# Lecture Notes on Pattern Matching Algorithms

1. Introduction

Pattern matching is to find a pattern, which is relatively small, in a text, which is supposed to be very large. Patterns and texts can be one-dimensional, or two-dimensional. In the case of one-dimensional, examples can be text editor and DNA analysis. In the text editor, we have 26 characters and some special symbols, whereas in the DNA case, we have four characters of A, C, G, and T. In the text editor, a pattern is often a word, whose length is around 10, and the length of the text is a few hundred up to one million. In the DNA analysis, a gene in a few hundred long and the human genome is about 3 billion long.

    In the case of two-dimensional, the typical application is a pattern matching in computer vision. A pattern is about (100, 100) and text typically (1024,1024). Either one-dimensional or two-dimensional, the text is very large, and therefore a fast algorithm to find the occurrence(s) of pattern in it is needed. We start from a naive algorithm for one-dimensional.
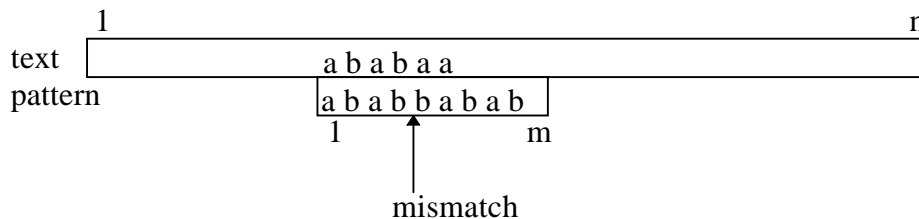
At first the pattern is set to the left end of the text, and matching process starts. After a mismatch is found, pattern is shifted one place right and a new matching process starts, and so on. The pattern and text are in arrays pat[1..m] and text[1..n] respectively.

Algorithm 1. Naive pattern matching algorithm

```
1. j:=1;
2. while j <= n-m+1 do begin
3.    i:=1;
4.    while (i<=m) and (pat[i]=text[j]) do begin
5.       i:=i+1;
6.       j:=j+1
7.    end;
8.    if i<=m then j:=j-i+2   /* shift the pattern one place right */
9.            else write("found at ", j-i+1)
10. end.
```

The worst case happens when pat and text are all a's but b at the end, such as pat = aaaaab and text = aaaaaaaaaaaaaaaaaaaaaaaaaaab. The time is obviously O(mn). On average the situation is not as bad, but in the next section we introduce a much better algorithm. We call the operation pat[i]=text[j] a comparison between characters, and

measure the complexity of a given algorithm by the number of character comparisons. The rest of computing time is proportional to this measure.

2. Knuth-Morris-Pratt algorithm (KMP algorithm)

When we shift the pattern to the right after a mismatch is found at i on the pattern and j on the text, we did not utilise the matching history for the portion pat[1..i] and text[j-i+1..j]. If we can get information on how much to shift in advance, we can shift the pattern more, as shown in the following example.

Example 1.

```
        1  2  3  4  5  6  7  8  9 10 11 12 13 14
text    a  b  a  b  a  a  b  b  a  b  a  b  b  a
pattern a  b  a  b  b
           a  b  a  b  b
              a  b  a  b  b
                    a  b  a  b  b
```

 After mismatch at 5, there is no point in shifting the pattern one place. On the other hand we know "ab" repeats itself in the pattern. We also need the condition pat[3] <> pat[5] to ensure that we do not waste a match at position 5. Thus after shifting two places, we resume matching at position 5. Now we have a mismatch at position 6. This time shifting two places does not work, since "ab" repeats itself and we know that shifting two places will invite a mismatch. The condition pat[1]<>pat[4] is satisfied. Thus we shift pat three places and resume matching at position 6, and find a mismatch at position 8. For a similar reason to the previous case, we shift pat three places, and finally we find the pattern at position 9 of the text. We spent 15 comparisons between characters. If we followed Algorithm 1, we would spend 23 comparisons. Confirm this.
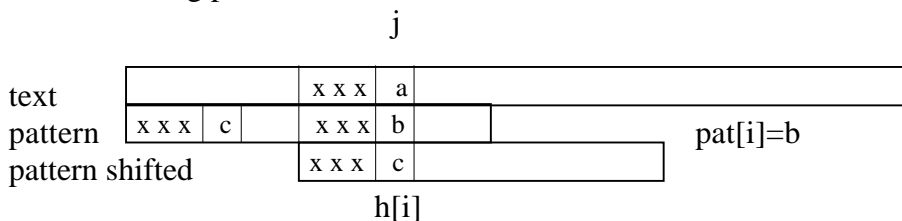
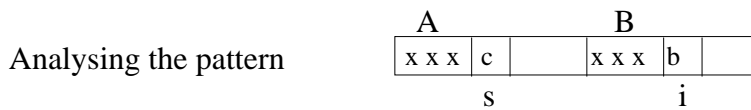The information of how much to shift is given by array h[1..m], which is defined by

$$h[1] = 0$$

$$h[i] = \max \{ s \mid (s=0) \text{ or } (pat[1 .. s-1] = pat[i-s+1 .. i-1] \text{ and } pat[s] <> pat[i]) \}$$

The situation is illustrated in the following figure.

Main matching process

Analysing the pattern

```
          A             B
      x x x  c       x x x  b
          s               i
```

The meaning of h[i] is to maximise the portion A and B in the above figure, and require b<>c. The value of h[i] is such maximum s. Then in the main matching process, we can resume matching after we shift the pattern after a mismatch at i on the pattern to position h[i] on the pattern, and we can keep going with the pointer j on the text. In other words, we need not to backtrack on the text. The maximisation of such s, (or minimisation of shift), is necessary in order not to overlook an occurrence of the pattern in the text. The main matching algorithm follows.
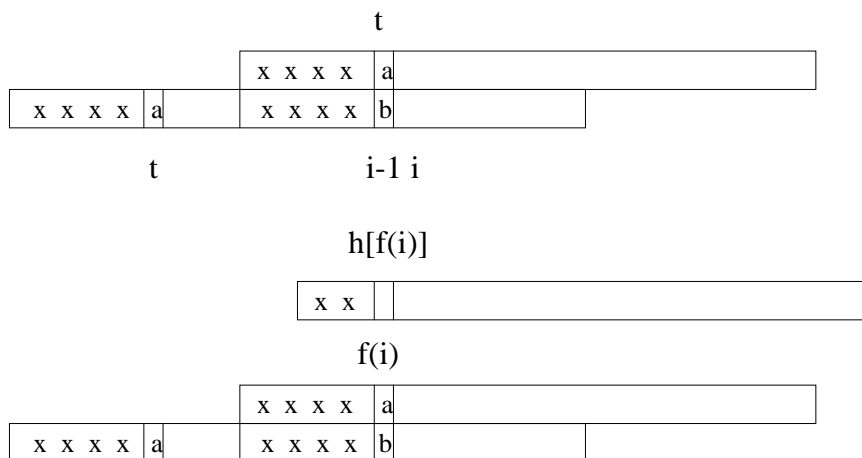
Algorithm 2. Matching algorithm

1. i:=1; j:=1;
2. **while** (i<=m) **and** (j<=n) **do begin**
3.   **while** (i>=0) **and** (pat[i]<>text[j]) **do** i:=h[i];
4.   i:=i+1; j:=j+1
5. **end**
6. **if** i <= m **then write**("not found")
7.   **else write**("found at", j-i+1)

Let the function f(i) be defined by

   $f(1) = 0$

   $f(i) = \max\{\ s\ |\ (1 <= s < i)\text{ and }(pat[1 .. s-1] = pat[i-s+1 .. i-1])\ \}$

The definitions of h[i] and f(i) are similar. In the latter we do not care about pat[s]<>pat[i]. The computation of h is like pattern matching of pat on itself.

```
                        t
                   x x x x  a
         x x x x  a      x x x x  b
              t            i-1 i

                    h[f(i)]
                   x x
                    f(i)
                   x x x x  a
         x x x x  a      x x x x  b
```

50

Algorithm 3. Computation of h

1. t:=0; h[1]:=0;
2. **for** i:=2 **to** m **do begin**
3.   /* t = f(i-1) */
4.   **while** (t>0) **and** (pat[i-1]<>pat[t] **do** t:=h[t];
5.   t:=t+1;
6.   /* t=f(i) */
7.   **if** pat[i]<>pat[t] **then** h[i]:=t **else** h[i]:=h[t]
8. **end**

The computation of h proceeds from left to right. Suppose t=f(i-1). If pat[t]=pat[i-1], we can extend the matched portion by one at line 5. If pat[t]<>pat[i-1], by repeating t:=h[t], we effectively slide the pattern over itself until we find pat[t]=pat[i-1], and we can extend the matched portion. If pat[i]<>pat[t] at line 7, the position t satisfies the condition for h, and so h[i] can be set to t. If pat[i]=pat[t], by going one step by h[t], the position will satisfy the condition for h, and thus h[i] can be set to h[t].

Example. pat = a b a b b

i = 2, at line 7 t=1, pat[1]<>pat[2], f(2) = 1, h[2]:=1

```
            i
        a b a b b
          a b a b b
            t
```
i = 3, t=1 at line 4. pat[1]<>pat[2], t:=h[1]=0, at line 7 t=1, f(3)=1, pat[1]=pat[3], h[3]:=0

```
        a b a b b
            a b a b b
```
i = 4, t=1 at line 4. pat[3]=pat[1], t:=t+1, t=2. At line 7, pat[4]=pat[2], h[4]:=h[2]=1

i = 5, t=2, f(4)=2. At line 4, pat[4]=pat[2], t:=t+1=3. f(5)=3. At line 7, pat[5]<>pat[3], h[5]:=t=3

Finally we have

```
   i | 1  2  3  4  5
-----------------------------
 pat | a  b  a  b  b
   f | 0  1  1  2  3
   h | 0  1  0  1  3
```

The time of Algorithm 2 is clearly O(n), since each character in the text is examined at most twice, which gives the upper bound on the number of comparisons. Also, whenever

a mismatch is found, the pattern is shifted to the right. The shifting can not take place more than n-m_1 times. The analysis of Algorithm 3 is a little more tricky. Trace the changes on the value of t in the algorithm. We have a doubly nested loop, one by the outer for and the other by while. The value of t can be increased by one at line 5, and m-1 times in total, which we regard as income. If we get into the while loop, the value of t is decreased, which we regard as expenditure. Since the total income is m-1, and t can not go to negative, the total number of executions of t:=h[t] can not exceed m-1. Thus the total time is O(m).

  Summarising these analyses, the total time for the KMP algorithm, which includes the pre-processing of the pattern, is O(m+n), which is linear.

**A Brief Sketch of the Boyer-Moore Algorithm**

In the KMP algorithm, we started matching process from the left end of the pattern. In the Boyer-Moore algorithm, we start matching from the right end of algorithm. By doing pre-processing on the pattern, we know how much to shift the pattern over the text when a mismatch is detected. See the following figure.

Text

x x x x x

Pattern

x x x x x

x x
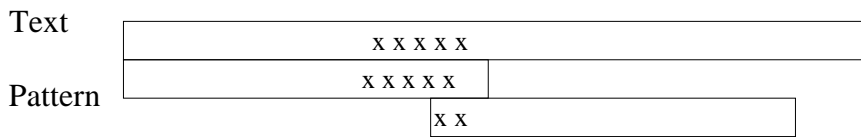
If the matched portion or a part of it exists within the pattern, we make an alignment as shown in the figure, and start the matching process from the right end of the pattern. For a random text and pattern, the probability that the matched portion appears in the pattern is very small, which means that the pattern can move the distance of m, which in turn gives us the time of O(n/m). As the parameter m is in the denominator, this complexity is called sub-linear, much faster than the KMP algorithm for a random text and pattern.

A Brief Sketch of Bird's Algorithm for Two-Dimensional Matching

The Aho-Corasick (AC) algorithm find an occurrence of a pattern of several patterns in O(n) time where n is the length of the text, with pre-processing of the patterns in linear time, that is, in $O(m_1 + ... + m_l)$ time where there are l patterns. This is called the multiple pattern matching and is a generalization of the KMP method. In Bird's algorithm, we use the AC algorithm horizontally and the KMP algorithm vertically. See the following figure.

j

row 1

row k-1
row R

i
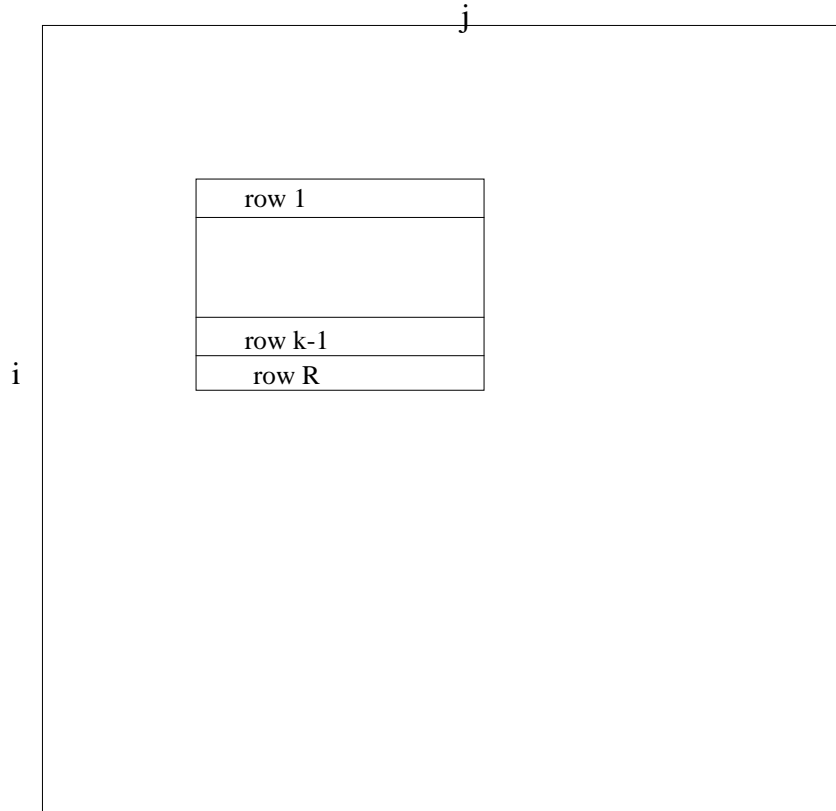
We call the m rows of the pattern row 1, ..., row m. Some of the rows may be equal. Thus we can compute the h function of row 1, ..., row m for the vertical pattern matching. We prepare a one-dimensional array a[1..n] to keep track of the partial matching. If a[j]=k, it means we have found row 1, ..., row k-1 of the pattern in the region of the text from the point (i,j) towards up and left. If the newly found row R is not found to be any row of the pattern, a[j] is set to 1. If row R belongs to the pattern, we use the h function of the vertical pattern. While R <> row k, we perform k:=h[k]. After this, if R matches row k, a[j] is set to k+1. If a[j] becomes m+1, the pattern is found. By a similar reasoning to the one-dimensional case, we can show that the time is $O(m^2 + n^2)$, which is linear. This is much better than the exhaustive method with $O(m^2n^2)$, as the data size is enormous in two-dimensions. There is an algorithm with sub-linear time, which is better than Bird's.

Example. row 1 = row 4, row 3 = row 5

Pattern	1 2 3 4 5	position	Vertical pattern
1	a a b b a	1	1
2	a a a b b	2	2
3	a b a b a	3	3
4	a a b b a	4	1
5	a b a b a	5	3

h function

```
              1  2  3  4  5
           -------------------------
     pat     1  2  3  1  3
     h       0  1  1  0  2
```

Two snapshots in the matching process in a portion of text.

```
                    j
           a a a b a b a
           b a a b b a b              a[j]=2
i          a a b a b a b     changed to a[j]=1


                    j
           a a a b a b a
           b a a b b a b              a[j]=2
i          a a a a b b b     changed to a[j]=3
```

Lecture Notes on Computational Geometry

1. Data structures for geometry
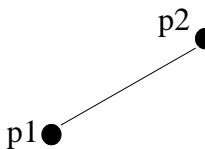
We deal with two-dimensional geometry. We express points by the struct type in C as follows:

```
struct point {
   int x, y;                        (x, y)  ●
}
```

We express line segments by two points as follows:

```
struct segment {                              p2 ●
   struct point p1, p2;
}                              p1 ●
```

2. Intersection of two line segments

In this section, we develop an algorithm for determining whether two line segments intersect. Let two line segments be q1 and q2. That is, the two end points of q1 are q1.p1 and q1.p2, and those of q2 are q2.p1 and q2.p2. We use the following condition for intersection. Note that a line segment defines an infinite (straight) line
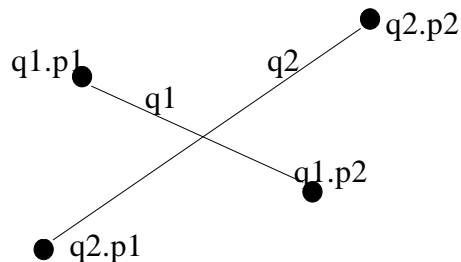
Condition: The two end points q1.p1 and q1.p2 must be on the opposite sides of the line defined by q2, and the two end points q2.p1 and q2.p2 must be on the opposite sides of the line defined by q1.

As an auxiliary function, we make the function "turn" that tests whether the angle made by three points p0, p1, and p2. The function test whether the vector p0 -> p2 is ahead of the vector p0 -> p1 anti-clock wise. If it is true, it returns 1, and -1 otherwise. See the following figure. The function value is 1 in this case.
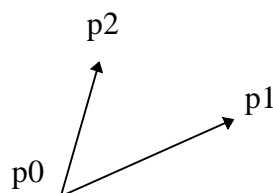
The function "intersect" together with "turn" is given in the following.
struct point { int x, y; }

struct segment { struct point p1, p2;}

int turn {struct point p0, struct point p1, struct point p2)
{ /* The last three cases are when three points are on a straight line */
   int dx1, dx2, dy1, dy2;
   dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
   dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
   if (dx1*dy2 > dy1*dx2) return 1;
   if (dx1*dy2 < dy1*dx2) return -1;
   if ((dx1*dx2 < 0)||(dy1*dy2 < 0) return -1;
   if ((dx1*dx1 + dy1*dy1) < (dx2*dx2 + dy2*dy2)) return 1;
   return 0;
}
int intersect(struct segment s1, struct segment s2)
{   if(((s1.p1.x==s1.p2.x)&&(s1.p1.y=s1.p2.y))||
      ((s2.p1.x==s2.p2.x)&&(s2.p2.x==s2.p2.x)))
    /* case 1: One of two segments shrinks to one point */
     return ((turn(s1.p1, s1.p2, s2.p1)*turn(s1.p1,s1.p2,s2.p2))<=0)
         || ((turn(s2.p1, s2.p2, s1.p1)*turn(s2.p1,s2.p2,s1.p2))<=0);
    else /* case 2: Neither of segments is a single point */
     return ((turn(s1.p1, s1.p2, s2.p1)*(turn(s1.p1, s1.p2, s2.p2)) <= 0)
       &&((turn(s2.p1, s2.p2, s1.p1)*trun(s2.p1, s2.p2, s1.p2))) <= 0);
}
main( ) /* for test */
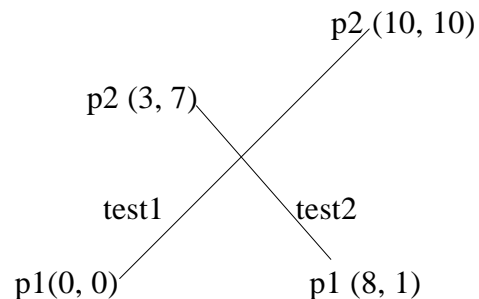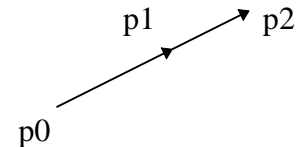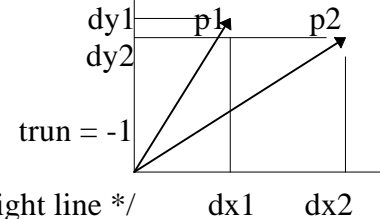{
   struct segment test1, test2;
   test1.p1.x = 0;   test1.p1.y = 0;
   test1.p2.x = 10; test1.p2.y = 10;
   test2.p1.x = 8;   test2.p1.y = 1;
   test2.p2.x = 3;   test2.p2.y = 7;
   printf(" %d\n", intersect(test1, test2));
}

This program correctly computes whether two segments intersect even in special cases like one point is on the other segment or one end point of one segment and one point of the other segment fall on the same point .

3. Interior point

We test if a point is inside a polygon. The polygon in not necessarily convex. We draw a semi-infinite line from the given point. If it intersects with the edges of the polygon an odd number of times, it is inside, and outside if even. When we traverse vertices of the

polygon, we check if they are on the semi-infinite line. If so we skip them and a both-infinite line is used for intersection test. See the following figure.
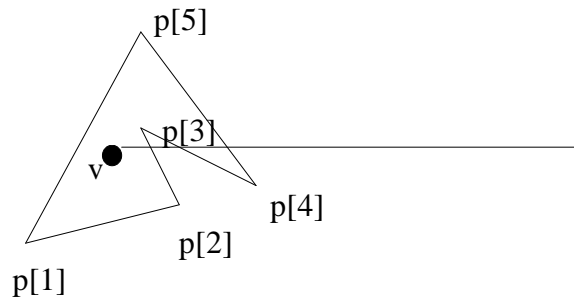
This program is easy to design if we use the function "intersect". We give the polygon by array p[1], ..., p[n] of points. The program follows.

```
int inside(struct point p[ ], int n, struct point v)
{
    /*  y-coordinate of p[1] is the smallest.  */
    int i,  j = 0, count = 0;
    struct segment sw, sp;
    p[0] = p[n];
    sw.p1 = v;  sw.p2.x = 20000;  sw.p2.y = v.y;     /* sw is the semi-infinite segment */
    ss.p1.y=v.y; ss.p2.y = v.y; ss.p1.x = -20000; ss.p2.x = 20000; /* ss is both-infinite */
    for (i=1; i<=n; i++) {
        sp.p1 = p[i];  sp.p2 = p[i];
        if (!intersect(sp, sw)) {  /* if p[i] is on sw we skip */
            sp.p2 = p[j];
            if (i==j+1)
            if (intersect(sp, sw) count++;   /* sp is segment from p[i] to p[i-1] */
            else if  (intersect(sp, ss)) count++;
            j = i;               /*  j goes one step behind i  */
        }
    }
    return count & 1  /* return 1 if count is odd, 0 otherwise */
}
main( ) /*  for test  */
{
    struct point v;
    struct point p[10];
    int n = 5;
    p[1].x = 0;  p[1].y = 0;
```

```
p[2].x = 5;  p[2].y = 1;
p[3].x = 4;  p[3].y = 5;
p[4].x = 9;  p[4].y = 2;
p[5].x = 2;  p[5].y = 8;
v.x = 3;  v.y = 3;
printf(" %d\n", inside(p, n, v));
```

4. Finding the Convex Hull

A convex polygon is a polygon such that the inner angle at each vertex is less than 180 degrees. The convex hull of n points is the smallest convex polygon that includes those points. We first describe the package wrapping method with $O(n^2)$ time and then Graham's algorithm with O(n log n) time.

Package wrapping: We start from a specified point and select the point we first encounter by going anti-clockwise, and repeat this process by going to the selected point. In the following figure we start from point 1 and select point 2, ..., 6. Finally we select 1 and finish.
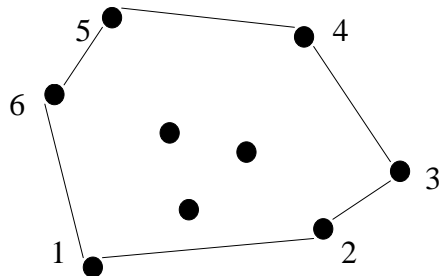
We need to compare the angles between segments. The C library function arctan will give us the angle in radian. If we just determine which of two angles is greater, we can use the following function "angle", which is more efficient.

```
double angle(struct point q1, struct point q2)
{
   /*  gives angle between 0 and 360 degrees */
   int dx, dy;
   double a = 0.0;
   dx = q2.x - q1.x;  dy = q2.y - q1.y;
   if (dx != 0 || dy != 0) a = (double)dy / (double)(abs(dx) + abs(dy));
   if (dx < 0) a = 2.0 - a;
   else if (dy < 0) a = a + 4.0;
   return a*90.0;
}
int convex_hull(struct point p[ ], int n)
```
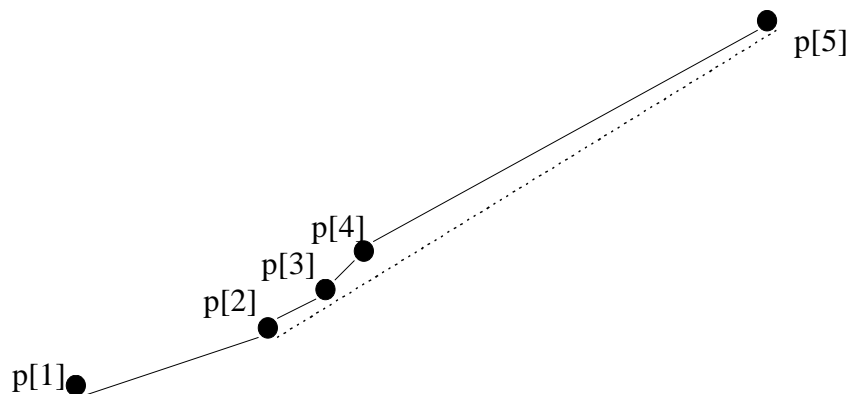
```
{
   /* p[1] through p[n] are vertices of a polygon */
   int i, m, min; double theta, prev;
   struct point t;
   min = 1;
   for (i=2; i<=n; i++) if (p[i].y<p[min].y) min = i;
   t = p[1];  p[1] = p[min]; p[min] = t;
   p[n+1] = p[1];
   min = n; theta = 360.0;
   for (i=2; i<=n; i++)
      if (angle(p[1], p[i]) < theta) {min = i; theta = angle(p[1], p[min]);
      }
   for (m=2; m<=n; m++) {
      t = p[m];  p[m] = p[min]; p[min] = t;  /* swap p[m] and p[min] */
      prev = theta;
      min = n + 1;  theta = 360.0;
  /*** This part finds the point p[min] such that angle(p[m], p[min]) is minimum
      and greater than the previous angle. ***/
      for (i=m+1; i<=n+1; i++)
         if ((angle(p[m], p[i]) < theta )&& (angle(p[m], p[i]) >= prev))
            {min = i; theta = angle(p[m], p[min]);}
      if (min == n+1) return m;  /* m is the number of points on the hull */
   }
}
```

As the program has a double nested loop structure, the computing time is $O(n^2)$, which is not very efficient. In the following, we introduce Graham's algorithm invented by R. L. Graham in 1972, which runs in $O(n \log n)$ time.

Graham's algorithm: We start from the same point as before, that is, the lowest point p[1]. We sort the angles angle(p[1], p[i]) for i=1, ..., n in increasing order by any fast sorting algorithm, such as quicksort. Let p[1], ..., p[n] be sorted in this way. Then we pick up p[2], p[3], ...in this order. We keep selecting points as long as we go anti-clockwise from the previous angle. If we hit a clockwise turn, we throw away already selected points until we recover clockwise angle. See the following figure.

After we select up to p[4], we throw away p[3] and p[4], and connect p[2] and p[5]. To determine clockwise turn or anti-clockwise turn, we use the function 'turn', introduced before. The main part of the algorithm follows.

```
struct point { int x, y; double angle;};
struct point a[100000];
int convex_hull(struct point p[], int n)
{
   /* p[1] through p[n] are vertices of a polygon */
   int i, m, min; double theta, prev;
   struct point t;
   min = 1;
   for (i=2; i<=n; i++) if (p[i].y < p[min].y) min = i;
   for (i=1; i<=n; i++) if (p[i].y==p[min].y)
                   if (p[i].x > p[min].x) min = i;
   t = p[1];  p[1] = p[min]; p[min] = t;
   for (i=1; i<=n; i++) p[i].angle=angle(p[1],p[i]);
   for (i=1; i<=n; i++) a[i]=p[i];
   quick(1, n); /* This sorts a[] using keys a[i].angle for i=1, ..., n */
   for (i=1; i<=n; i++) p[i]=a[i];
   p[0] = p[n];
   m = 3;
   for (i=4; i<=n; i++) {
      while (turn(p[m], p[m-1], p[i]) >= 0) m--; /* throw away points */
      m++;
      t = p[m];  p[m] = p[i]; p[i] = t;
   }
   return m;  /* m is the number of points on the hull */
} /* p[1], ..., p[m] are points on the hull */
```

The main complexity is that of sorting. The time for the rest is O(n) as we examine each point once. Hence the total time is O(n log n).

5. Voronoi Diagram

Suppose we have n points on a plane such as a square. The Voronoi diagram separates those points by line segments or semi-infinte lines in such a way that in any closed region there is only one point and any point in the region is closer to the point than to any other point. If we imagine a big circle surrounding those points, we can define a region with the semi-infinite lines and the circle. The regions define territories for those points, so to speak.

Example.
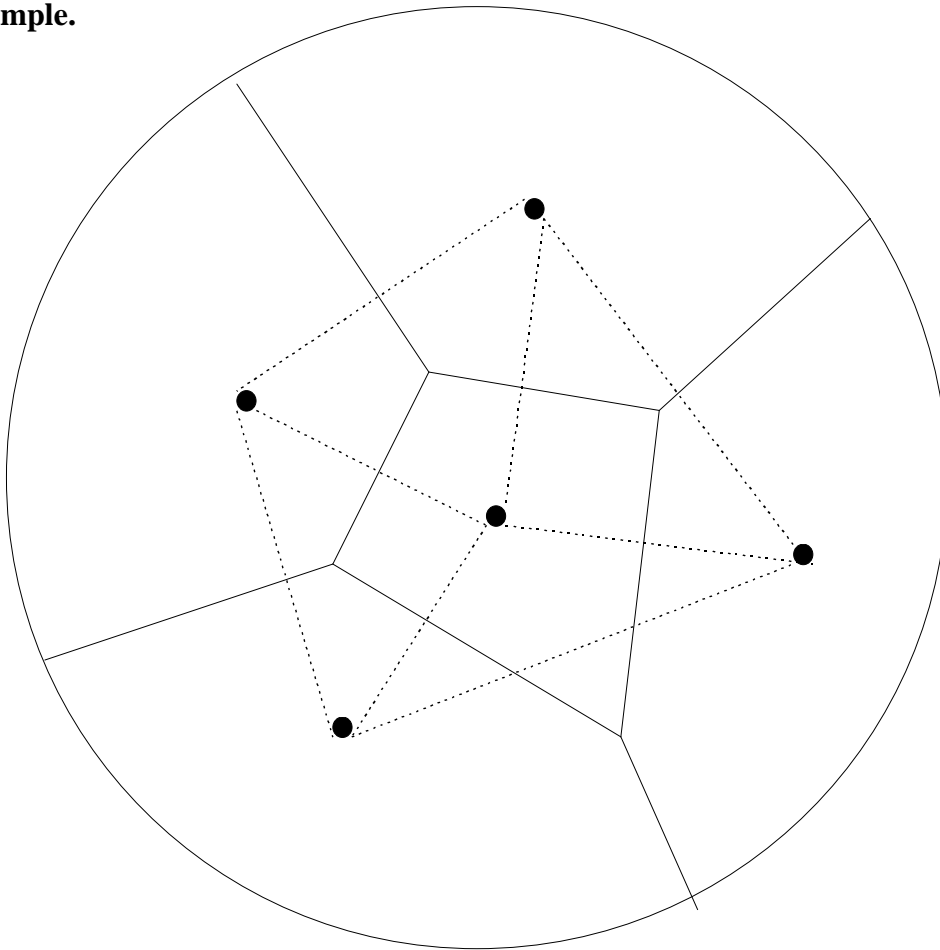
Note that the three perpendiculars of a triangle meet at one point. In this example five points are surrounded by line segments and the outer circle, and the five regions are defined. The line segment between two points, say, a and b, is the perpendicular of the points a and b. Any pair of two points which are in adjacent regions are connected by a dotted line. Those dotted lines define triangles, and the resulting diagram is called the Delauney diagram.

There are a number of algorithms for Voronoi diagrams. The following is based on the divide-and-conquer paradigm.

Algorithm. Voronoi Diagram

1. Sort the n points according to the x-co-ordinates and call the p[1], ..., p[n].
2. Call Voronoi(1, n, P, V).
3. Define Voronoi(*l*, r, P, V) as follows:
4. If there are only three or less in P, draw the diagram directly.
5. Let m=(l+r).
6. Let P1 = {p[l], ..., p[m]}.
7. Call Voronoi(l, m, P1, V1).
8. Let P2 = {p[m+1], ..., p[r]}..
9. Call Voronoi(m, r, P2, V2).
10. Obtain the zipping line Z for V1 and V2 by algorithm M
11. Delete portions of line segments in V1 exceeding Z to the right.
12. Delete portions of line segments in V2 exceeding Z to the left.
13. Return the resulting diagram as V

Algorithm M

1. Obtain the convex hull for the combined set of P1 and P2
2. Let L = (a, b) be the lower segment that connect a point in P1 and
   a point in P2 in the hull.
3. Originate Z as a point at the bottom.
4. Draw a perpendicular Z for L upwards until hitting a segment S in V1 or V2.
5. If there is no such intersection, halt.
6. If S is in V1, it is a perpendicular for and x for some x. Let a be x and go to 3.
7. If S is in V2, it is a perpendicular for b and y for some y. Let b be y and go to 3.

The time for Algorithm M is O(n). As usual, we can establish the following recurrence for the computing time.

$$T(3) = c_1$$
$$T(n) = 2T(n) + c_2 n$$

The solution is T(n) = O(n log n). Together with the sorting, the total time is O(n logn).

Example.

Legends

L   -------------

Z   ———————