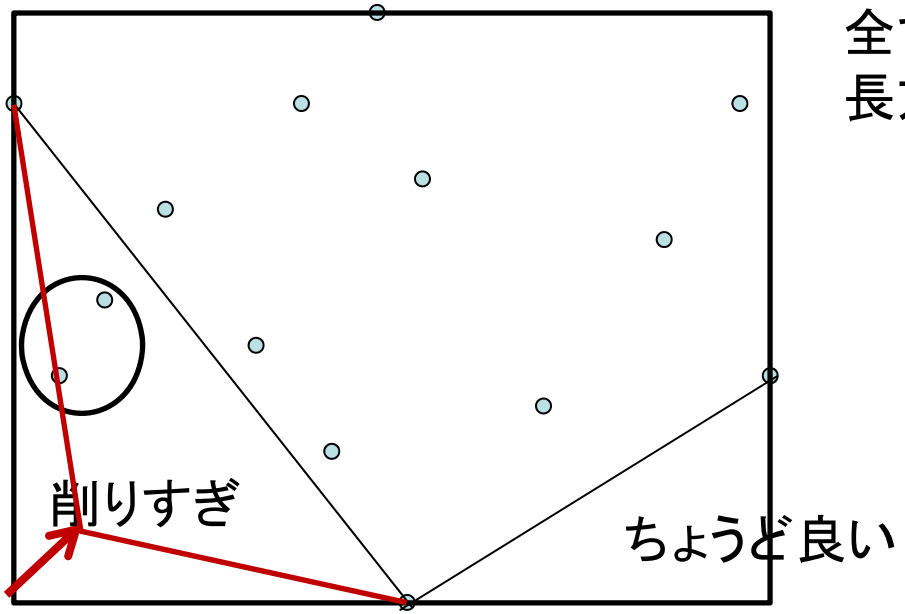


１点ずつ加えて行く.
現在の凸包の内部の点なら
　何もしない.
現在の凸包の外部の点なら
　その点から凸包に接線をひき,
　凸包を修正する.

問題１:凸包の内部/外部の判定方法
問題２:凸包への接線の求め方

凸包の定義

凸包(convex hull)とは,
与えられた点をすべて包含する最小の凸多角形(凸多面体)のこと.



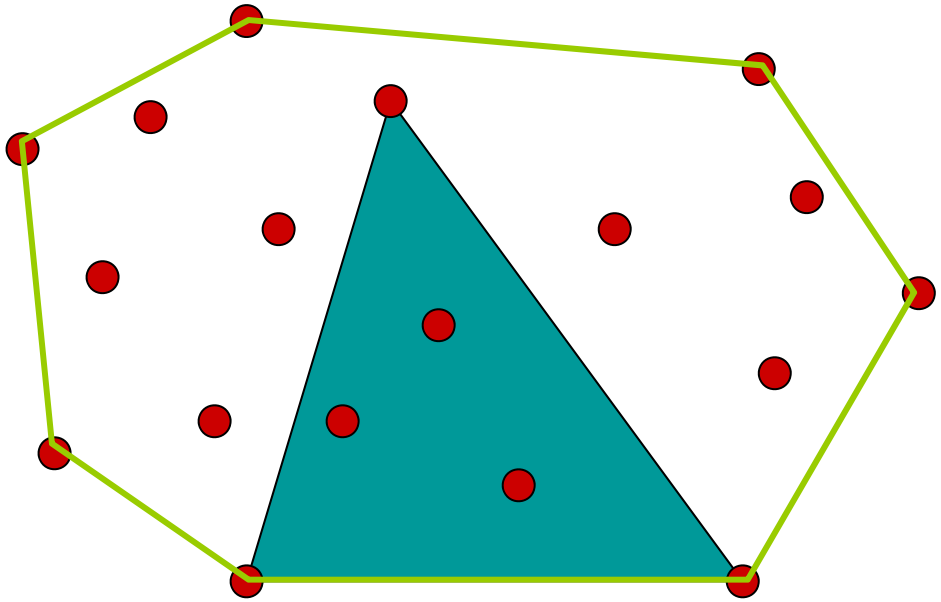
全ての点を含む最小の軸平行
長方形から始め, ここから削っていく.

削りすぎた分をどのように
復元するか？

凸包構成アルゴリズム(1)

凸包の基本的な性質１:
　平面上の点集合Sの凸包の頂点はすべてSの点である.

凸包の基本的な性質２:
　平面上の点集合Sの点が凸包の頂点となるための必要十分条件は,
　その点を内部に含む三角形をなす３点がSの中に存在しないことである.



性質１を証明せよ(背理法).

凸包構成アルゴリズム(2)

凸包構成の素朴なアルゴリズム

(0)S: 平面上に与えられた点集合

(1)Sのすべての3点の組(p, q, r)について，この3点で決まる三角形の内部に含まれる点を集合Sから取り除く.

(2)すべての3点組について調べた後で残った点が凸包の頂点となる.

(3)残った点を凸包内部の点（たとえば重心点）に関して偏角順に並べると凸包を構成できる.

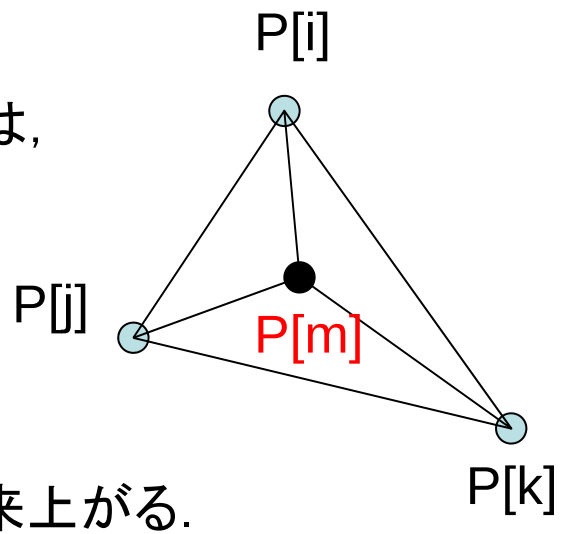
点P[m]が三角形(P[i], P[j], P[k])の内部に存在するのは,

d1 = orientation(P[m], P[i], P[j]);

d2 = orientation(P[m], P[j], P[k]);

d3 = orientation(P[m], P[k], P[i]);

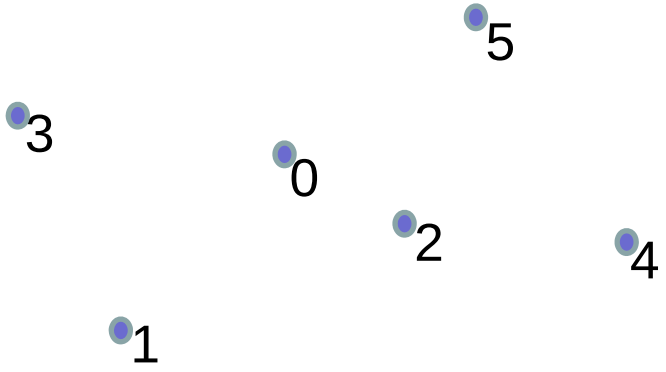
がすべて同じ方向に順序付けられているとき.



この性質を使うと，素朴な凸包構成アルゴリズムが出来上がる.

計算時間は$O(n^4)$.

次の点集合に対してアルゴリズムの動作を確かめよ.
三角形を順に生成してみること.



凸包構成アルゴリズム(2)

凸包構成の素朴なアルゴリズム

(0)S: 平面上に与えられた点集合
(1)Sのすべての３点の組(p, q, r)について，この３点で決まる三角形の内部に含まれる点を集合Sから取り除く.
(2)すべての３点組について調べた後で残った点が凸包の頂点となる.
(3)残った点を凸包内部の点（たとえば重心点）に関して偏角順に並べると凸包を構成できる.

この性質を使うと，素朴な凸包構成アルゴリズムが出来上がる.
計算時間は$O(n^4)$.

すべての３点の組を列挙すると，n個から3個取る組合せ${}_nC_3=n(n+1)(2n+1)/6 = O(n^3)$だけある．それらすべてについて，Sの他の点がその3個で作られる三角形の中に含まれているかどうかを調べるのに，O(n)の時間がかかる.
よって，全体では$O(n^4)$時間かかる.

凸包構成アルゴリズム(2)

凸包構成の素朴なアルゴリズム

(0)S: 平面上に与えられた点集合

(1)Sのすべての3点の組(p, q, r)について，この3点で決まる三角形の内部に含まれる点を集合Sから取り除く.

(2)すべての3点組について調べた後で残った点が凸包の頂点となる.

(3)残った点を凸包内部の点（たとえば重心点）に関して偏角順に並べると凸包を構成できる.

```
プログラム
for p=1 to n-2
  for q=p+1 to n-1
     for r=q+1 to n
        3点p, q, rで決まる三角形をTとする.
        for i=1 to n
           if 点iは三角形Tの内部にある  then 点iを集合から除く.
残った点のx，y座標の平均を計算することにより，重心点を求める.
残った点を重心点に関して偏角順に並べる.
```

```
#include <LEDA/window.h>

using namespace leda;
using namespace std;

point          P[1000];
int                              hull[1000];
window W (500, 500);

int
main(void)
{
    point p, lastp;
    int i, j, k, q, left, n=0, inside, d1, d2, d3;

    W.display();
    while(W >> p){
        W << p; P[n++] = p;
    }
    for(i=0; i<n; i++)
        hull[i] = 1;
```

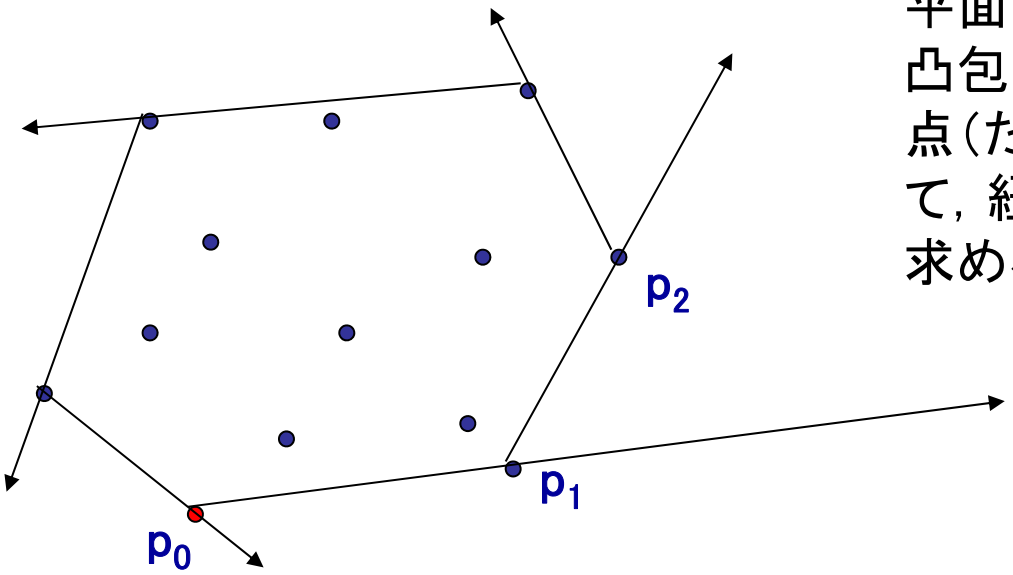
```
    for(i=0; i<n-2; i++)
        for(j=i+1; j<n-2; j++)
            for(k=j+1; k<n; k++){
                for(q=0; q<n; q++){
                        d1 = orientation(P[i], P[j], P[q]);
                        d2 = orientation(P[j], P[k], P[q]);
                        d3 = orientation(P[k], P[i], P[q]);
                         if(d1 == d2 && d2 == d3) {
                              hull[q] = 0;  // point q is not on the convex hull
                         }
                }
             }

    for(i=0; i<n; i++)
         if(hull[i] == 1) W.draw_circle(P[i], 0.5, red);

    W.read_mouse();
    return 1;
}
```

```
#include <LEDA/window.h>

using namespace leda;
using namespace std;

point          P[1000];
int                              hull[1000];
window W (500, 500);

void
draw_triangle(int i, int j, int k, color col)
{
    W.draw_segment(P[i], P[j], col);
    W.draw_segment(P[j], P[k], col);
    W.draw_segment(P[k], P[i], col);
}

int
main(void)
{
    point p, lastp;
    int i, j, k, q, left, n=0, inside, d1, d2, d3;

    W.display();
    while(W >> p){
        W << p; P[n++] = p;
    }
    for(i=0; i<n; i++)
        hull[i] = 1;
```

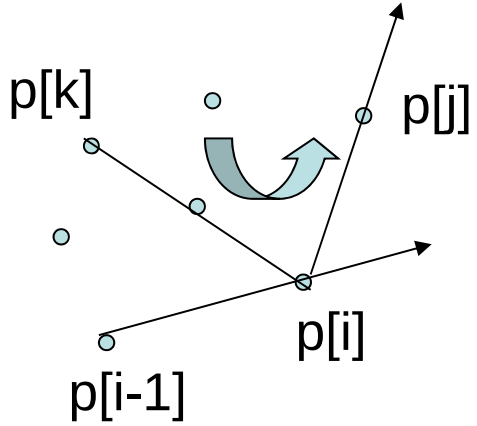
```
for(i=0; i<n-2; i++)
    for(j=i+1; j<n-2; j++)
        for(k=j+1; k<n; k++){
            draw_triangle(i, j, k, black);
            for(q=0; q<n; q++){
                d1 = orientation(P[i], P[j], P[q]);
                d2 = orientation(P[j], P[k], P[q]);
                d3 = orientation(P[k], P[i], P[q]);
                if(d1 == d2 && d2 == d3) {
                    hull[q] = 0;  // point q is not on the convex hull
                    W.draw_circle(P[q], 0.5, blue);
                }
            }
            W.read_mouse();
            draw_triangle(i, j, k, white);
    }

    for(i=0; i<n; i++)
        if(hull[i] == 1) W.draw_circle(P[i], 0.5, red);

    W.read_mouse();
    return 1;
}
```

凸包構成アルゴリズム(2)

Jarvisのアルゴリズム（ギフト包装法）



平面に釘が打たれているとする．凸包上にあることが分かっている点（たとえば最も下の点）から始めて，紐をかけていく要領で凸包を求める．

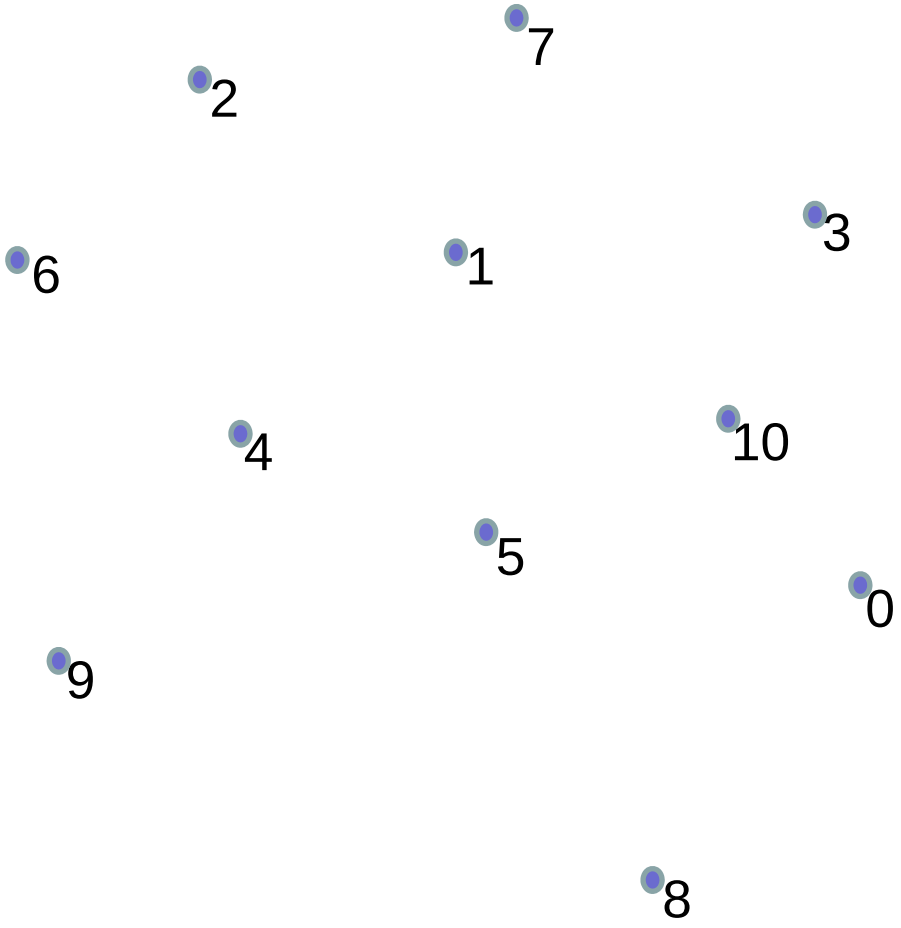


点p[i+1]の求め方：

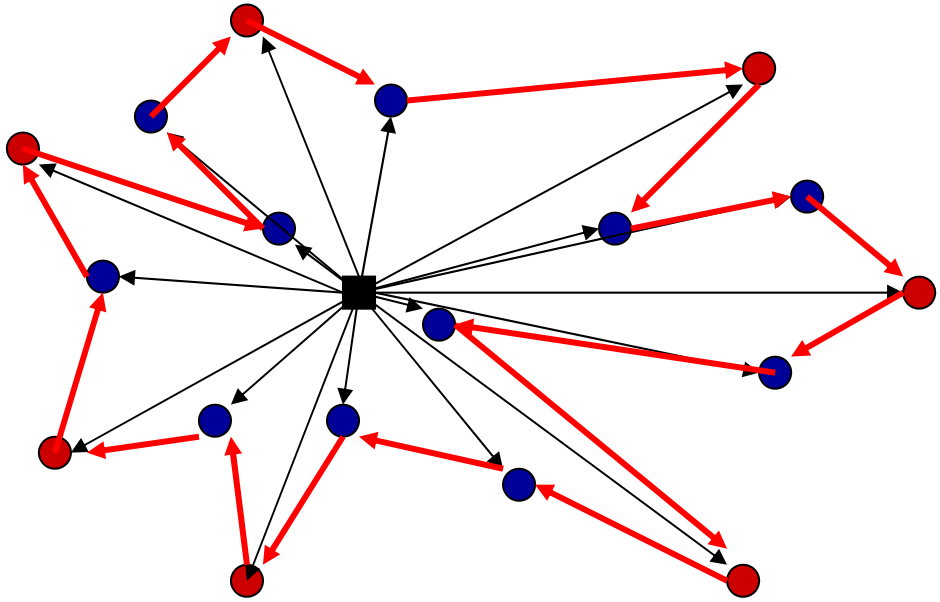
任意のp[k]に対して，(p[j], p[i], p[k])が右回りであるような点p[j]

凸包のサイズ（頂点数）をhとすると，計算時間はO(nh)となる．
h=O(n)と考えると，$O(n^2)$となる．

次の点集合に対してアルゴリズムの動作を確かめよ.



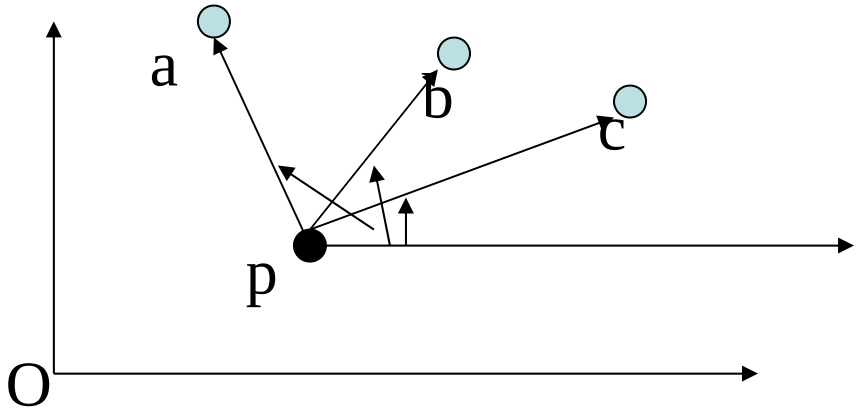
効率の良い方法



凸包内部の点（たとえば，任意の３点でできる三角形の重心）
pを取り，すべての点をpの周りに角度順にソート.
ソート順に点を調べて，凹部の点を順次取り除く.

この説明だけではアルゴリズムを実装するのは難しい.

点pの周りに点を角度順にソートするにはどうするか？



$a(x_a, y_a)$, $b(x_b, y_b)$, $c(x_c, y_c)$
$p(x, y)$

$$\theta_a = \tan^{-1}\frac{y_a - y}{x_a - x}, \theta_b = \tan^{-1}\frac{y_b - y}{x_b - x}, \theta_c = \tan^{-1}\frac{y_c - y}{x_c - x},$$

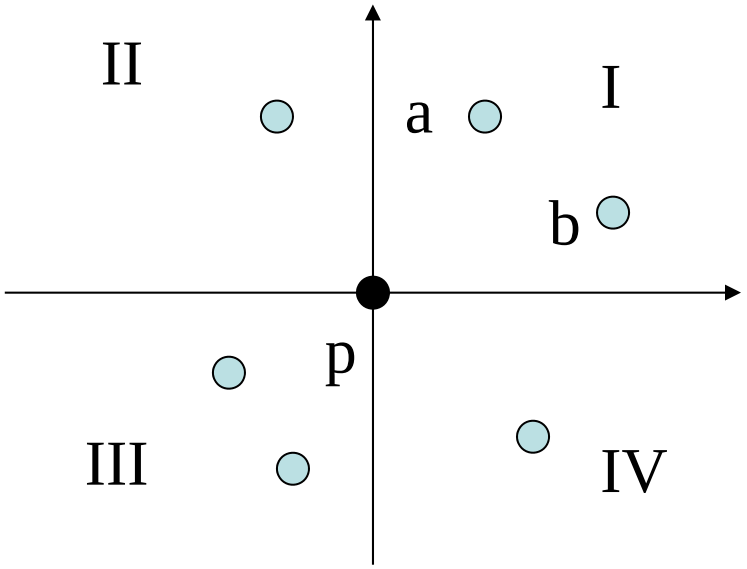
角度表現が得られたら，後はソートするだけ．

しかし，この方法では除算を使うので計算誤差が心配．

練習問題: 計算誤差のせいで角度順のソートが難しいような３点を与えよ．

計算誤差に強い方法

考え方： 三角形の符号付き面積を用いる



最初に，点を4つの象限に分類することで大雑把にソート

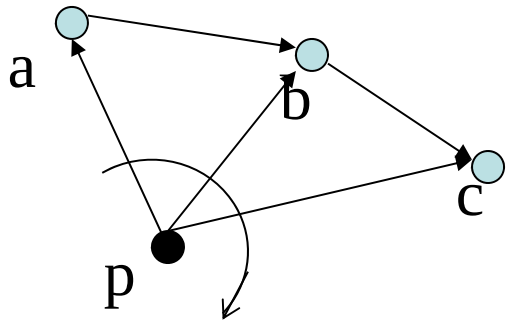
第一象限の２点 a と b について

if (p,a,b) が時計回り

then angle(a) > angle(b)

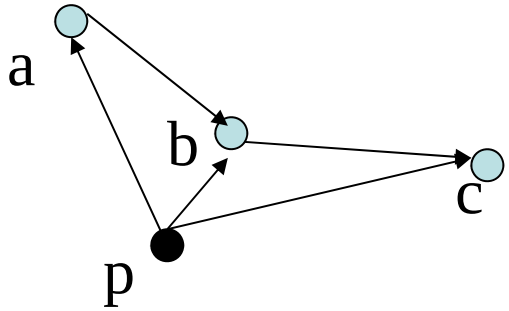
３点が凹部を作るか？

１点pの回りに時計回りの順に並んだ3点(a, b, c)について，
３点が凹部を成すような配置になっているか？



(a,b,c) は凸か?

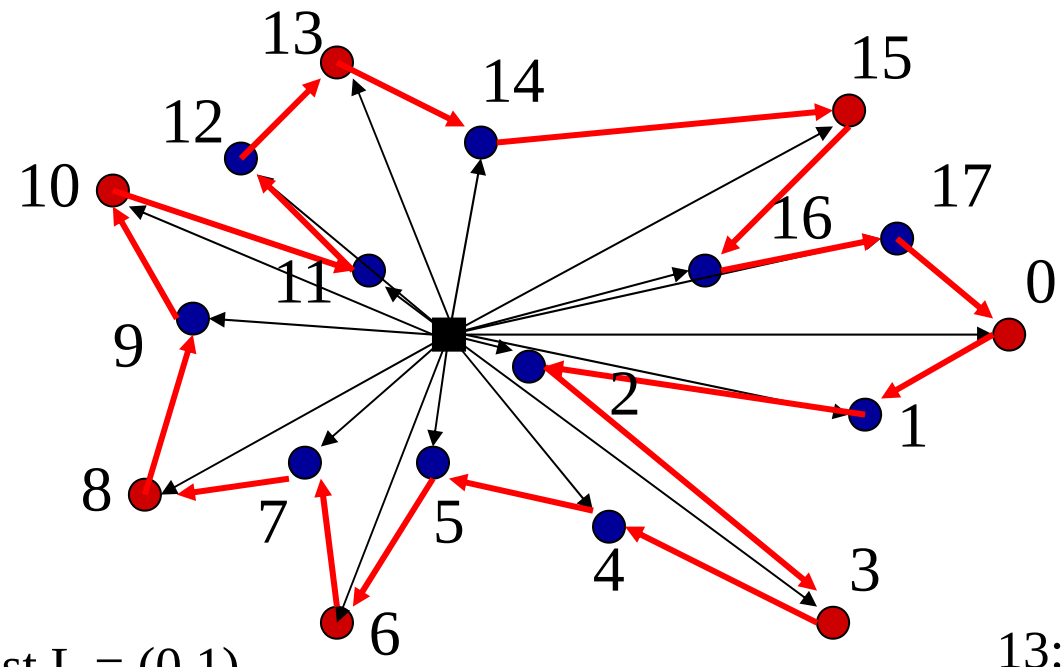
(a,b,c) が凸である
⇐ (a, b, c) が時計回りの順



(a,b,c) が凹である
⇐ (a,b,c) が反時計回りの順

Grahamの方法 (R. Graham, 1972)

入力: n点からなる点集合S
凸包CH(S) 内部にある点 p を選ぶ
　(最初の３点で作られる三角形の重心点を計算する)
集合Sの点を点pの周りの角度順にソート
　(点をpの周りに時計回りの順にソート)
最大のx座標をもつ点 p_0 を求める.
($p_0, p_1, p_2, \ldots, p_n = p_0$) をソート列とする.
L: 点のリスト(p_0, p_1).
for i=2 to n do{
　do{
　　a と ***b*** をリスト***L*** の最後の２点とする (***b*** が最後の点);
　　if ***(a,b, p_i)*** が凸 then ***b*** を***L***から削除;
　} until(***(a, b, p_i)*** は凸);
　点***p_i*** を ***L***の末尾に挿入;
}



練習問題:
x座標最大の点を最初に選ぶのはなぜか？

list L = (0,1)
2: add 2 (0,1,2)
3: delete 2
delete 1
add 3 (0,3)
4: add 4 (0, 3,4)
5: delete 4
add 5 (0,3,5)
6: delete 5
add 6 (0,3,6)

7: add 7 (0,3,6,7)
8: delete 7
add 8 (0,3,6,8)
9: add 9 (0,3,6,8,9)
10: delete 9
add 10 (0,3,6,8,10)
11: add 11 (0,3,6,8,10,11)
12: delete 11
add 12 (0,3,6,8,10,12)

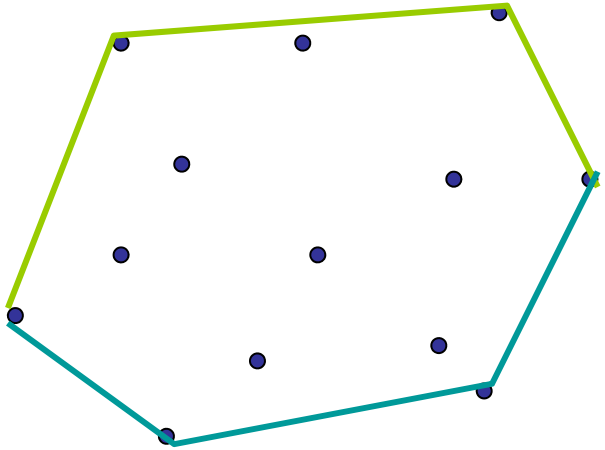
13: delete 12
add 13 (0,3,6,8,10,13)
14: add 14 (0,3,6,8,10,13,14)
15: delete 14
add 15 (0,3,6,8,10,13,15)
16: add 16 (0,3,6,8,10,13,15,16)
17: delete 16
add 17 (0,3,6,8,10,13,15,17)
0: delete 17
add 0 (0,3,6,8,10,13,15,0)

定理: Grahamの方法では n 点から成る点集合を O(n log n) の時間とO(n) のメモリで構成できる.

証明:
ソーティングの部分はO(n log n) 時間でできる.
メインループでの繰り返し回数は O(n)
　毎回の繰り返しでは
　　点を削除するか，あるいは
　　点をリストに挿入する
　注意：どの点も1度しか削除されない．また，リストへの追加も各点高々1回のみ.
　　よって，繰り返し回数は O(n).
必要なメモリは O(n).

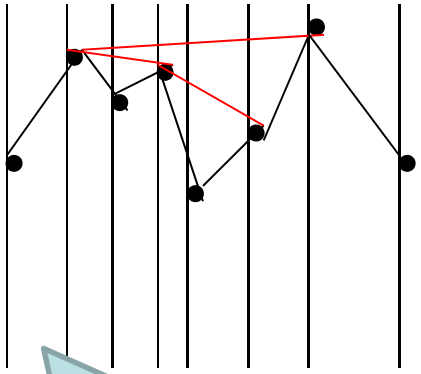
凸包構成アルゴリズム(3)

点のソーティングに基づく方法



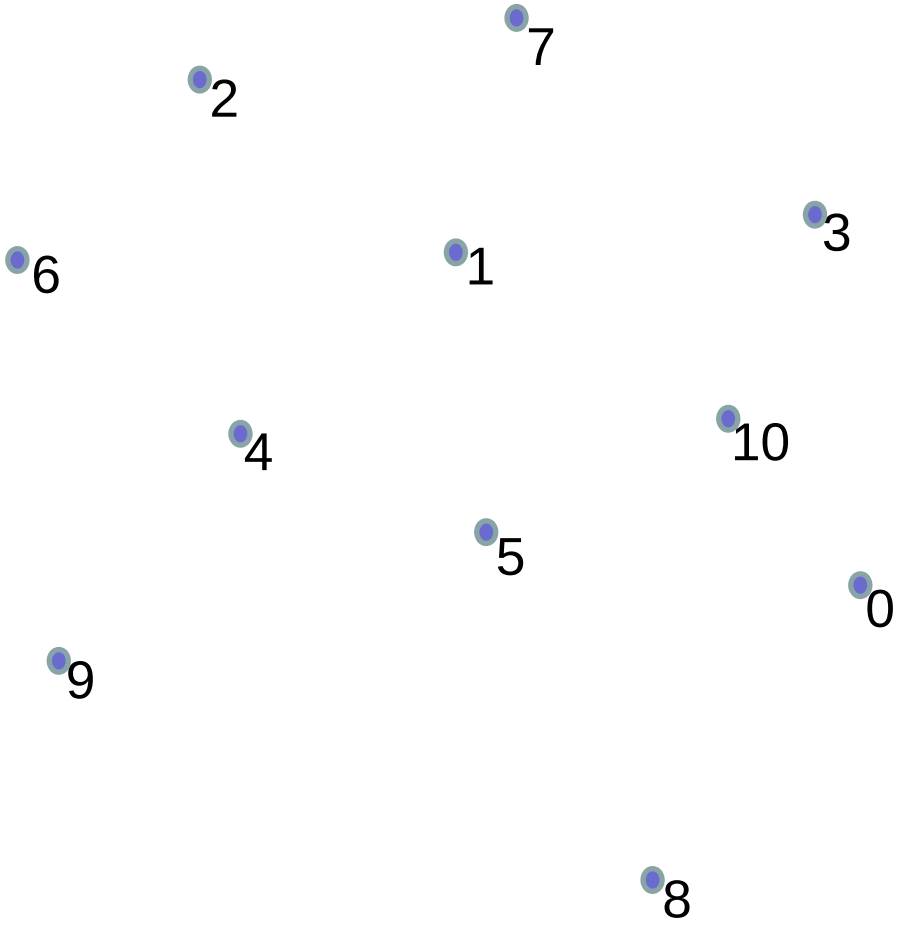
(1)x座標の昇順に点を整列.
(2)順に見ることによって
　　上側凸包を構成.
(3)同じ要領で，下側凸包を構成.

上側凸包上では，
連続する3点は右回りの順でなければならない.
もし左回りになっていれば，真ん中の点を除去する.
この操作を繰り返すと，上側凸包が構成できる.



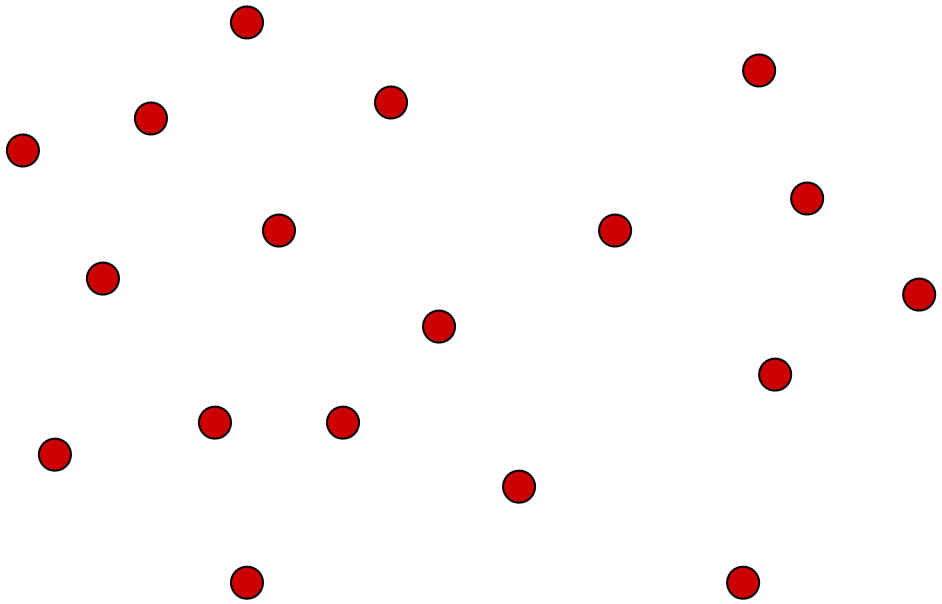
ちょっと休憩…

次の点集合に対してアルゴリズムの動作を確かめよ.

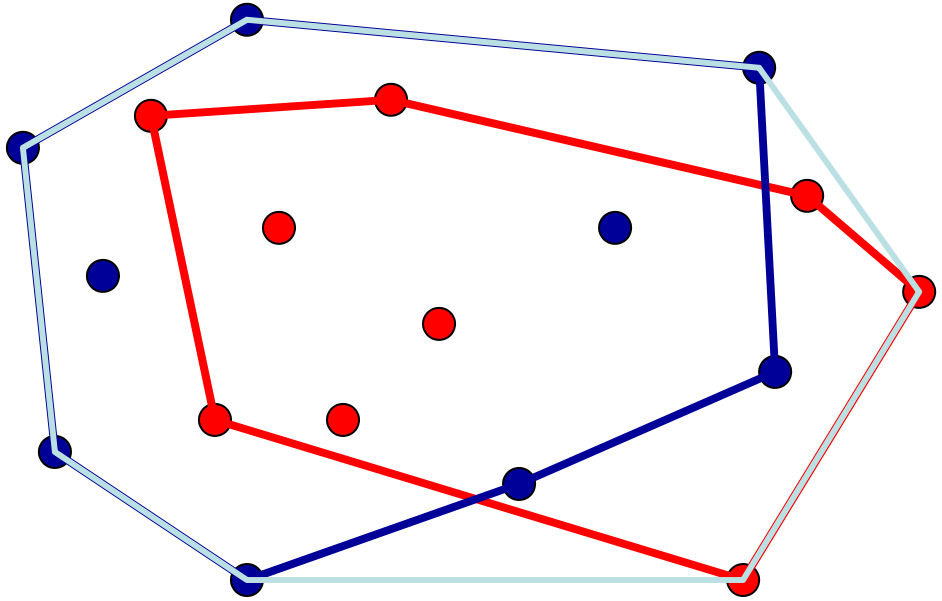


分割統治法に基づく凸包構成アルゴリズム

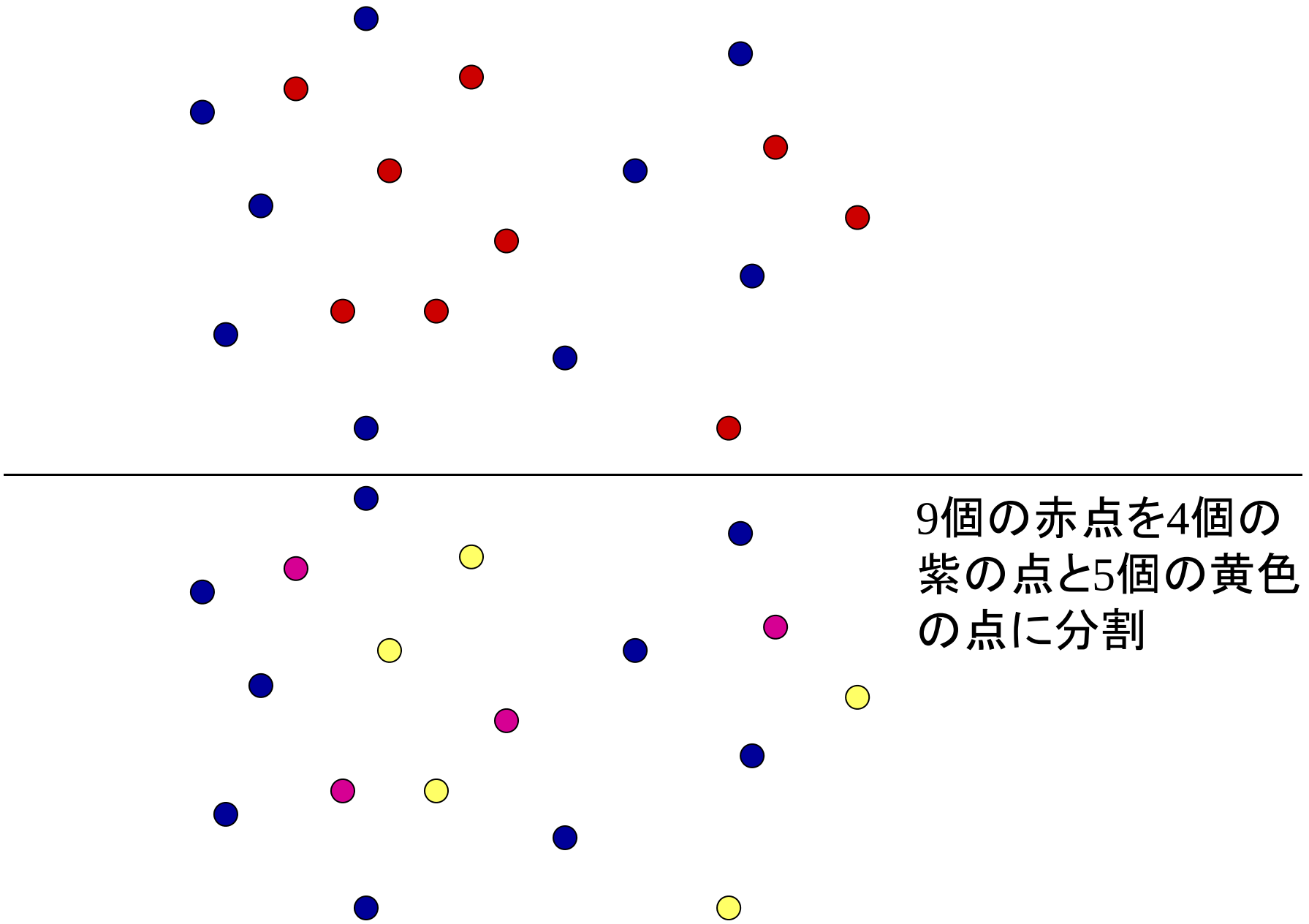
分割統治法
- 分割： ほぼ同じサイズの2つの部分集合に分割
- 再帰：それぞれの部分集合に対する凸包を再帰的に構成.
- 統治： 得られた凸包を一つの凸多角形に統合する
  (2つの凸包を含む最小の凸多角形を求めること).



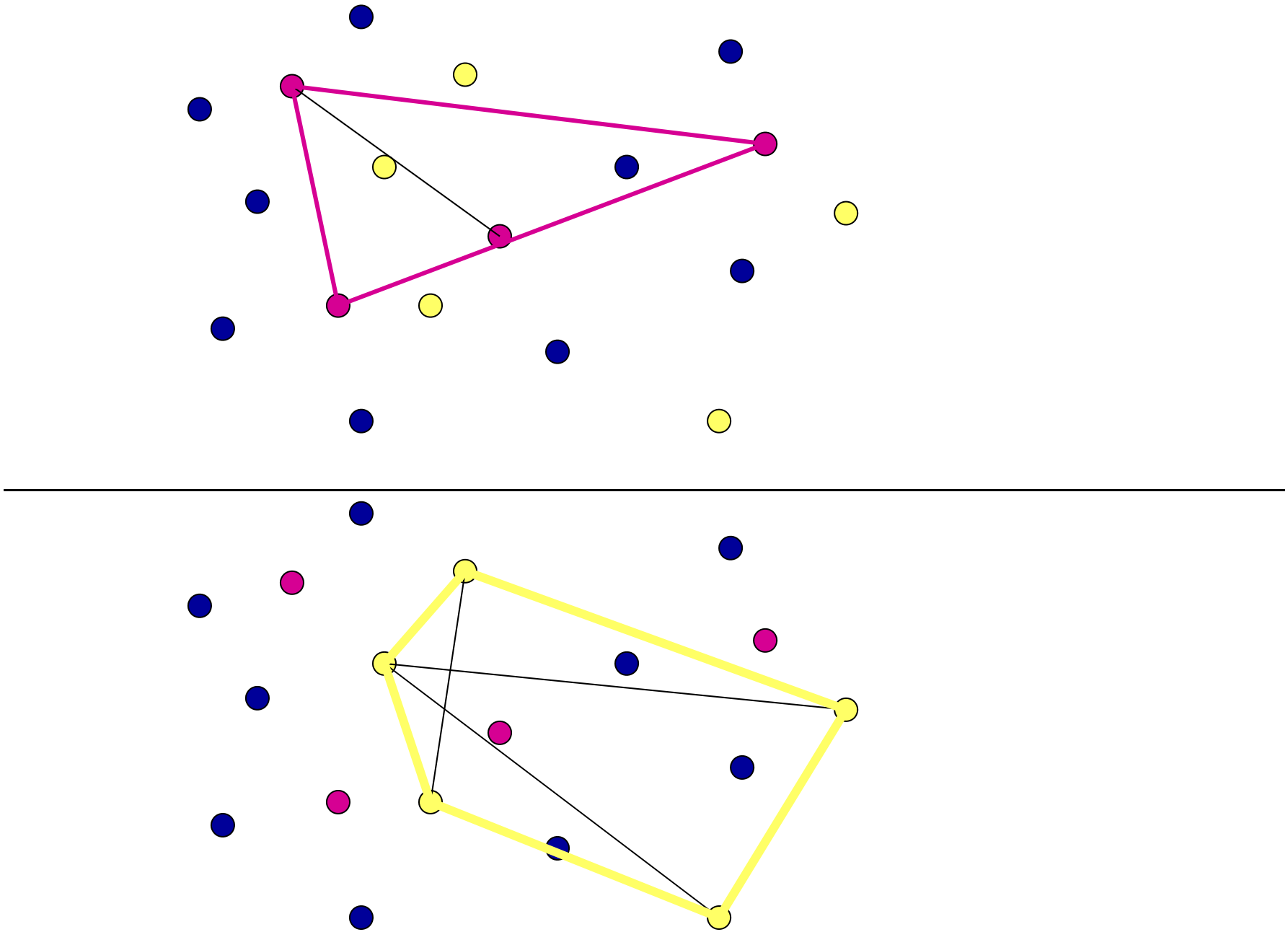
18点

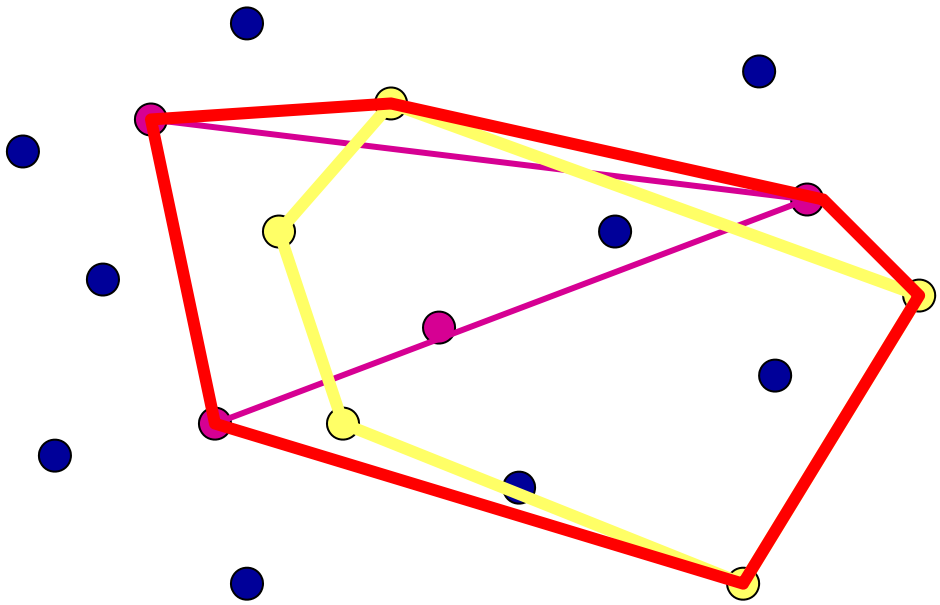


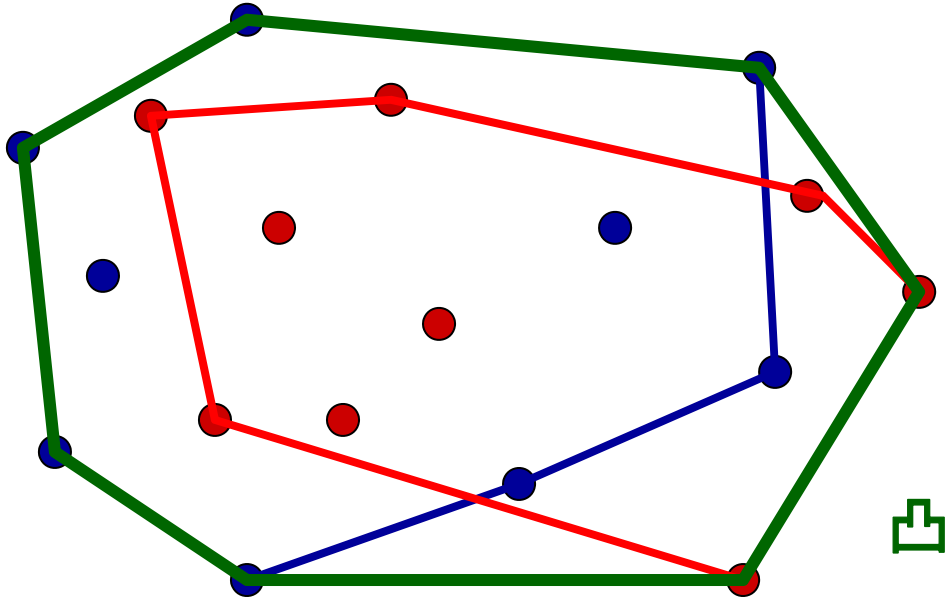
9個の赤点と
9個の青点



9個の赤点を4個の紫の点と5個の黄色の点に分割





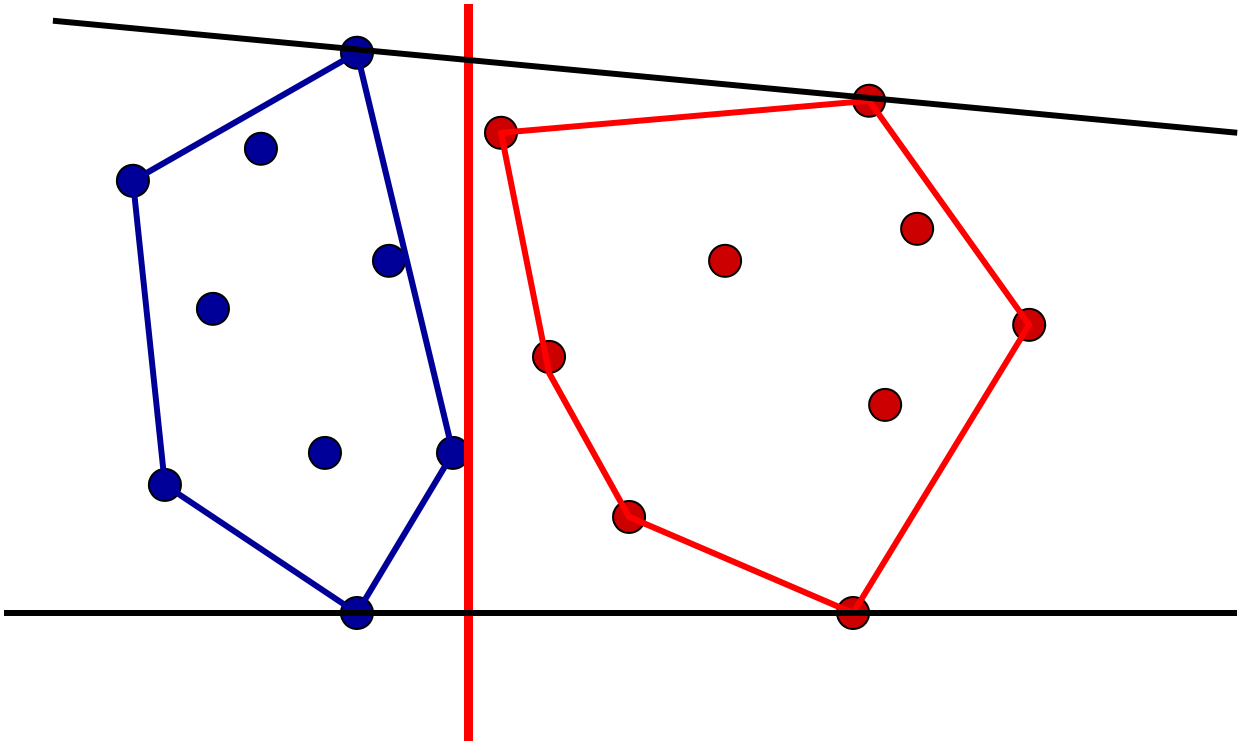

凸包

定理: 分割統治法のアルゴリズムはn点からなる点集合の凸包を $O(n \log n)$ の時間と $O(n)$ のメモリで構成する.

分割部はO(n) 時間でできる.
再帰部は 2T(n/2) 時間でできる.
統合部は O(n) でできる.
　再帰部では，点は既にソートされているので，２つのソート列の
　マージは線形時間でできることに注意.
　ソートの後の走査はO(n) 時間でできる.
よって，次の漸化式とその解を得る.
　T(n) = 2T(n/2) + O(n) ➔ T(n) = O(n log n)

分割統治法によるアルゴリズム(2)

1. 点集合を，それらの点のx座標の中央値を通る垂直線
   によって2分割する.
2. それぞれの部分集合に対する凸包を再帰的に計算する.
3. 出来上がった2つの凸包を一つの凸多角形に統合する.
   (統合のために，2本の外接線を求める)



n 個の値の中央値を求めるアルゴリズム

1. n 個の値をO(n log n)時間でソートし，中央にある値を求める.
2. n個の値の集合をサイズ5のグループに分け，それぞれのグループで中央値を求める（O(n)時間でできる）.
   次に，それらn/5個のグループ中央値の中央値*x*を再帰的に求め，*x*より大きい要素の数gを求める.
   if $g \geqq n/2$ then (全体の中央値は*x*より大きい筈)
   A を*x* より大きい要素の集合とする.
   Aにおいてn/2番目に大きい値を再帰的に求め，それを返す.
   else
   A を*x* 以下の要素の集合とする.
   Aにおいてn/2番目に小さい値を再帰的に求め，それを返す.

上記のアルゴリズムで，集合 A のサイズは高々3n/4.
よって，次式を得る.

T(n) = T(n/5) + T(3n/4) + O(n),

これより，T(n) = O(n)を得る.

例題: 24個の値の場合

S = {12,56,43,22,31,25,57,75,45,33,39,85,37,44,19,28,18,23,92,73,77,28,64,35}

S = {12,56,43,22,31,25,57,75,45,33,39,85,37,44,19,28,18,23,92,73,77,28,64,35}

グループ中央値 ={31,45,39,28,35}　グループ中央値の中央値 = 35

値 > 35 = {56,43,57,75,45,39,85,37,44,92,73,77,64}　13個の要素 > 24/2

　全体の中央値はこの集合にあるはず

この集合で12番目に大きい値を再帰的に求める

　S = {56,43,57,75,45,39,85,37,44,92,73,77,64}

　グループ中央値 = {56,44,73},グループ中央値の中央値= 56

　値 > 56 = {57,75,85,92,73,77,64}　7個の要素 > 13/2

　12番目に大きい値はこの集合には含まれない

残りの値の集合で(12-7)番目に大きい要素を求める

　S = {56,43,45,39,37,44}　5番目に大きい値 = 39

全体の中央値は 39

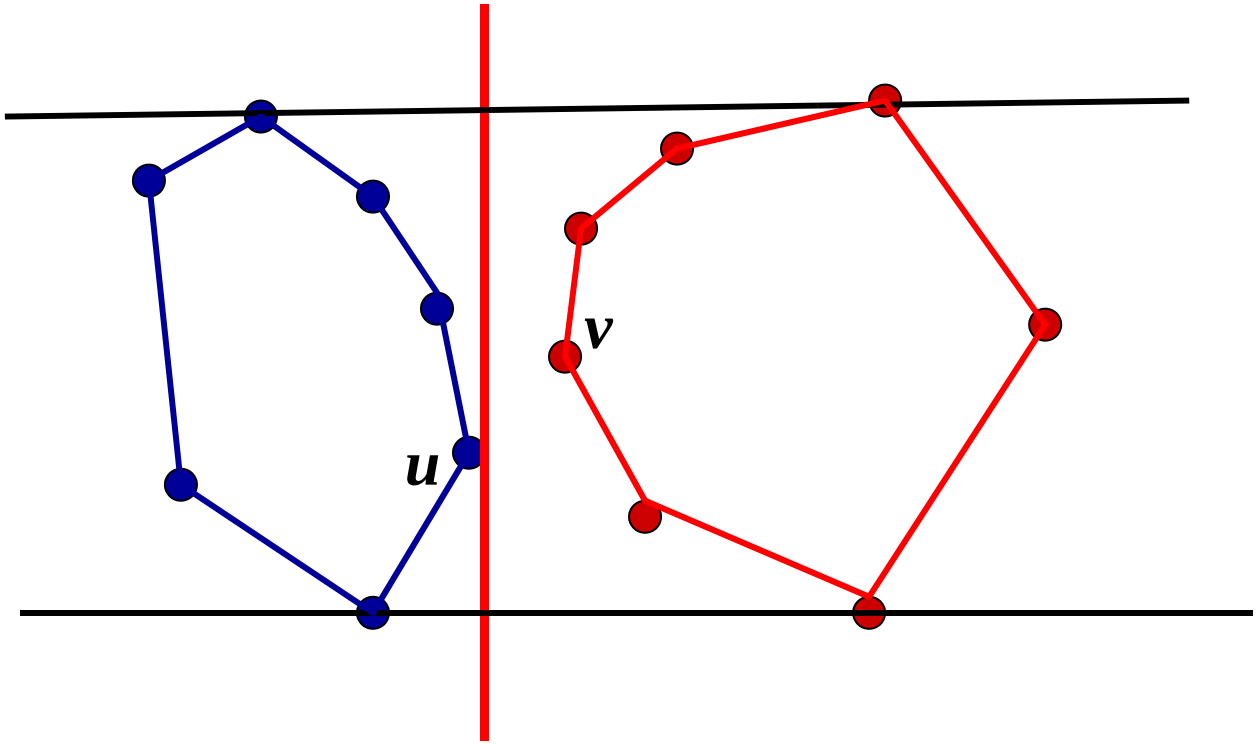
実際,

　> 39:　56,43,57,75,45,85,44,92,73,77,64,

　　39

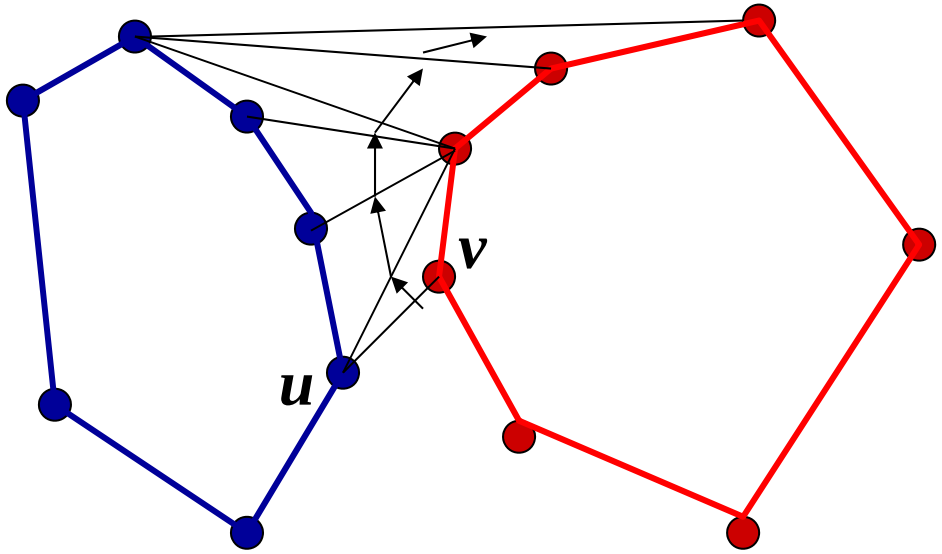
　< 39:　12,22,31,25,33,37,19,28,18,23,28,35

2つの凸包の統合



左の凸多角形で最も右の頂点 u と,
右の凸多角形で最も左の頂点 v
を求める.

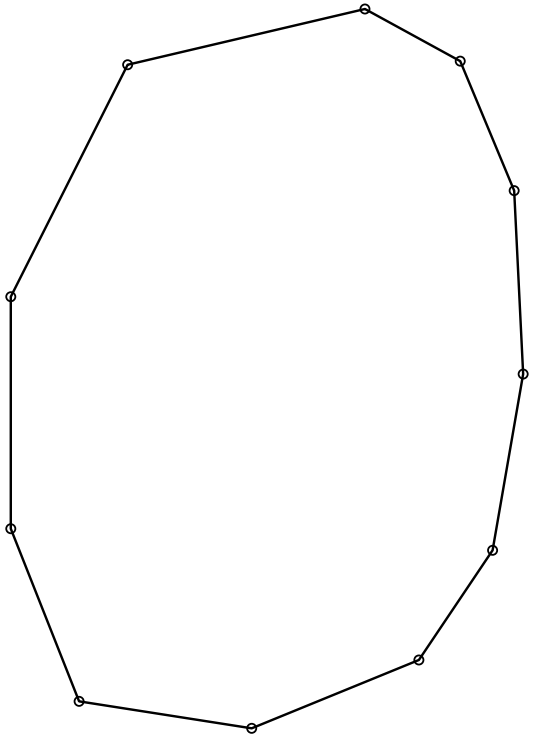
点対 (u,v)　から始めて,
　上部接線に到達するまで上へ上へと点対を更新し，次に,
　下部接線に到達するまで下へ下へと点対を更新していく.

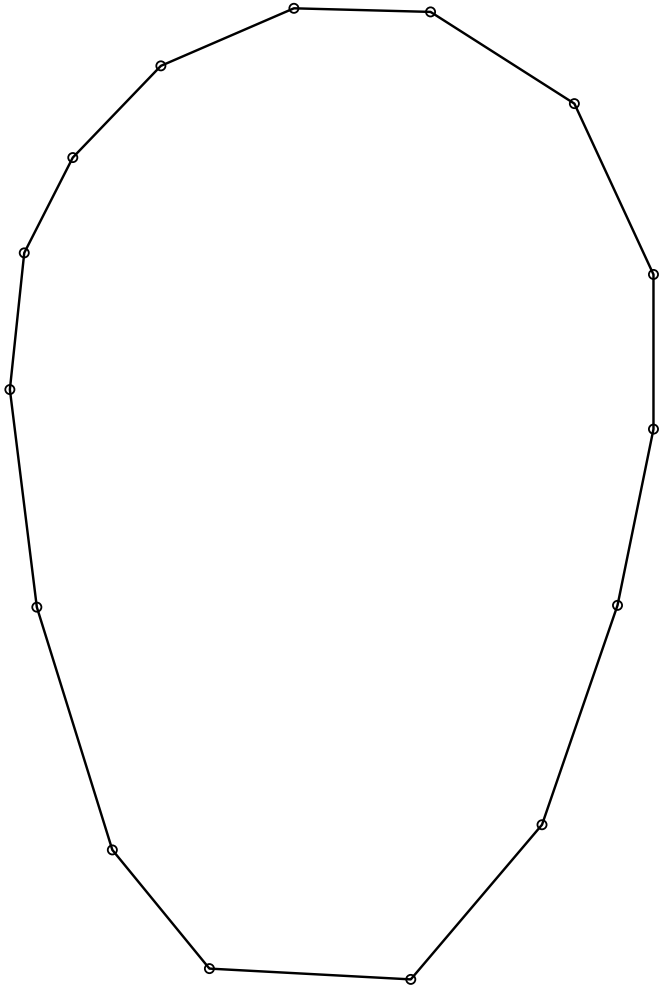


この操作は線形時間で実行できる.
なぜなら，毎回どちらかの頂点が移動するから.

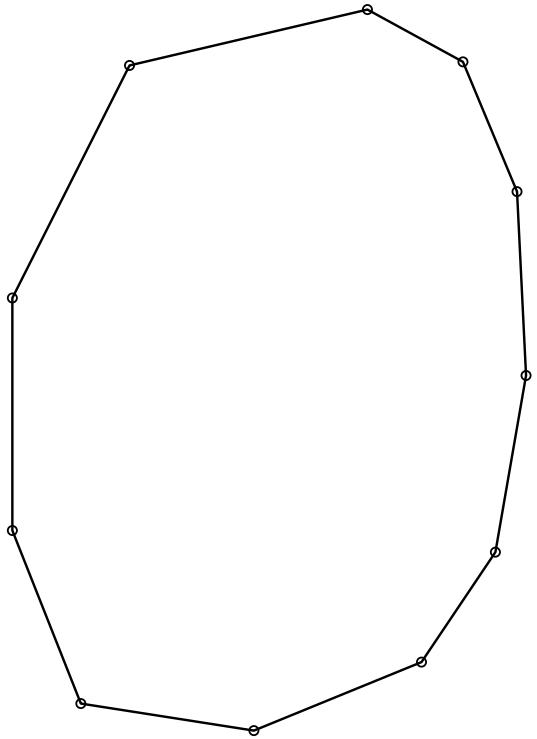
直線(u,v) がvを上部接点とする接線であるのは,
　(u, v, t)が時計回りの順になっているとき.
　ただし，tは反時計回りの順でvの次の点
直線(u,v) がuを上部接点とする接線であるのは,
　(v, u, w)が時計回りの順になっているとき.
　ただし，wは時計回りの順でuの次の点

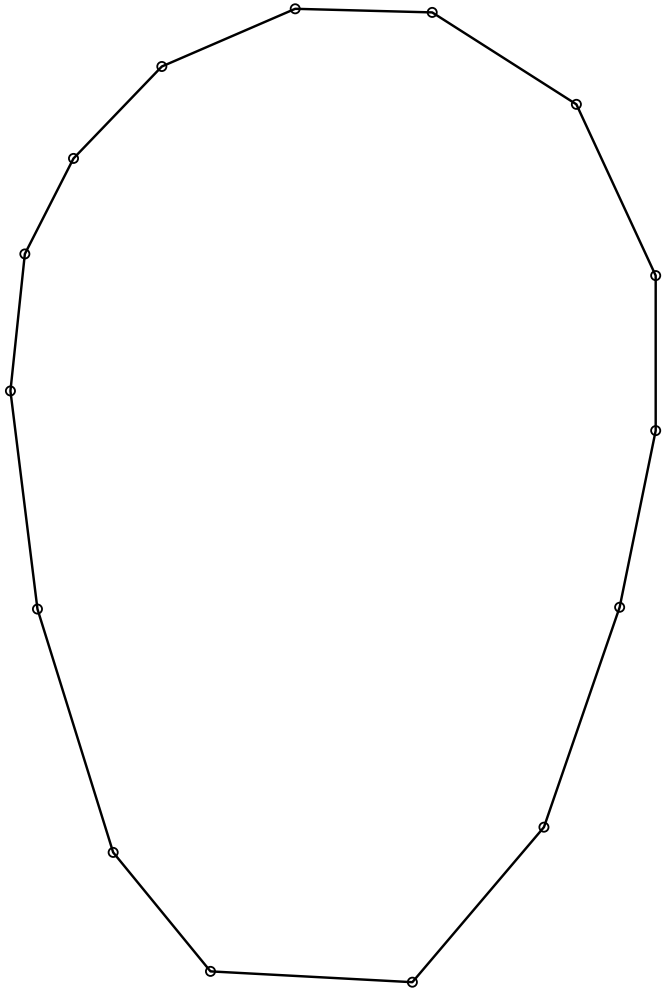
次の2つの凸多角形について前頁のアルゴリズムの動作を調べよ.





次の2つの凸多角形について前頁のアルゴリズムの動作を調べよ.





アルゴリズムの解析

1. 点集合を，それらの点のx座標の中央値を通る垂直線
   によって2分割する.
2. それぞれの部分集合に対する凸包を再帰的に計算する.
3. 出来上がった2つの凸包を一つの凸多角形に統合する.
   (統合のために，２本の外接線を求める)

1: O(n)時間（中央値発見アルゴリズム）
2: 2T(n/2)時間
3: O(n)時間
よって，次式を得る.

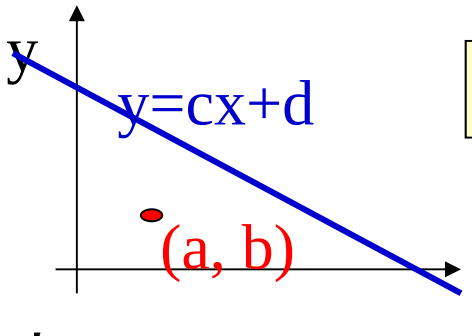
$$T(n) = 2T(n/2) + O(n),$$

これより，次式を得る.

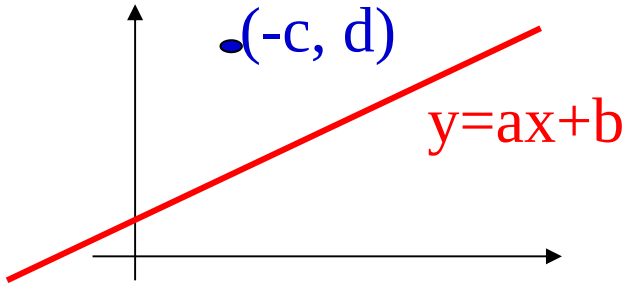
$$T(n) = O(n \log n)$$

定理: 分割統治法で凸包を求めるアルゴリズムの計算時間は $O(n \log n)$ で必要なメモリは$O(n)$ である.

双対変換に基づくアルゴリズム



双対変換

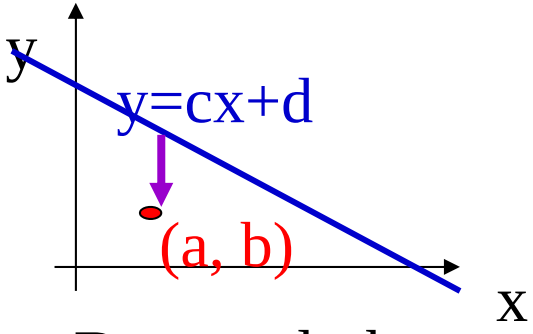


点 (a, b)
直線 y= cx + d

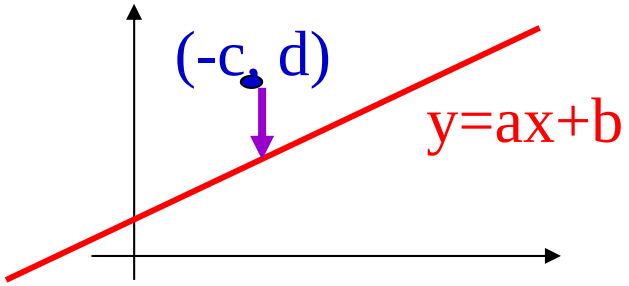
⇔

直線 y = ax + b
点 (-c, d)

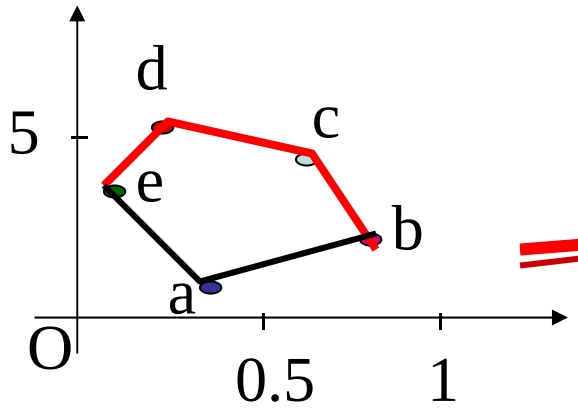
双対変換は，点と直線の間の（符号つき）垂直距離を保存する.

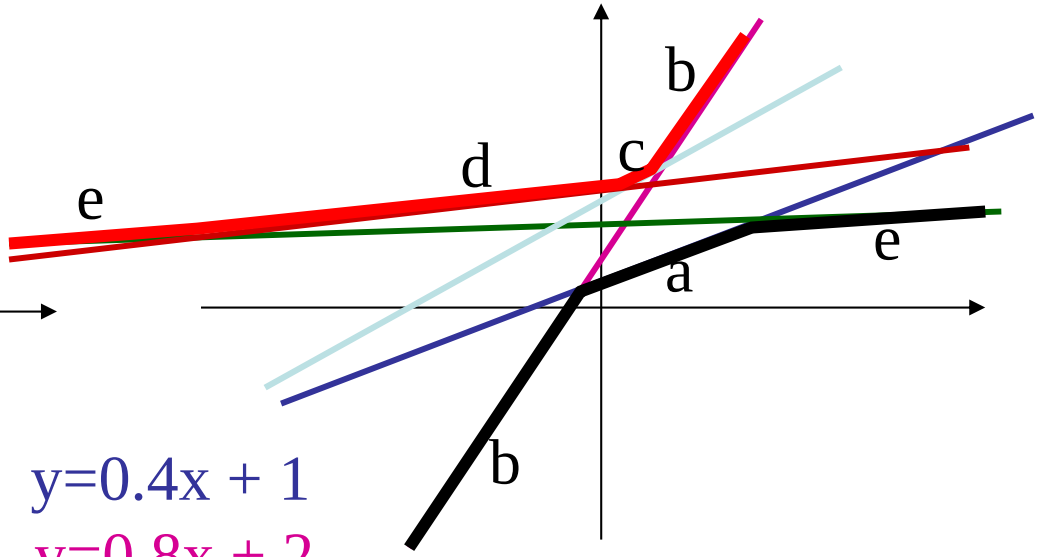


D=ca + d - b



D’=d - (-ac+b)
=ca + d - b = D





a(0.4, 1) → y=0.4x + 1

b(0.8, 2) → y=0.8x + 2

c(0.6, 4) → y=0.6x + 4

d(0.2, 5) → y=0.2x + 5

e(0.1, 3) → y=0.1x + 3

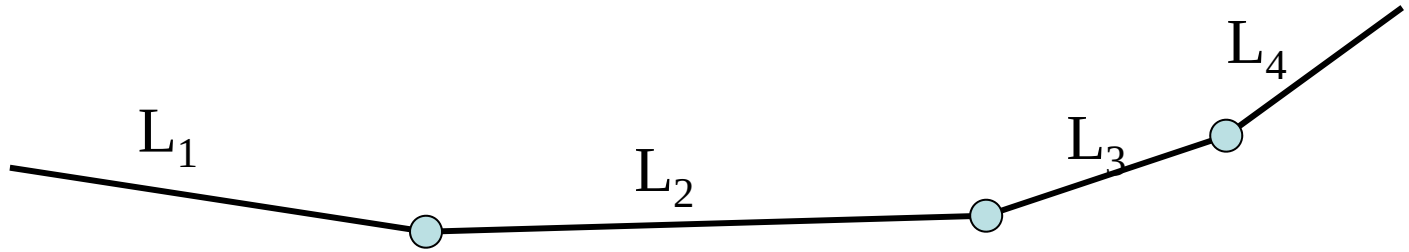
上部凸包 → 上側エンベロープ（包絡線）
上部の無限領域の境界

下部凸包 → 下側エンベロープ（包絡線）
下部の無限領域の境界

よって，問題は双対平面における直線のアレンジメントの上側と下側のエンベロープを求める問題に帰着される．

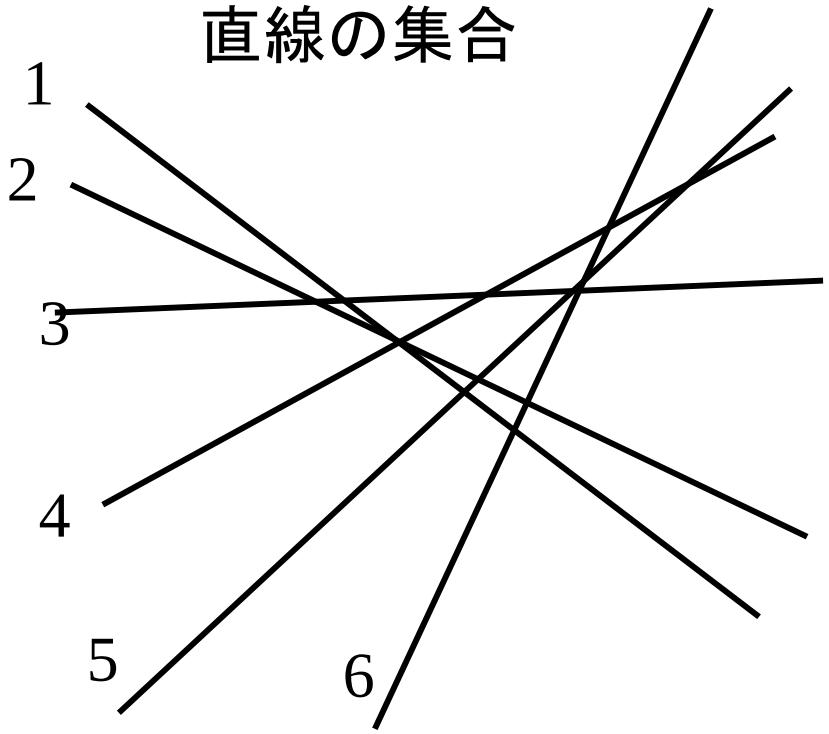
上部包絡線の計算

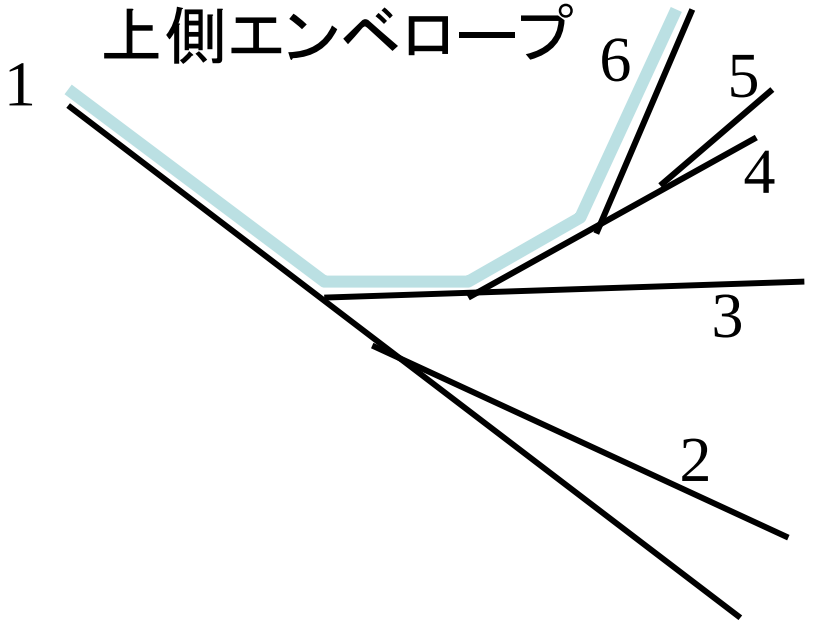
$(L_1, L_2, \ldots, L_n)$: を傾きの昇順に並べられた直線の列とする.
下のアルゴリズムでは，直線のリスト$(L_1, L_2, \ldots, L_k)$は，
左から右へと並んだ多角形の辺列を表す.




```
T=(L1);
for i=2 to n do{
   L=リストTにおける最後の直線;
   while(Tにおける線分Lが Li と交差しない）
       TからLを削除し，Lをその直前の線分で置き換える.
  直線 Li をリストTの最後尾に追加する
}
```

例





補題: n本の直線をO(n log n)時間でソートすると，上側エンベロープはO(n)時間で計算できる.

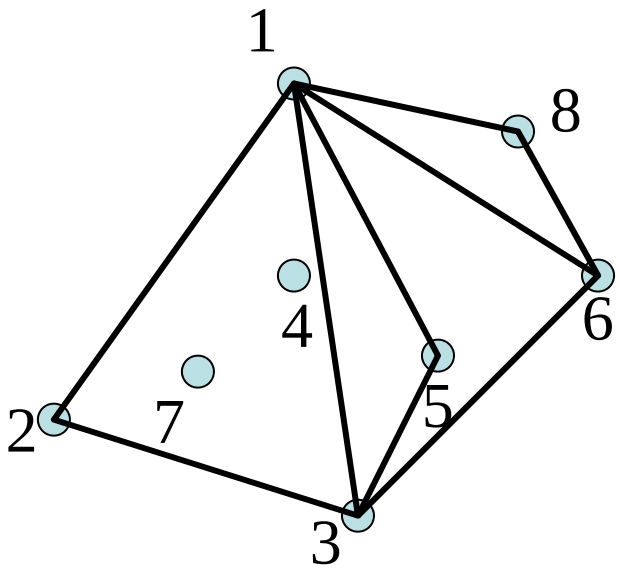
略証:
新たな直線を挿入するとき，多数の直線を調べることもあるが，
調べた直線は，最後の直線以外，すべて削除される.

逐次構成法

３点からなる凸包から始めて，１点ずつ加えながら，凸包を更新していく.

アルゴリズム 1:

CH(3) = 最初の３点で構成される凸包（三角形）;
for i=4 to n do{
　if i 番目の点 p がCH(i-1)の内部にある
　　then CH(i) = CH(i-1);
　else // p は CH(i-1)の外にある
　　pからCH(i-1) への２本の接線を求め，２本の接線の間にある点を削除することによって，CH(i-1) から CH(i) への更新を実行する.
}



困難な点:

　１点が凸多角形の内部にあるかどうかをどう判定するか.

　効率よく実行できるか？

　１点からの接線をどのように求めるか？

両方とも O(log n) 時間で実行したい. できるか？

アルゴリズム 2:

与えられた点をx座標の昇順にソートしておく.

CH(3) = 最初の３点からなる凸包（三角形）;

for i=4 to n do{

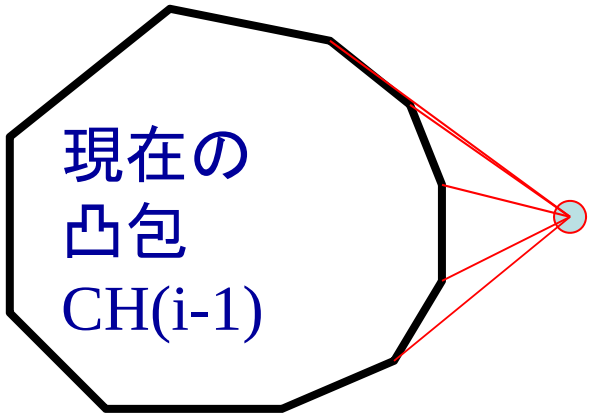
　// i 番目の点 p は常に CH(i-1)の外部にある

　pから CH(i-1) への２本の接線を求めよ.

　　２本の接線の間にある点を削除することによって,

　　CH(i-1) から CH(i) への更新を実行する.

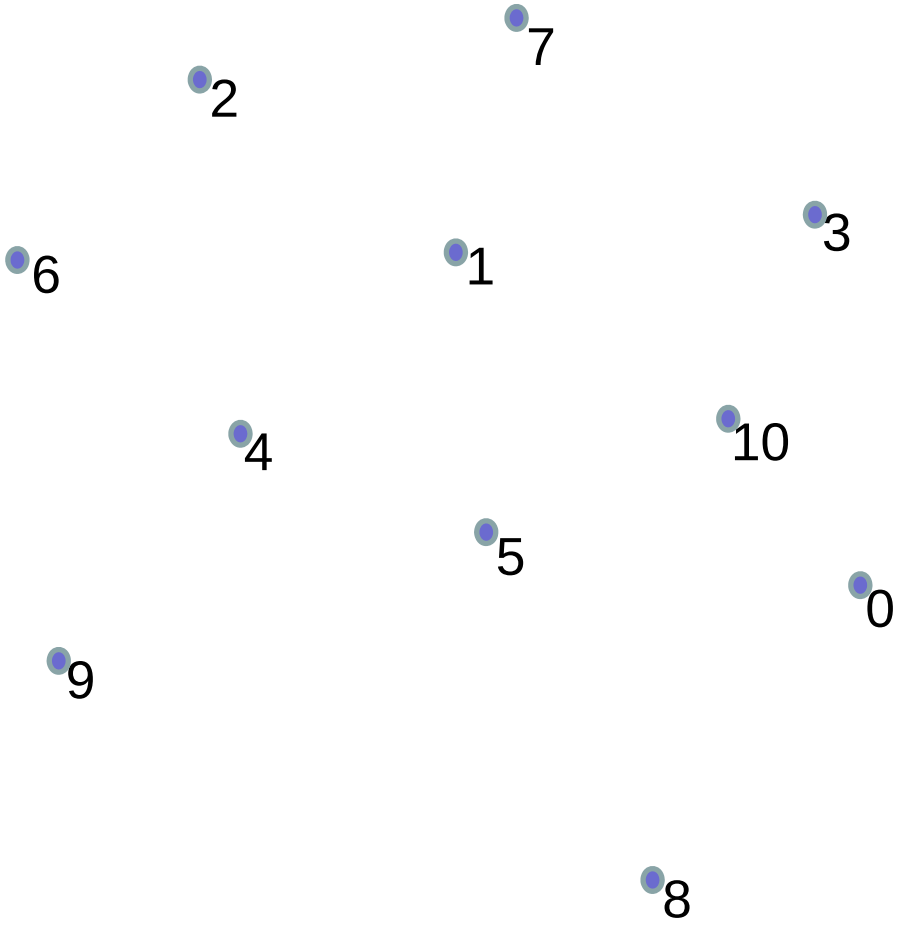
}



現在の凸包を更新するには,
直前の点p_{i-1}から接点に至るまで
凸包の境界上を上と下に移動する.

定理: アルゴリズム2は，$O(n \log n)$ の計算時間と $O(n)$ のメモリで，n点の凸包を求める.

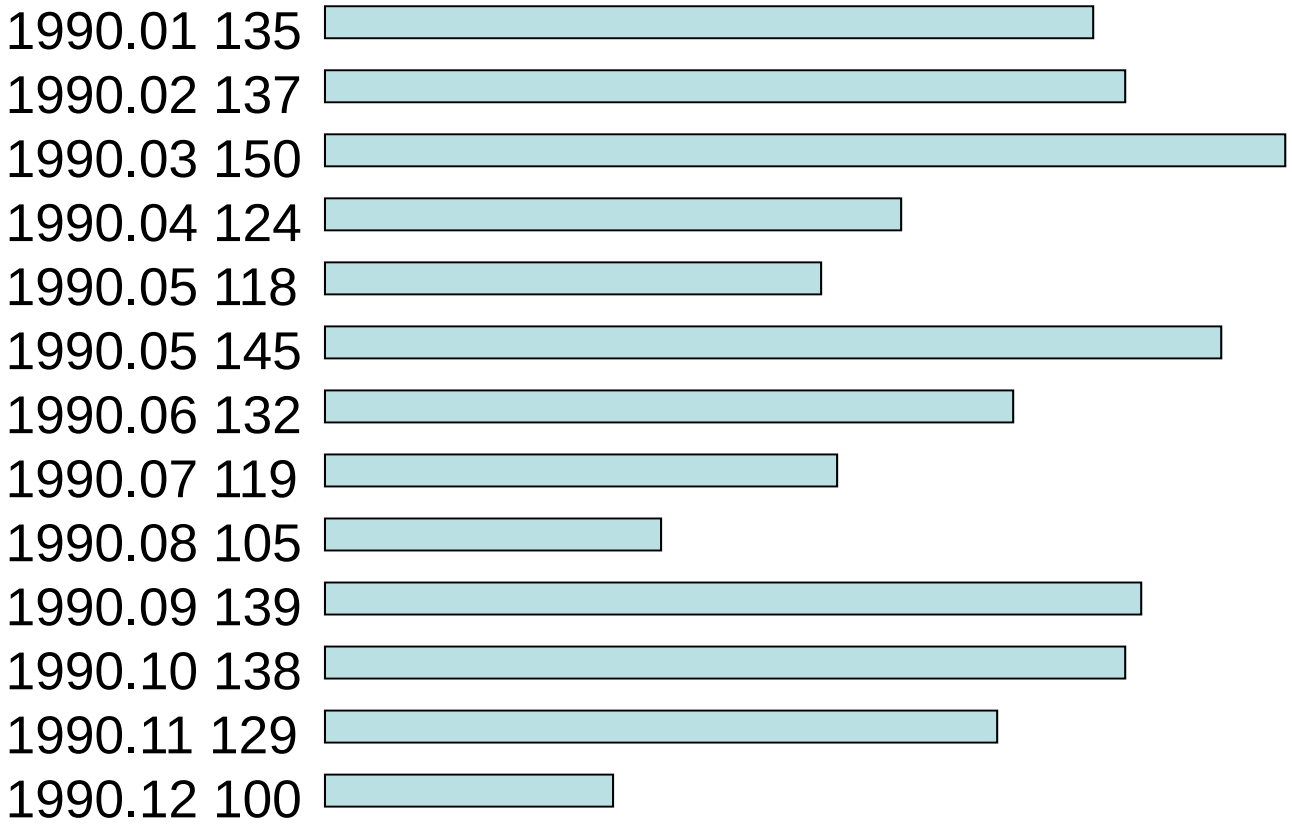
次の点集合に対してアルゴリズムの動作を確かめよ.



付録

アルゴリズムの記述

株で儲ける方法！



何月に買って何月に売れば利益が最大になるか？

sp[ ]: 株価を蓄える配列

　　sp[0]からsp[n-1]に株価が順に蓄えられているとする.

i月に買って，j月に売ると

　買値：sp[i]

　売値：sp[j]

　利益：sp[j]-sp[i]　　これを最大にしたい.　ただし，i<j.

アルゴリズム１：

```
for i=0 to n-2
    for j=i+1 to n-1
        利益sp[j]-sp[i]を求める;
```

アルゴリズム２：

```
for j=1 to n-1
    for i=0 to j-1
        利益sp[j]-sp[i]を求める;
```

株式投資における最大売却益:(A)方式

```
入力:毎月の株価sp[0], ... , sp[n-1].
mxp = 0; // 利益の最大値 mxp を0に初期化
for i=0 to n-2
   for j=i+1 to n-1{
      d = sp[j]-sp[i]; //売却益 = 売値 - 買値
      if d > mxp then mxp = d;
            // 今まで以上の売却益であればmxpの値を更新する
   }
mxpを答として返す;
```

改良1:

買った月 i を固定すると，i 月以降で値段が最大になったときが最良の売り月 j であるから，毎回引き算をしなくても，最大値を求めるだけでよい.（これで引き算の回数が減る）

株式投資における最大売却益:(A)方式

```
入力：毎月の株価sp[0], ... , sp[n-1].
mxp = 0; // 利益の最大値 mxp を0に初期化
for i=0 to n-2{
   mxsp = sp[i];          // 株価の最高値 mxsp を sp[i]に初期化
   for j=i+1 to n-1
       if sp[j] > mxsp then mxsp = sp[j];
   d = mxsp - sp[i];    //売却益=買値とそれ以降の最高値との差
   if d > mxp then mxp = d;
                  // 今まで以上の売却益であればmxpの値を更新する
}
mxpを答として返す;
```

株式投資における最大売却益:(B)方式

```
入力:毎月の株価sp[0], ... , sp[n-1].
mxp = 0;                    //利益の最大値 mxp を0に初期化
for j=1 to n-1{
    mnsp = sp[j];           // 株価の最安値 mnsp を sp[i]に初期化
    for i=0 to j-1
        if sp[i] < mnsp then mnsp = sp[i];
    d = sp[j] - mnsp;       //売却益=売値とそれ以前の最安値との差
    if d > mxp then mxp = d;
                    // 今まで以上の売却益であればmxpの値を更新する
}
mxpを答として返す;
```

アルゴリズムの効率

それぞれのアルゴリズムにおける繰り返し回数を評価

方式(A):

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}n-i-1=(n-1)^2-\frac{1}{2}(n-2)(n-1)$$

$$=\frac{1}{2}(n^2-n)\le\frac{1}{2}n^2$$

よって，方式(A)は$O(n^2)$

方式(B):

$$\sum_{j=1}^{n-1}\sum_{i=0}^{j-1}1=\sum_{j=1}^{n-1}j=\frac{1}{2}n(n-1)\le\frac{1}{2}n^2$$

よって，方式(B)も$O(n^2)$

$O(n^2)$: n^2に比例する程度の量，ビッグオーn^2，またはn^2のオーダ

アルゴリズムの改善

(A)方式では，各 i に対して，i 月以降での株価の最大値を求める
　　MAX[i, n-1]: i 月からn-1月までの最大値とすると，
　　(A)方式では，MAX[1,n-1], MAX[2, n-1], ... の順に計算
　　MAX[i-1,n-1]の値がMAX[i,n-1]の値に役立つか？

NO!

(B)方式では，各 j に対して，j 月以前での株価の最安値を求める
　　MIN[0, j-1]: 0 月からj-1月までの最安値とすると，
　　(B)方式では，MIN[0,0], MIN[0,1], MIN[0,2] ... の順に計算
　　MIN[0,j-1]の値がMIN[0,j]の値に役立つか？

YES!

MIN[0, j] = min{ MIN[0, j-1], sp[j]}
配列を探索しなくても，１回の比較だけで十分.

株式投資における最大売却益(改良版)

入力 : 毎月の株価sp[0], ... , sp[n-1].

```
mxp = 0;          // 利益の最大値 mxp を0に初期化
msf = sp[0];      // これまでの最安値msfをsp[0]に初期化
for j=1 to n-1{
    d = sp[j] - msf;      // 売却益
    if d > mxp then mxp = d;
          // 今まで以上の売却益であればmxpの値を更新する
    if sp[j] < msf then msf = sp[j];
         // これまでの最安値の更新
}
mxpを答として返す;
```

上記のアルゴリズムの計算時間は明らかにO(n)である.

線形配列上での直近上位要素

問題: n個の実数値が読み出し専用の配列A[1..n]に蓄えられているとする．各要素A[i]に対して，A[i]より大きな値をもつ要素の中でインデックスの意味でA[i]に最も近い要素（直近上位要素）を求めよ．

	1	2	3	4	5	6	7	8	9	10
A	87	32	12	54	28	35	14	61	18	53
NLN	-10	1	2	1	4	4	6	1	8	8

	1	2	3	4	5	6	7	8	9	10
A	87	32	12	54	28	35	14	61	18	53
NLN	-10	1	2	1	4	4	6	1	8	8

Algorithm 1: 単純な双方向スキャン

```
for i=1 to n
    NLN[i] = -n;
    for d=1 to n-1{
        if i-d≧1 and A[i-d]>A[i] then NLN[i] = i-d; break;
        if i+d≦n and A[i+d]>A[i] then NLN[i] = i+d; break;
    }
```

$O(n^2)$ 時間，作業領域は$O(1)$

Algorithm 2: ダブルスタック法

```
スタック S を空に初期化
for i=1 to n{  // 左から右にスキャン
    while( S が空でなくA[i] ≧A[top(S)] ) pop(S);
    If ( S は空 ) LNLN[i] = -n;  else LNLN[i] = top(S);
    push A[i] onto S
}
スタック S を空に初期化;
for i=n downto 1 {  // 右から左にスキャン
    while( S が空でなくA[i] ≧A[top(S)] ) pop(S);
    if( S は空 ) RNLN[i] = -n;  else RNLN[i] = top(S);
    push A[i] onto S;
}
for i=1 to n{  // 最後のスキャン
    if( I - NLN[i] ≦RNLN[i] - I ) NLN[i] = LNLN[i]; else NLN[i] = RNLN[i];
}
```

O(n)時間, O(n)の作業領域　➔　省メモリアルゴリズム？

定理 [双方向探索の威力]
配列要素はすべて異なると仮定すると，双方向探索に
基づくアルゴリズムの計算時間はO(n log n)である.

証明

d[i] = |NLN[i] - i| i=1, …, n, を要素A[i]からその直近上位要素までの距離とする．最大値A[j]についてはd[j]=nとする.

このとき，計算時間は 2(d[1]+d[2]+… d[n]) で抑えられる.

k を整数とし，D_k = {d[i], d[j], … } をd[]の中で大きい方からk個の値の集合とし，C_k を対応するインデックスの集合とする.
そのような要素のことを重い要素と呼ぶことにしよう.

重い要素の間の最短間隔をd_k としよう

(i, j) を最も近い重い要素のペアとしよう.
どの要素も異なっているから, A[i]<A[j] または A[j]<A[i]で
ある. したがって, d[i] ≦d_k または d[j] ≦ d_k が成りたつ.
d_k は高々n/(k−1) であるから, これよりk番目に大きなd[]の
値は n/(k−1) で抑えられることが分かる.
よって, dの値の合計は次のようになる.
n + n/1 + n/2 + … n/(n−2) = O(n log n).