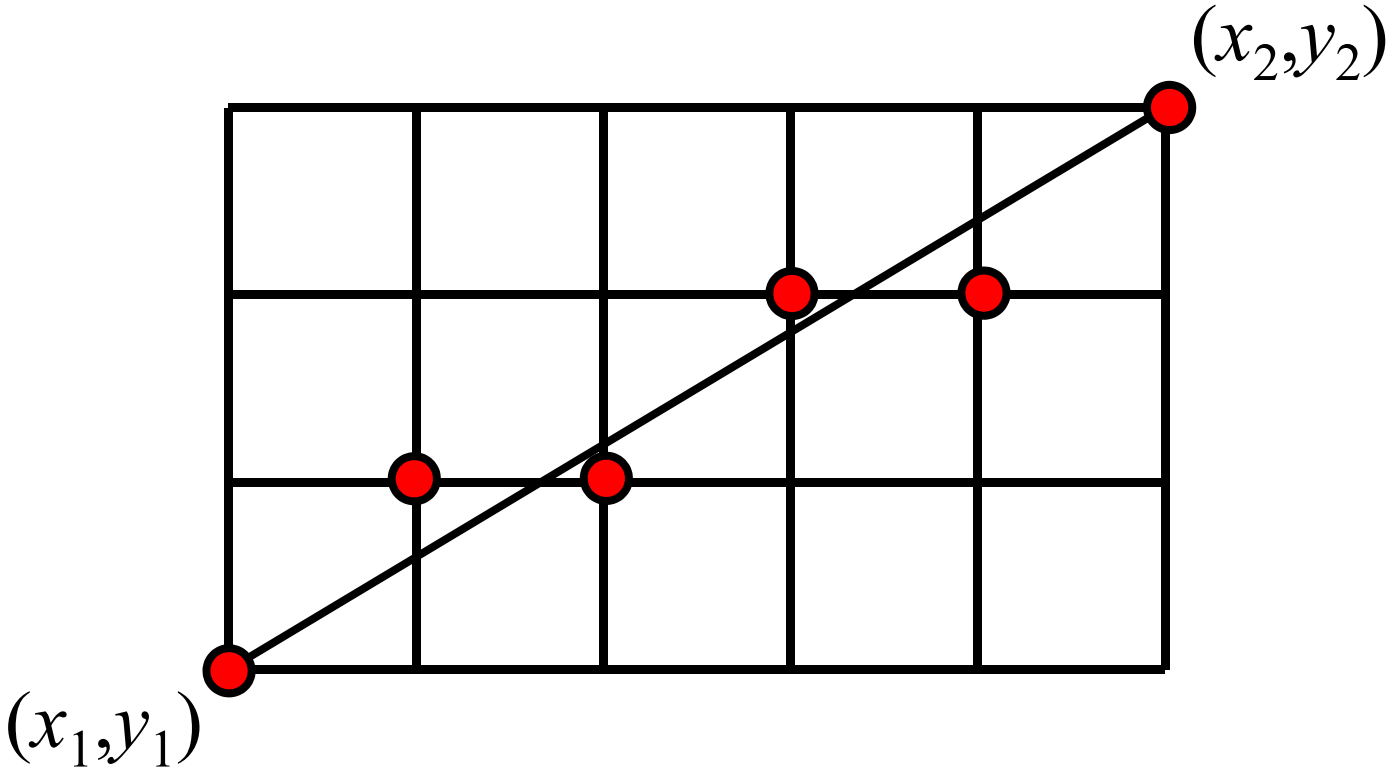
第二章 图形基元的显示

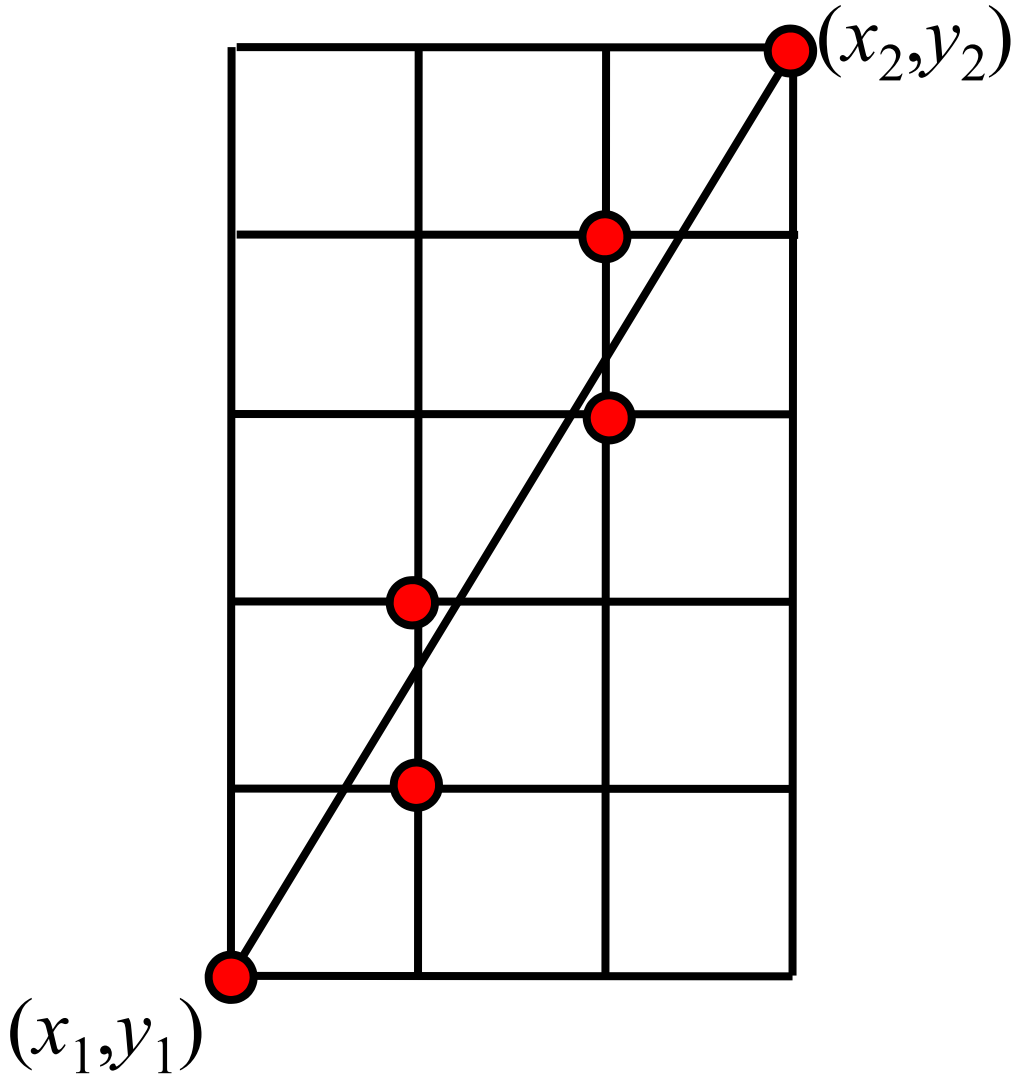
- 扫描转换 将图形描述转换成用象素矩阵表示的过程
- 图形基元（输出图形元素）图形系统能产生的最基本图形
- 线段、圆、椭圆、多边形

- 第一节 直线扫描转换算法
- 第二节 圆的扫描转换算法
- 第三节 椭圆扫描转换算法
- 第四节 区域填充

第一节 直线扫描转换算法

- DDA直线扫描转换算法
- 中点画线法
- Bresenham画线算法


(x_2,y_2)
(x_1,y_1)


(x_2,y_2)
(x_1,y_1)

1.DDA线段扫描转换算法

设待画线段两端点的坐标值(x_1,y_1)和(x_2,y_2)，假定

$x_1<x_2$

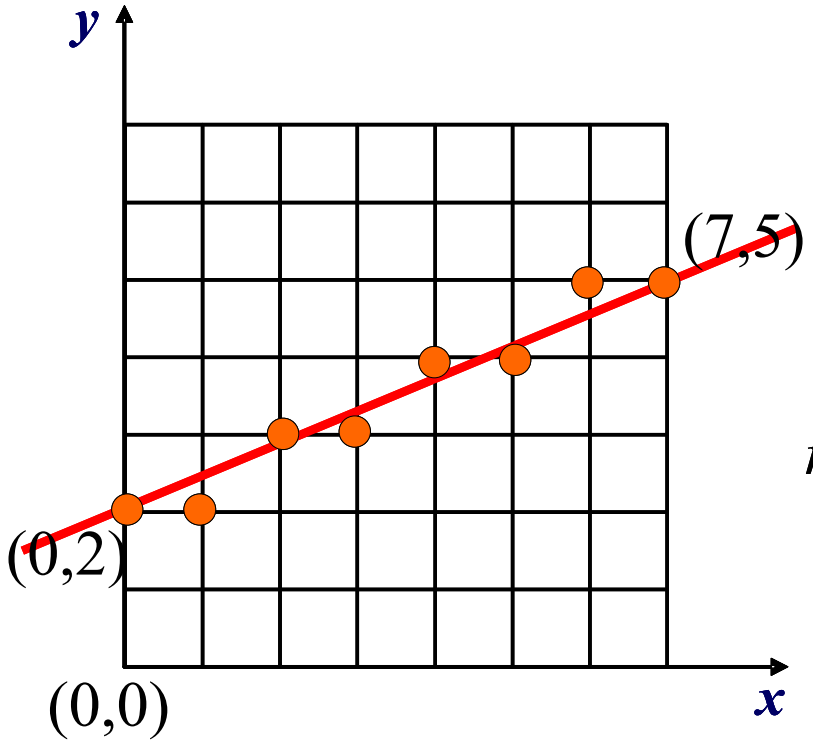
$y=mx+b$

$m=(y_2-y_1)/(x_2-x_1)$

$b=(x_2y_1-x_1y_2)/(x_2-x_1)$

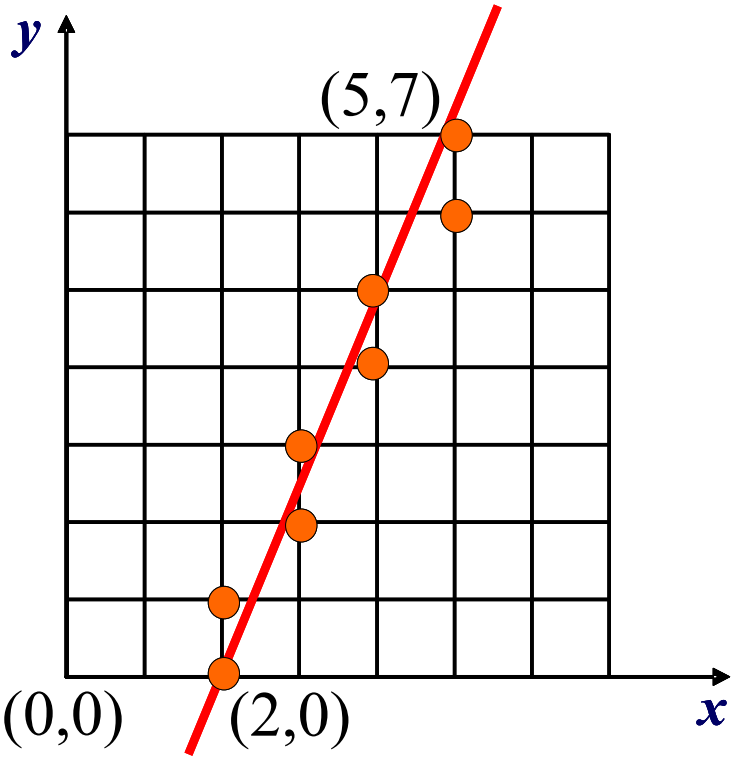
$|m|\leq 1$,对x每增1取允许的各整数值:

$$m=\frac{\Delta y}{\Delta x}=\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$$



$$y = mx + b$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$




```
void DDALine(int x1,int y1,int x2,int y2)
{
    double dx,dy,e,x,y;
    dx=x2-x1;
    dy=y2-y1;
    e=(fabs(dx)>fabs(dy))?fabs(dx):fabs(dy);
    dx/=e;
    dy/=e;
    x=x1;
    y=y1;
    for(int i=1;i<=e;i++)
    {
        SetPixel((int)(x+0.5), (int)(y+0.5));
        x+=dx;
        y+=dy;
    }
}
```

2.中点画线法

假定直线斜率在0、1之间，$x = x_i$时已选(x_i,y_i)像素，

确定$x = x_i + 1$与直线最近的像素

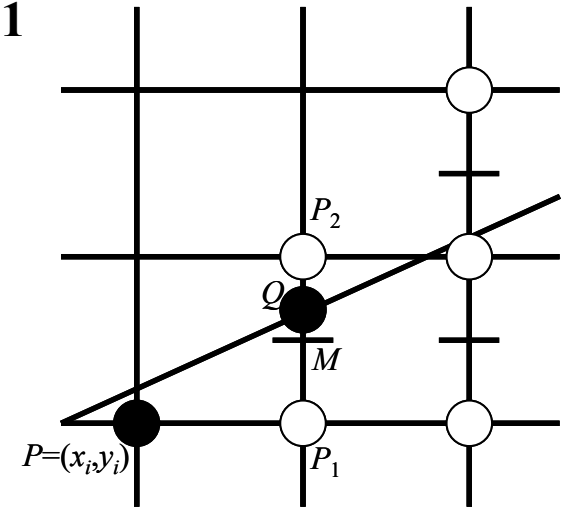
$P_1(x_i + 1, y_i)$、$P_2(x_i+1, y_i+1)$

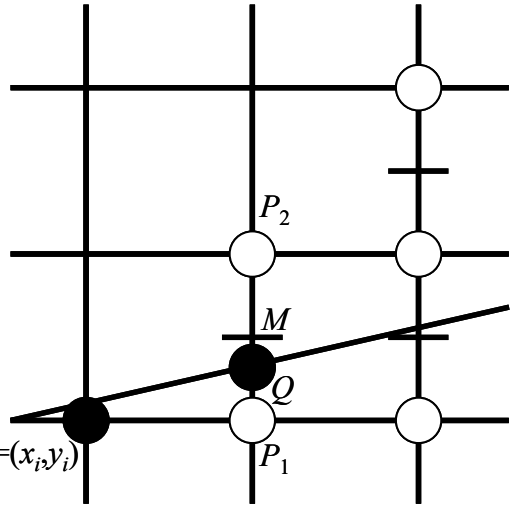
M表示P_1与P_2的中点，$M = (x_i+1, y_i + 0.5)$。

Q是直线与垂直线$x = x_i + 1$的交点

若M在Q的下方，则P_2离直线近，应取为下一个像素；

否则应取P_1





起点和终点分别为 (x_0 , y_0) 和(x_1 , y_1)。

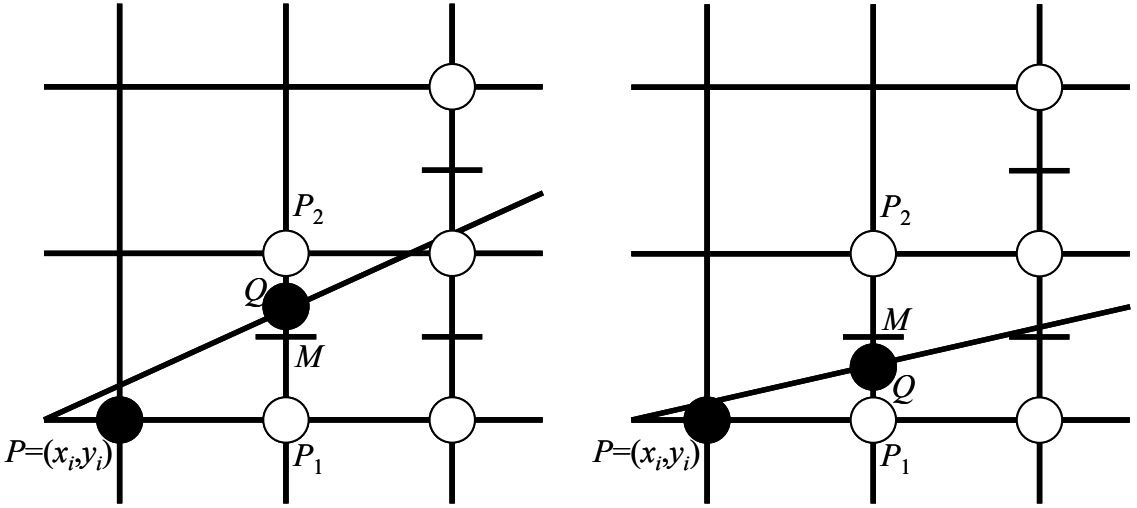
其所在直线F(x , y) = ax + by + c

其中，$a=y_0-y_1$，$b=x_1-x_0$，$c=x_0y_1-x_1y_0$。

直线上的点，F(x,y) = 0 ；

直线上方的点，F(x,y)>0 ；

直线下方的点，F(x,y)<0。



Q在M的上方还是下方，只要把M代入F(x，y)，并判断它的符号。

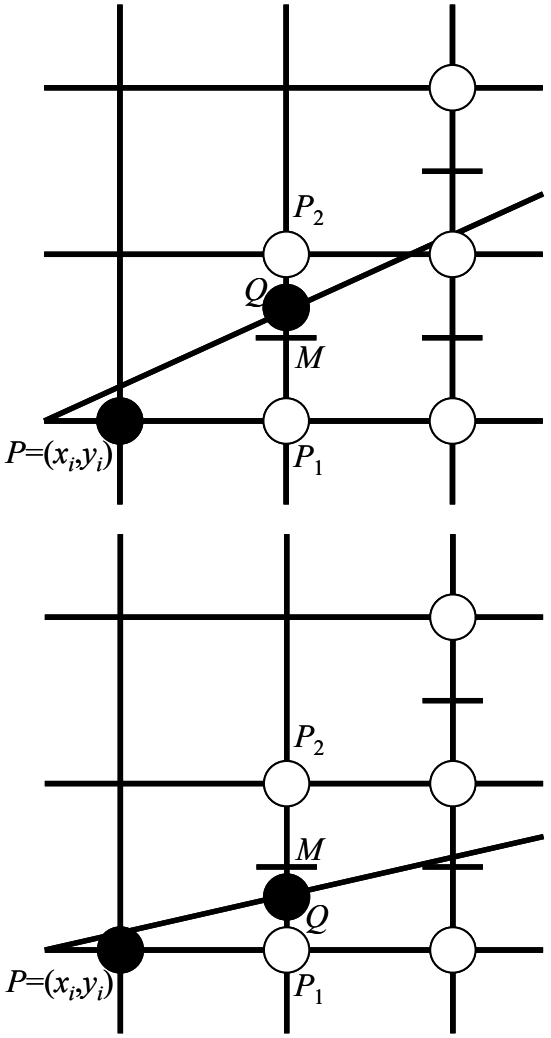
$$d = F(M) = F(x_i+1，y_i+0.5)$$
$$= a(x_i+1) + b(y_i+0.5) + c$$

当 $d < 0$ 时，M在直线下方(即在Q的下方)，应取右上方的P_2。

当 $d > 0$ 时，则取正右方的P_1。

当 $d = 0$ 时，二者一样合适，取P_1。

d是x_i, y_i的线性函数，因此可采用增量计算，提高运算效率。

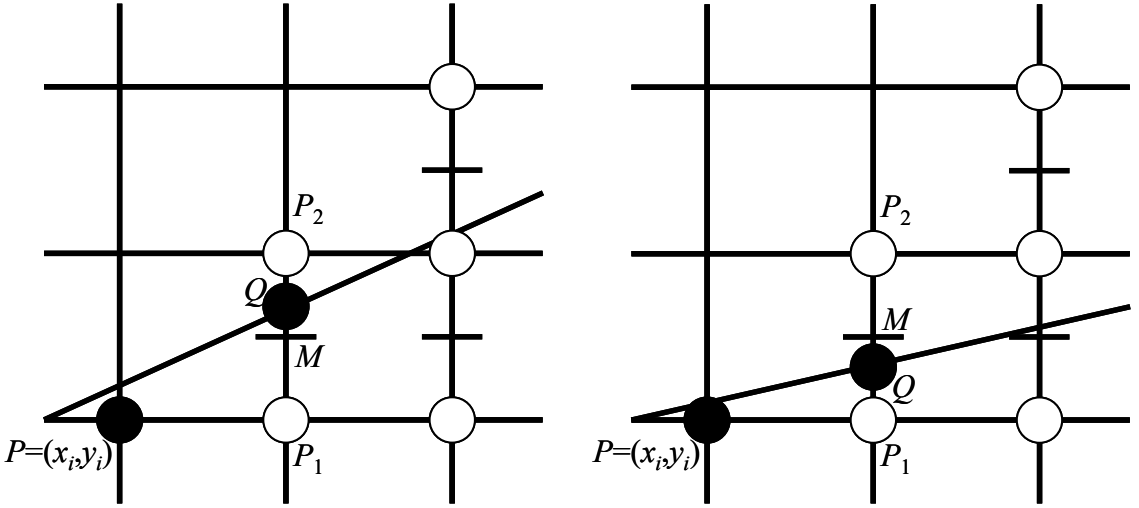


对每一个像素计算判别式d，根据它的符号确定下一像素。

若 $d \geq 0$ 时，取正右方像素P_1，判断再下一个像素应取哪个，应计算

$$d_1 = F(x_i+2, y_i+0.5)$$
$$= a(x_i+2) + b(y_i+0.5) + c = d + a$$

故d的增量为a。



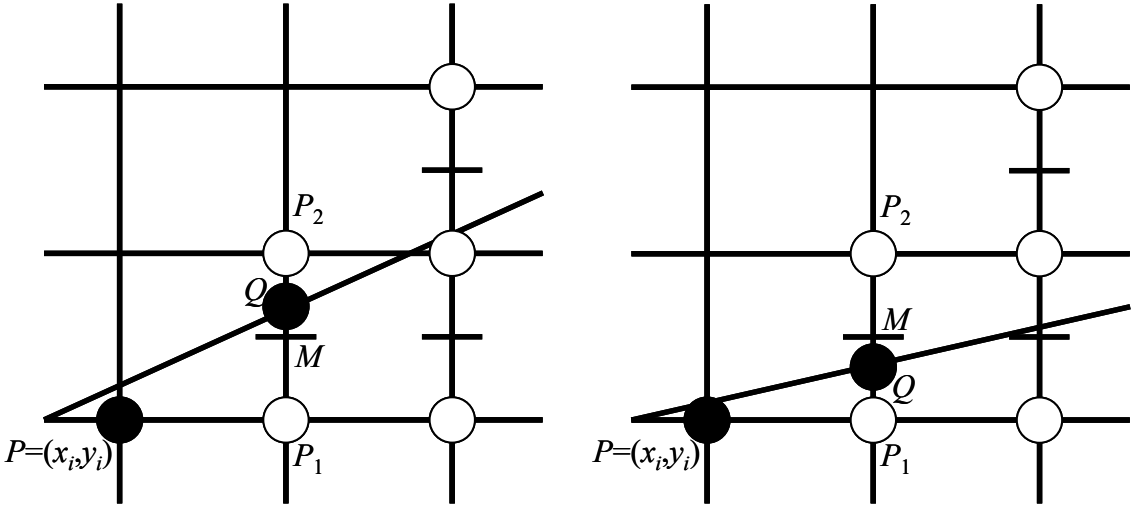
若 d<0 时，则取右上方像素P_2。要判断再下一个像素，则要计算

$d_2 = F(x_i+2, y_i+1.5)$

$= a(x_i+2) + b(y_i+1.5) + c$

$= d + a + b$

故在第二种情况，d的增量为a + b



再看d的初始值。显然，第一个像素应取左端点(x_0，y_0)，相应的判别式值为

$$
\begin{aligned}
d_0 &= F(x_0+1,\ y_0+0.5) \\
&= a(x_0+1)+b(y_0+0.5)+c \\
&= a x_0 + b y_0 + c + a + 0.5b \\
&= F(x_0,\ y_0) + a + 0.5b
\end{aligned}
$$

但由于(x_0，y_0)在直线上，故$F(x_0, y_0)=0$。因此，d的初始值为$d_0 = a+0.5b$

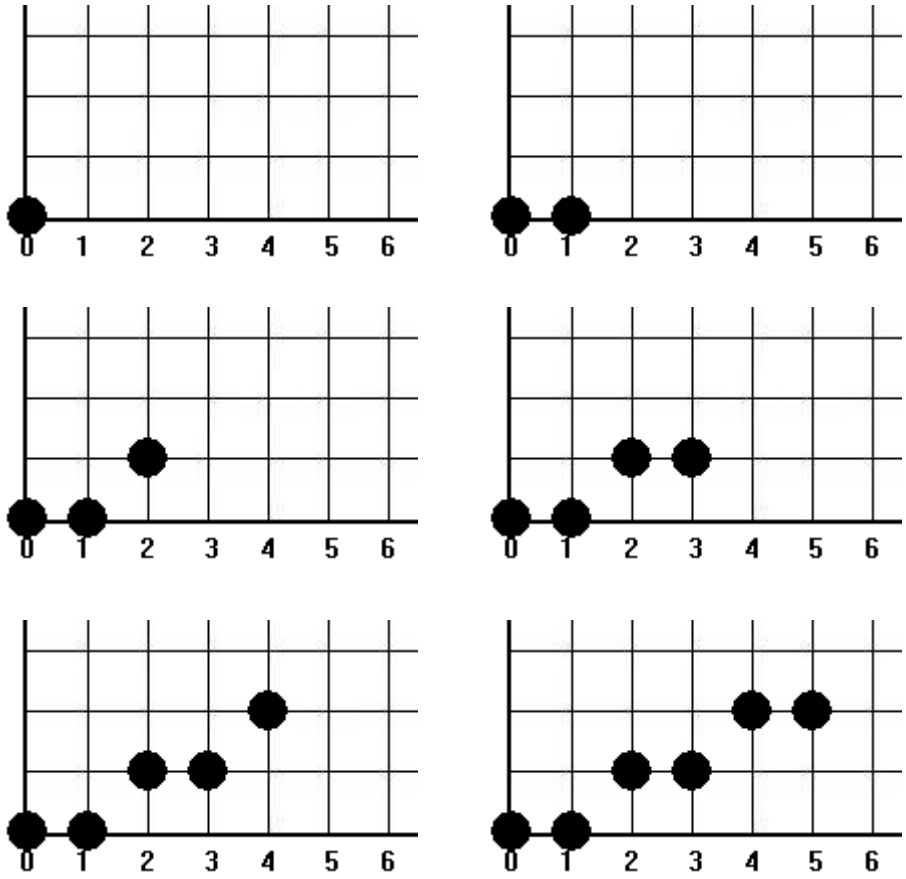
为避免小数运算，考虑用2d来代替d的计算


```
void MidpointLine (int x0,int y0,int x1,int y1){
    int a,b,delta1,delta2,d,x,y ;
    a = y0 - y1;     b = x1 - x0;     d = 2 * a + b ;
    delta1 = 2 * a ;          delta2 = 2 *( a + b);
    x = x0 ;          y = y0 ;
    SetPixel(x,y);
    while( x<x1  ) {
        if( d<0 ) {
            x ++;   y ++;
            d+= delta2;
        }
        else {
            x ++;
            d+= delta1;
        }
        SetPixel(x,y);
    } /* while */
}/* MidpointLine */
```

例:(0，0)、(5，2) $a = y_0 - y_1 = -2$,

$b = x_1 - x_0 = 5, d_0 = 2 * a + b = 1$,

delta1=2a=-4,delta2=2(a+b)= 6



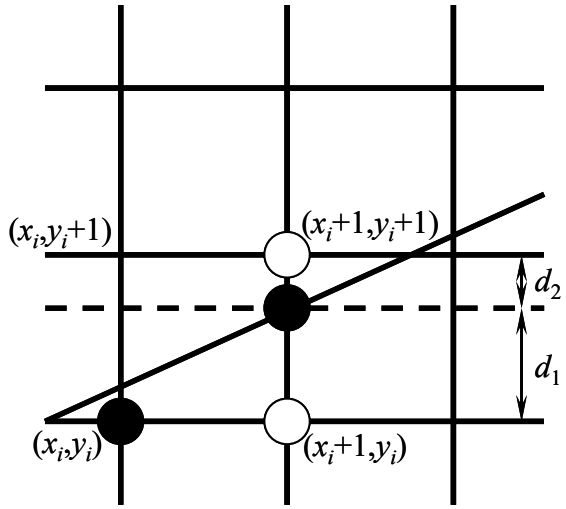
x	*y*	*d*	Δ*d*
0	0	1	-4
1	0	-3	6
2	1	3	-4
3	1	-1	6
4	2	5	-4
5	2	1	

```
if( d<0 ) {
        x ++;   y ++;  d+= delta2;
}
else {
        x ++;   d+= delta1;
}
```

3.Bresenham画线算法

斜率m在0到1之间，并且$x_2 > x_1$

设在第i步已经确定第i个像素点是(x_i, y_i)，现在看第i+1步如何确定第i+1个像素点的位置。



$$d_1 = y - y_i = m(x_i + 1) + b - y_i$$

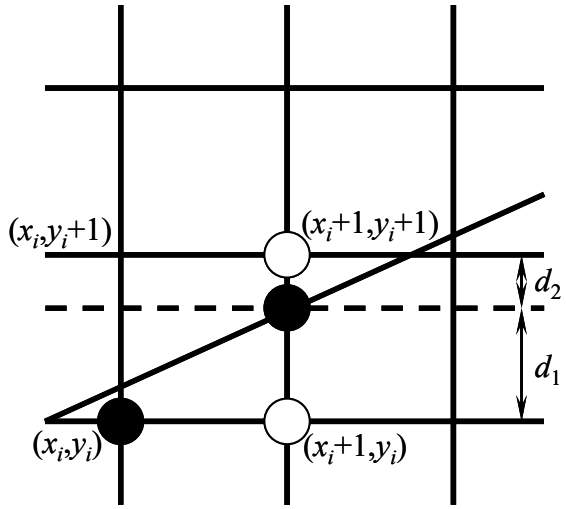
$$d_2 = (y_i + 1) - y = (y_i + 1) - m(x_i + 1) - b$$

$$d_1 - d_2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

$d_1 > d_2$,下一个像素点取$(x_i + 1, y_i + 1)$

$d_1 < d_2$,下一个像素点取$(x_i + 1, y_i)$

$d_1 = d_2$,取两像素点中的任意一个



$$d_1 - d_2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

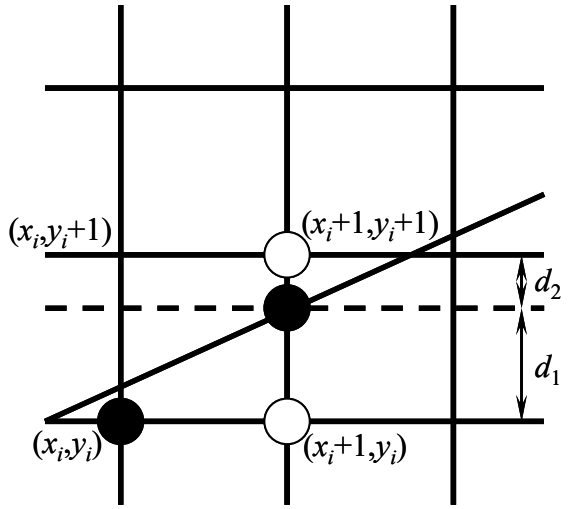
$$p_i = \Delta x(d_1 - d_2)$$

$$= 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + 2\Delta y + \Delta x(2b - 1)$$

$$= 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + c$$

$$p_{i+1} = 2\Delta y \cdot x_{i+1} - 2\Delta x \cdot y_{i+1} + c$$

$$p_{i+1} - p_i = 2\Delta y - 2\Delta x(y_{i+1} - y_i)$$



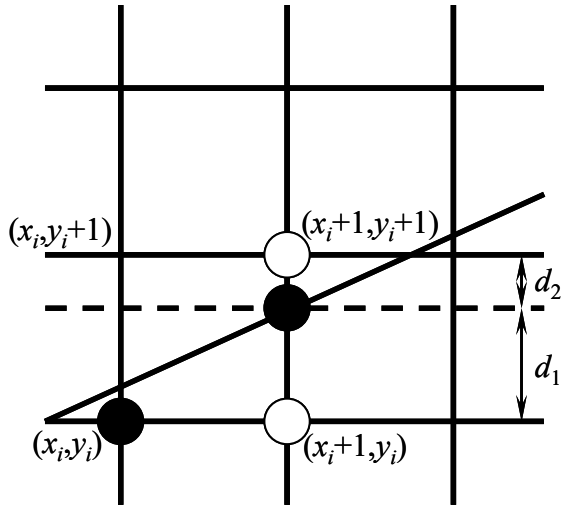
$$p_i = 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + c$$

$$p_{i+1} = 2\Delta y \cdot x_{i+1} - 2\Delta x \cdot y_{i+1} + c$$

$$p_{i+1} - p_i = 2\Delta y - 2\Delta x(y_{i+1} - y_i)$$

$p_i \geq 0$,应取$y_{i+1} = y_i + 1$, $p_{i+1} = p_i + 2(\Delta y - \Delta x)$

$p_i < 0$,应取$y_{i+1} = y_i$, $p_{i+1} = p_i + 2\Delta y$



确定P_1，

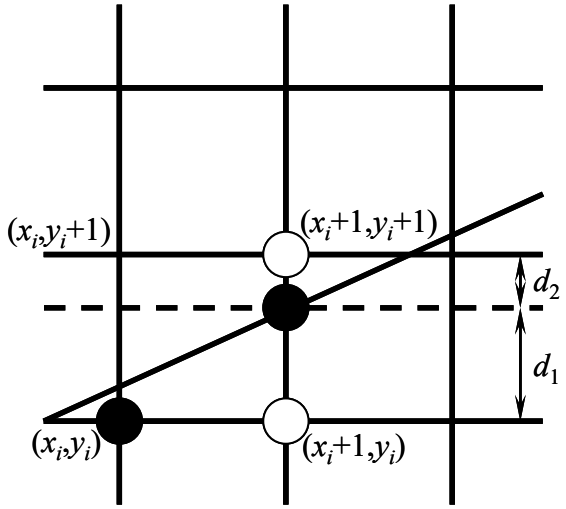
$$x_1 = x_1, y_1 = \frac{\Delta y}{\Delta x} x_1 + b \Rightarrow \Delta x \cdot y_1 = \Delta y \cdot x_1 + \Delta x \cdot b$$

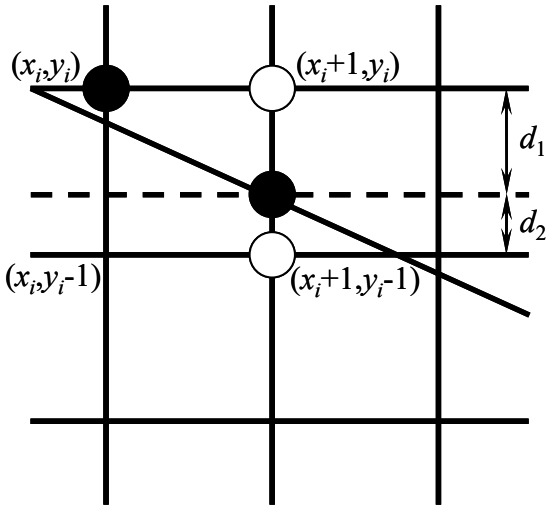
令i=1，可计算求出：

$$p_i = 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + 2\Delta y + \Delta x(2b-1)$$

$$\begin{aligned} p_1 &= 2\Delta y \cdot x_1 - 2\Delta x \cdot y_1 + 2\Delta y + \Delta x(2b-1) \\ &= 2\Delta y \cdot x_1 - 2\Delta y \cdot x_1 - 2\Delta x \cdot b + 2\Delta y + 2\Delta x \cdot b - \Delta x \\ &= 2\Delta y - \Delta x \end{aligned}$$

```
void BresenhamLine(int x1,int y1,int x2,int y2){
    int x,y,dx,dy,p;
    x=x1;
    y=y1;
    dx=x2-x1;
    dy=y2-y1;
    p=2*dy-dx;
    for(;x<=x2;x++) {
        SetPixel(x,y);
        if(p>=0) {
            y++;
            p+=2*(dy-dx);
        }
        else {
            p+=2*dy;
        }
    }
}
```



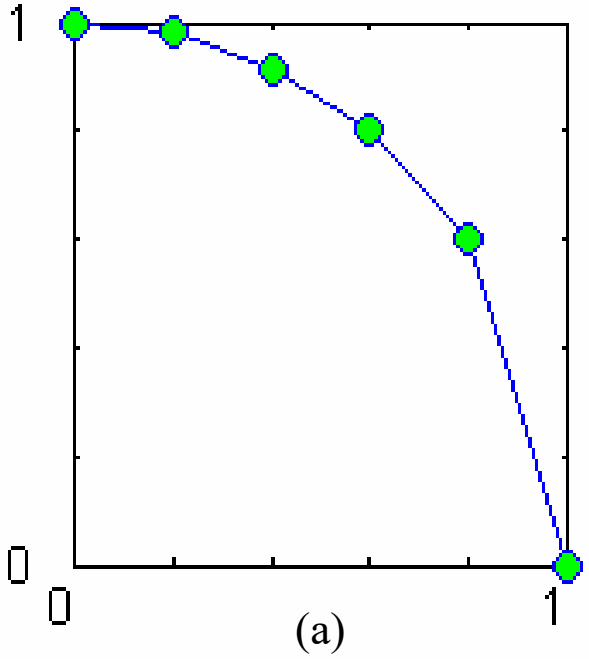


第二节 圆的扫描转换算法

$x^2+y^2=R^2$

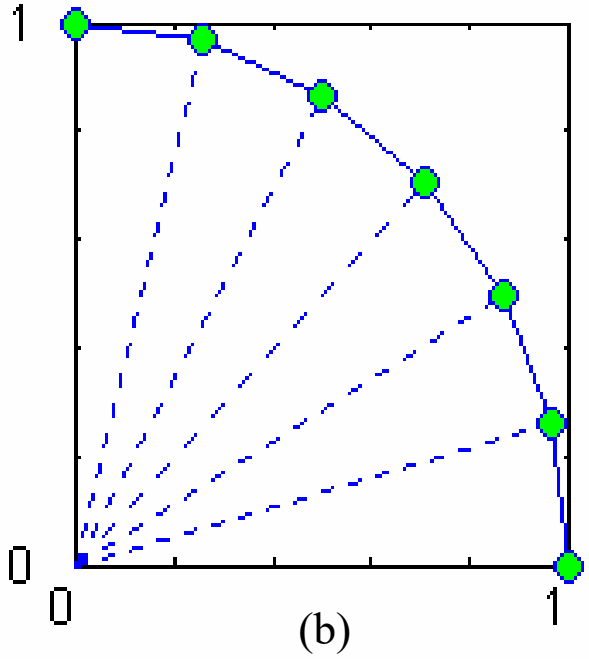
$x=R\cos(\theta)$

$y=R\sin(\theta)$



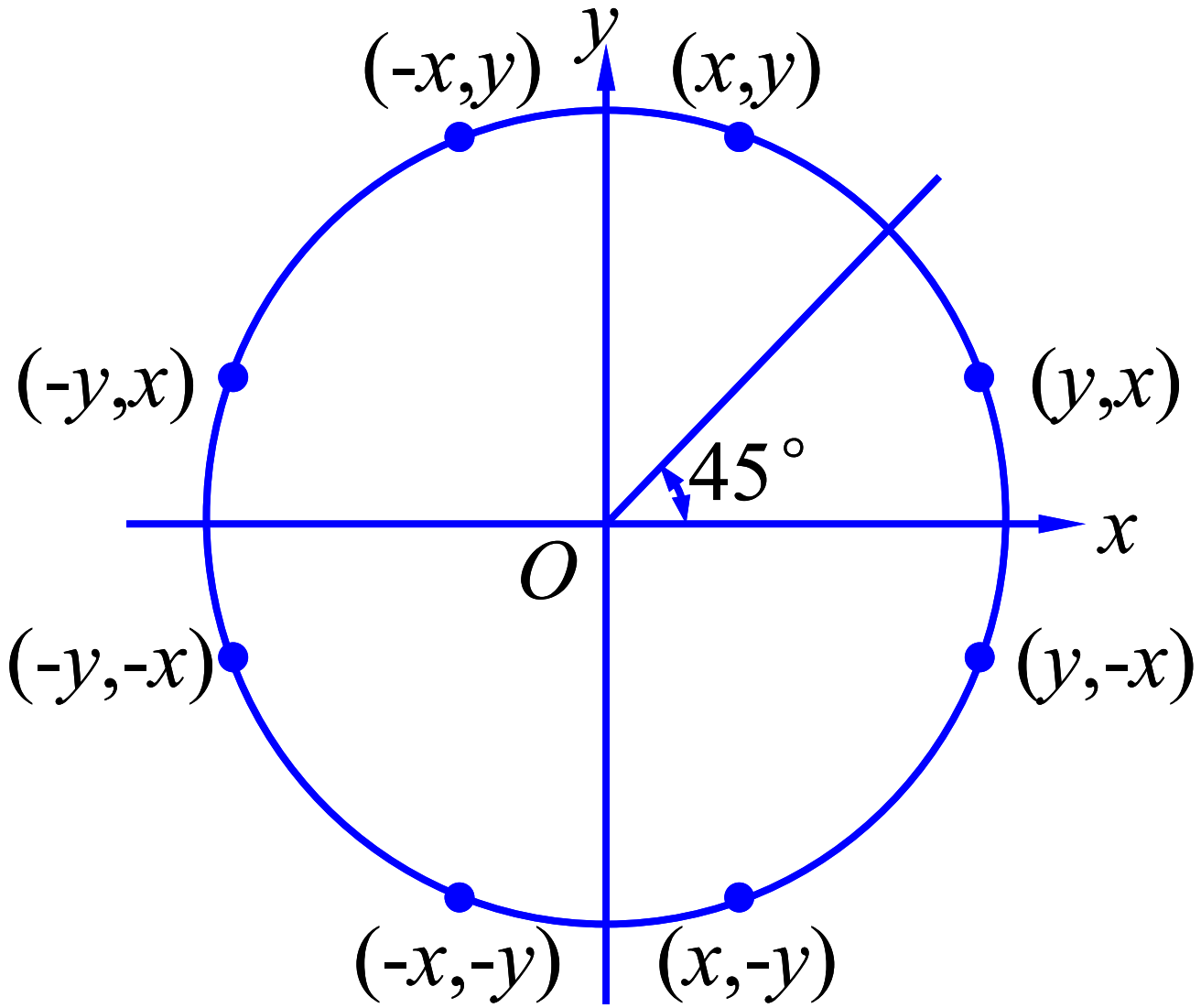
(a)

$y=\sqrt{1-x^2} \quad 0\le x\le 1$



(b)

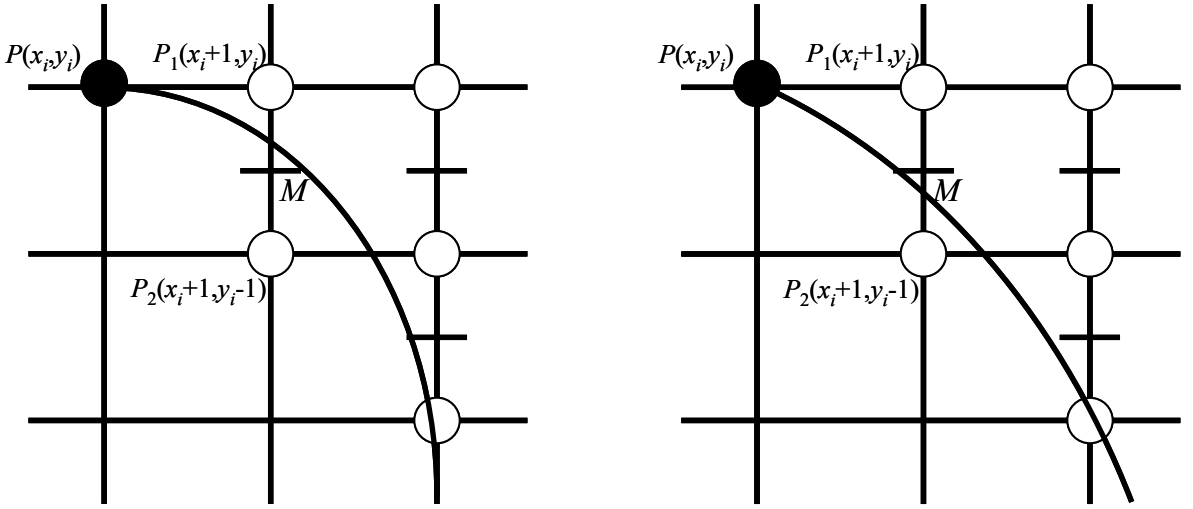
$x=\cos\theta \quad y=\sin\theta \quad 0\le\theta\le\pi/2$


(-x,y)
y
(x,y)
(-y,x)
(y,x)
45°
x
O
(-y,-x)
(y,-x)
(-x,-y)
(x,-y)

中点画圆法

讨论如何从点(0，R)至($R/\sqrt{2}$，$R/\sqrt{2}$)的1／8圆周

若x坐标为(x_i,y_i)，下一个像素只能是(x_i+1,y_i) 的P_1点或 (x_i+1,y_i-1) 的P_2点



构造函数：$F(x,y)=x^2+y^2-R^2$

圆上的点，$F(x,y)=0$

圆外的点，$F(x,y)>0$

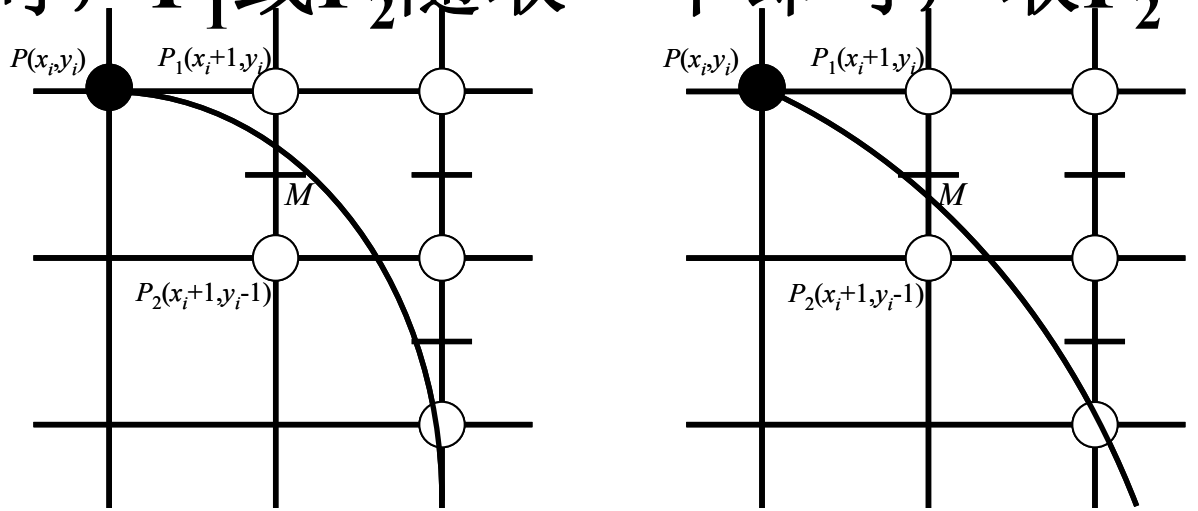
圆内的点，$F(x,y)<0$

P_1和P_2的中点为 $M=(x_i+1,y_i-0.5)$

$F(M)<0$时，M在圆内，取P_1

$F(M)>0$时，M在圆外，取P_2

$F(M)=0$时，P_1或P_2随取一个即可，取P_2



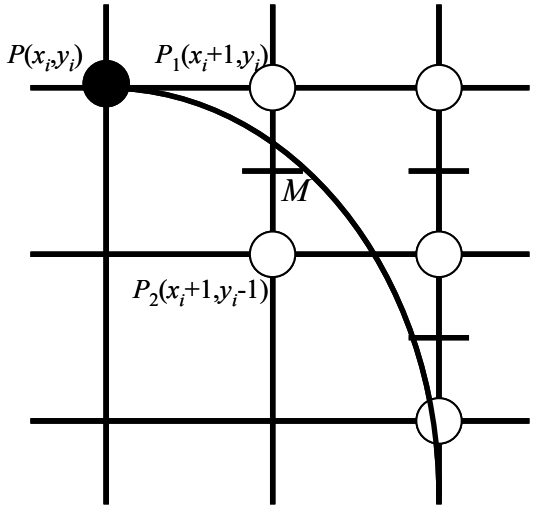
构造判别式 $d = F(M) = F(x_i + 1, y_i - 0.5)$

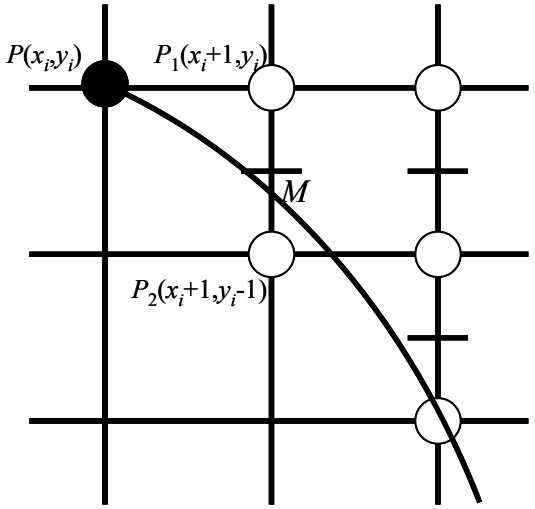
$$= (x_i + 1)^2 + (y_i - 0.5)^2 - R^2$$

若**d<0**，取P_1作为**下一个像素**，而且**再下一个像素**的判别式为

$$d' = F(x_i + 2, y_i - 0.5)$$

$$= (x_i + 2)^2 + (y_i - 0.5)^2 - R^2 = d + 2x_i + 3$$





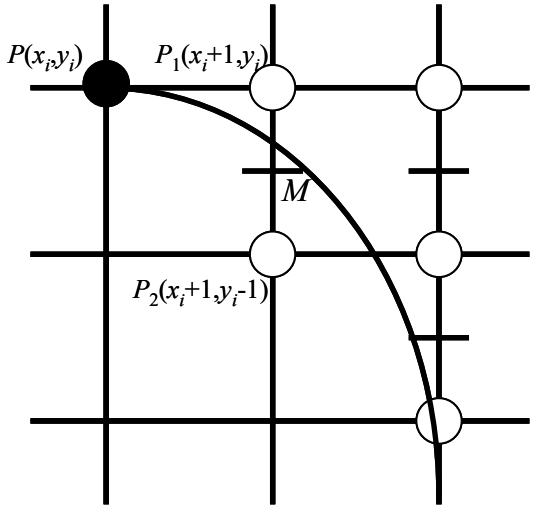
若**d≥0**，应取**P_2**作为**下一个像素**，而且**再下一个像素**的判别式

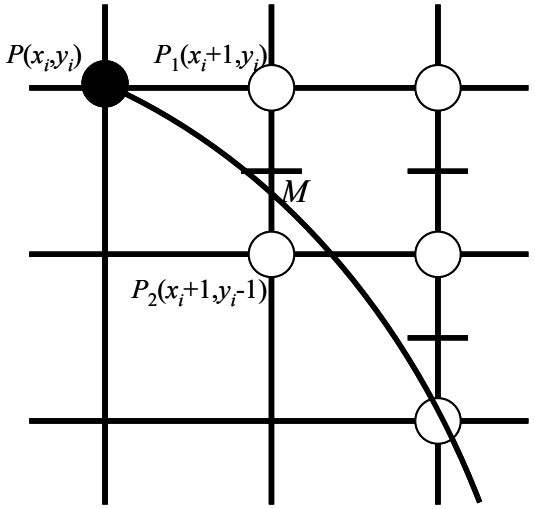
$$d' = F(x_i + 2, y_i - 1.5)$$

$$= (x_i + 2)^2 + (y_i - 1.5)^2 - R^2 = d + 2x_i - 2y_i + 5$$

第一个像素是(0,R)，判别式d的初始值为

d_0=F(1,R-0.5)=1+(R-0.5)^2-R^2=1.25-R。






```
void MidpointCircle(int R){
    int x,y;
    double d;
    x=0;y=R;d=1.25-R;
    SetPixel(x,y);
    while(x<y) {
        if(d<0) {
            d+=2*x+3;
            x++;
        }
        else {
            d+=2*(x-y)+5;
            x++;
            y--;
        }
        SetPixel(x,y);
    }
}
```

算法三：中点画圆法改进一

在上述算法中，使用了浮点数来表示判别式**d**。

简化算法，在算法中全部使用整数，使用**e=d-0.25**代替**d**。

初始化运算**d=1.25-R**对应于**e=1-R**。

由于**e**的初值为整数，且在运算过程中的增量也是整数，故**e**始终是整数，所以

判别式**d<0**对应于**e<-0.25**，等价于**e<0**。

判别式**d≥0**对应于**e≥-0.25**，等价于**e≥0**。


```
void MidpointCircle1(int R)
{    int x,y,d;
     x=0;y=R;d=1-R;
     SetPixel(x,y);
     while(x<y)
     {
          if(d<0)
          {d+=2*x+3;x++;}
          else
          {d+=2*(x-y)+5;x++;y--;}
          SetPixel(x,y);
     }
}
```

算法三：中点画圆法改进二

设Δx=2*x+3，Δy=-2*y+2，则

```
if(d<0)
{d+=Δx; Δx+=2;x++; }
else
{d+=(Δx+Δy);Δx+=2; Δy+=2;x++;y--;}
```

每当x递增1，Δx递增2。

每当y递减1，Δy递增2。

由于初始像素为(0,R)，所以Δx的初值为3，Δy的初值为-2R+2。

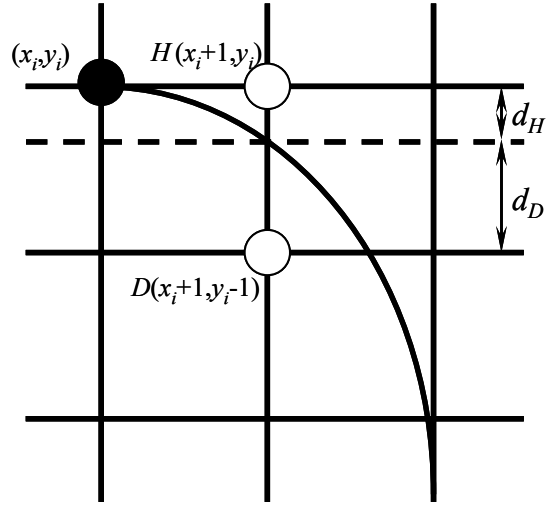
```
 if(d<0)
{d+=2*x+3;x++;}
else
{d+=2*(x-y)+5;x++;y--;}
```

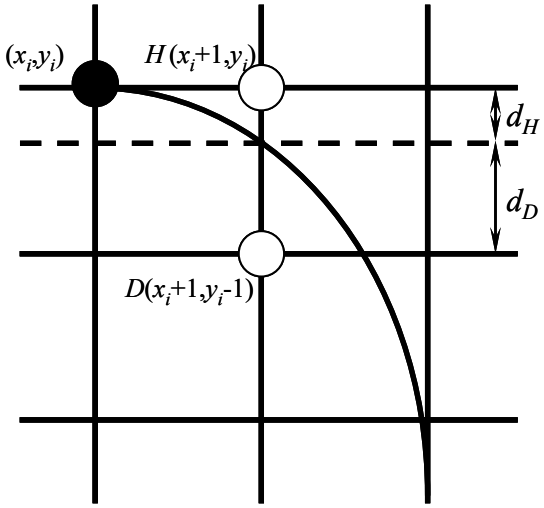

```
void MidpointCircle2(int R){
    int x,y,deltax,deltay,d;
    x=0;y=R;d=1-R;
    deltax=3;deltay=2-R-R;
    SetPixel(x,y);
    while(x<y){
        if(d<0) {
            d+=deltax; deltax+=2;
            x++;
        }
        else {
            d+=(deltax+deltay);
            deltax+=2;deltay+=2;
            x++;y--;
        }
        SetPixel(x,y);
    }
}
```

算法四：Bresenham画圆算法

在0≤x≤y的1/8圆周上，像素坐标x值单调增加，y值单调减少。

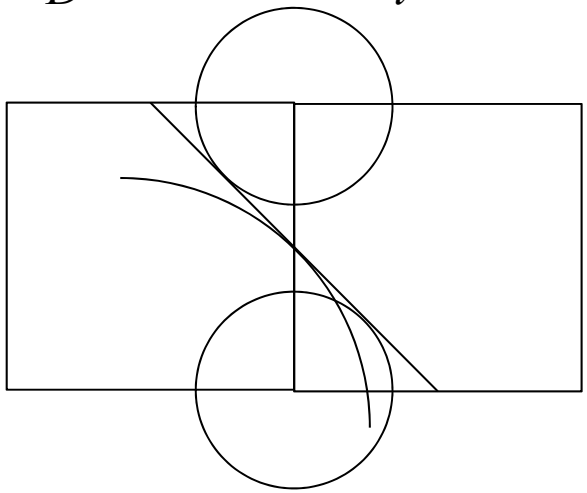
设第i步已确定（x_i,y_i）是要画圆上的像素点，看第i+1步像素点（x_{i+1},y_{i+1}）应如何确定。下一个像素点只能是（x_i+1,y_i）或者（x_i+1,y_i-1）中的一个





$$d_H = (x_i+1)^2 + y_i^2 - R^2$$

$$d_D = R^2 - (x_i+1)^2 - (y_i-1)^2$$

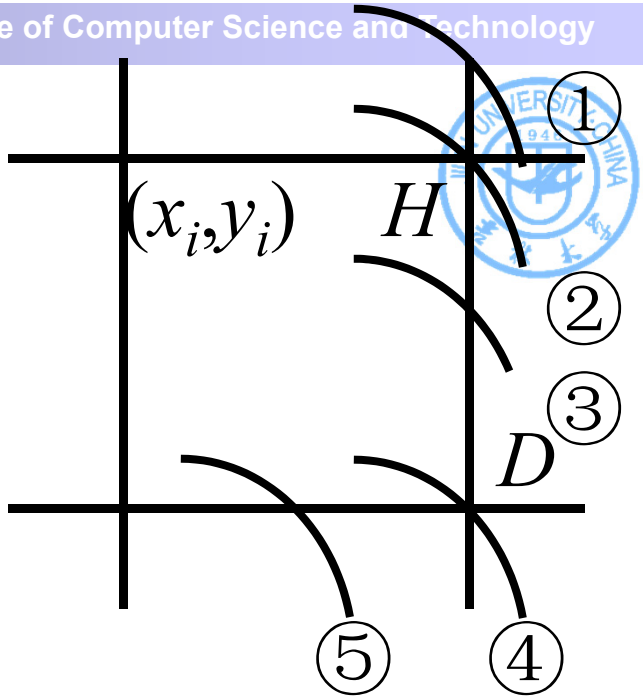


构造判别式：

$$p_i = d_H - d_D$$

$$= ((x_i+1)^2 + y_i^2 - R^2) - (R^2 - (x_i+1)^2 - (y_i-1)^2)$$

$$= 2(x_i+1)^2 + 2y_i^2 - 2y_i - 2R^2 + 1$$



构造判别式：$p_i = d_H - d_D$

1.精确圆弧是③，则$d_H > 0$和$d_D > 0$.

若$p_i < 0$，即$d_H < d_D$应选H点。

若$p_i \geq 0$，即$d_H \geq d_D$应选D点。

2.若精确圆弧是①或②，显然H是应选择点，而此时$d_H \leq 0$，$d_D > 0$，必有$p_i < 0$。

3.若精确圆弧是④或⑤，显然D是应选择点，而此时$d_H > 0$，$d_D \leq 0$，必有$p_i > 0$。

得出结论：p_i做判别量，$p_i < 0$时，选H点为下一个像素点；$p_i \geq 0$时，选D点为下一个像素点。

从p_i计算p_{i+1}

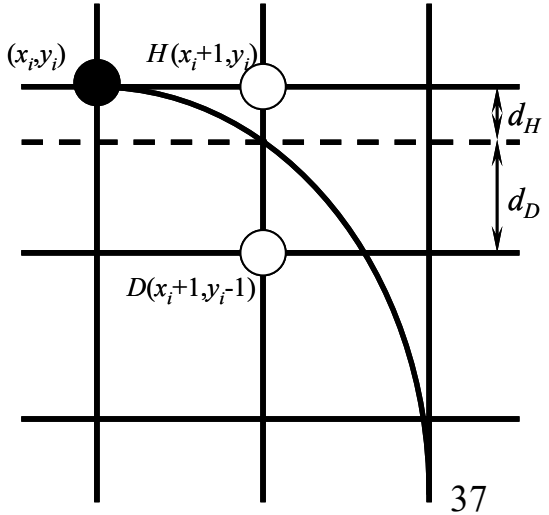
$$p_i = d_H - d_D$$

$$= ((x_i+1)^2 + y_i^2 - R^2) - (R^2 - (x_i+1)^2 - (y_i-1)^2)$$

$$= 2(x_i+1)^2 + 2y_i^2 - 2y_i - 2R^2 + 1$$

$$p_{i+1} = 2(x_{i+1}+1)^2 + 2y_{i+1}^2 - 2y_{i+1} - 2R^2 + 1$$

$$p_{i+1} - p_i = 4x_i + 6 + 2(y_{i+1}^2 - y_i^2 - y_{i+1} + y_i)$$



从p_i计算p_{i+1}

$$p_i = 2(x_i+1)^2 + 2y_i^2 - 2y_i - 2R^2 + 1$$

$$p_{i+1} - p_i = 2(y_{i+1}^2 - y_i^2 - y_{i+1} + y_i) + 4x_i + 6$$

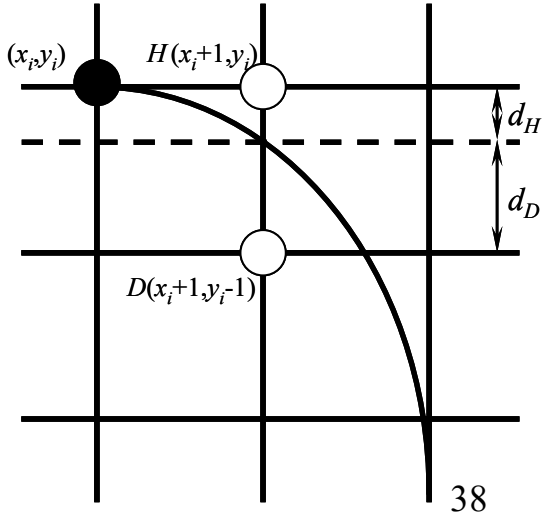
当$p_i \geq 0$时，应选D点，即选

$$y_{i+1} = y_i - 1,$$

$$p_{i+1} = p_i - 4y_i + 4 + 4x_i + 6 = p_i + 4(x_i - y_i) + 10$$

当$p_i < 0$时，应选H点，即选

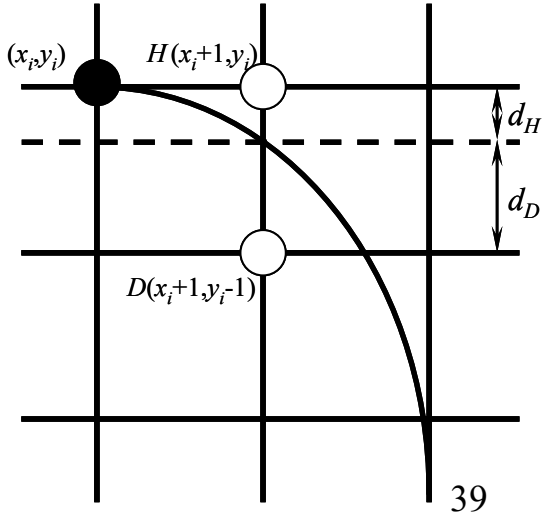
$$y_{i+1} = y_i, p_{i+1} = p_i + 4x_i + 6$$



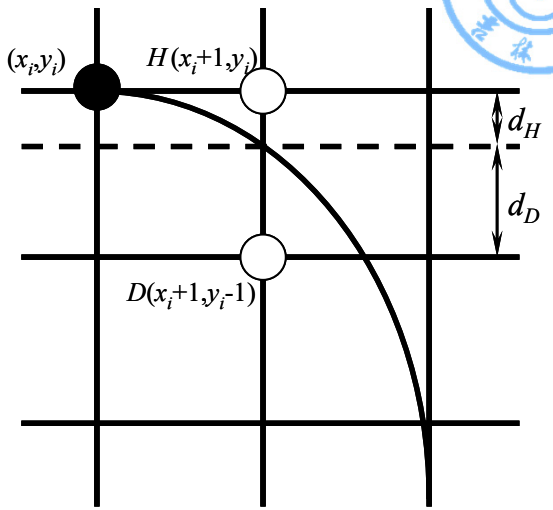
$$p_i = 2(x_i+1)^2 + 2y_i^2 - 2y_i - 2R^2 + 1$$

画圆的起始点是(0，R)，即x_1=0，y_1=R，代入前式，令i=1，就得到：

$$\begin{aligned} p_i &= 2(x_i+1)^2 + 2y_i^2 - 2y_i - 2R^2 + 1 \\ &= 2(0+1)^2 + 2R^2 - 2R - 2R^2 + 1 \\ &= 3 - 2R \end{aligned}$$



```
void BresenhamCircle(int R){
    int x,y,p;
    x=0;  y=R;
    p=3-2*R;
    for(;x<=y;x++){
        SetPixel(x,y);
        if(p>=0){
            p+=4*(x-y)+10;
            y--;
        }
        else {
            p+=4*x+6;
        }
    }
}
```

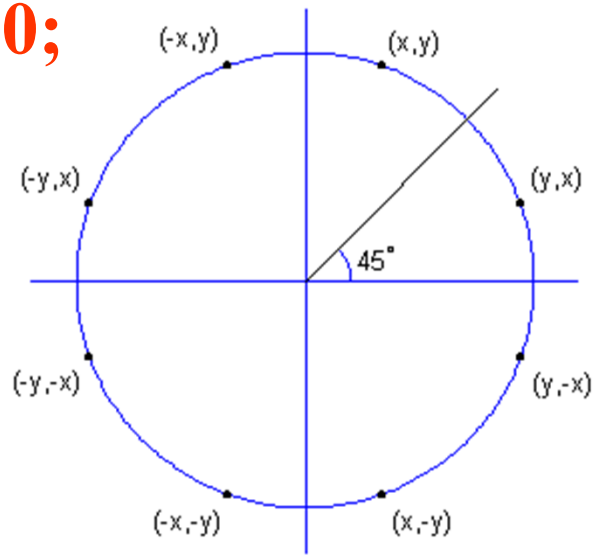


$$y_{i+1} = y_i - 1,$$
$$p_{i+1} = p_i + 4(x_i - y_i) + 10$$

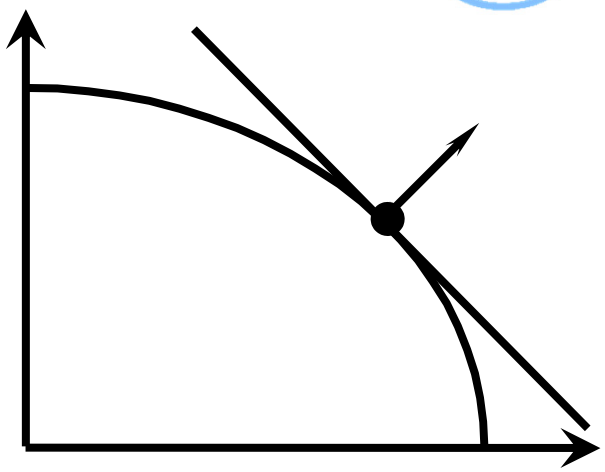
$$y_{i+1} = y_i,$$
$$p_{i+1} = p_i + 4x_i + 6$$

```
void BresenhamCircle(int R){
    int x,y,p;
    x=0;  y=R;
    p=3-2*R;
    for(;x<=y;x++){
        SetPixel(x,y);
        if(p>=0){
            p+=4*(x-y)+10;
            y--;
        }
        else {
            p+=4*x+6;
        }
    }
}
```

只需修改语句SetPixel(x,y)，画八个对称的点，就可以画出全部圆周。若加一个平移，就可以画出圆心在任意位置的圆周。



第三节 椭圆扫描转换算法



中点画圆法推广到一般的二次曲线

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

椭圆上一点(x,y)处的法向量为

$$N(x,y)=\frac{\partial F}{\partial x}i+\frac{\partial F}{\partial y}j=2b^2xi+2a^2yj$$

(i,j)为x轴, y轴的单位向量

上半部分： $(x_p,y_p),M(x_p+1,y_p-0.5)$,法向量的$y$的分量更大

下半部分： $(x_p,y_p),M(x_p+0.5,y_p-1)$,法向量的$x$的分量更大

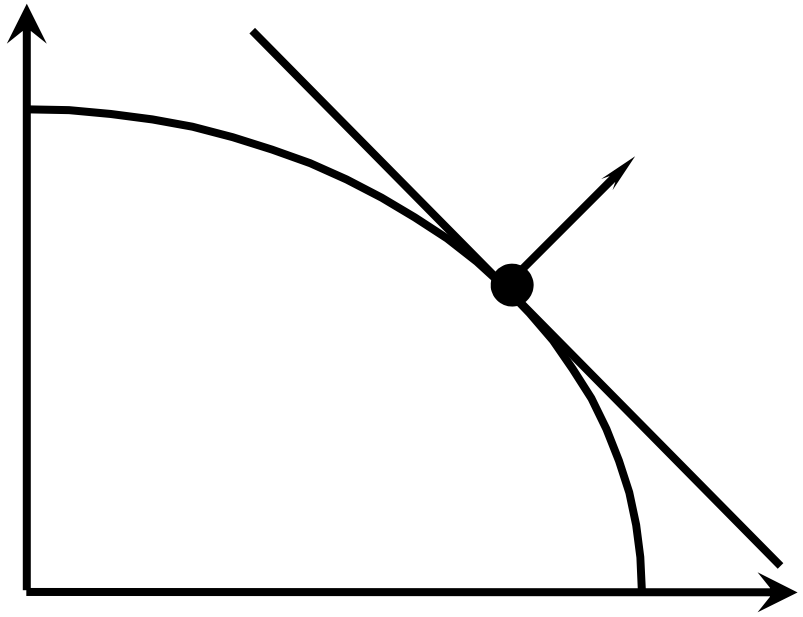
椭圆的扫描转换算法:

$F(x,y)=b^2x^2+a^2y^2-a^2b^2$

法向量为（$2b^2x, 2a^2y$)

当前中点:$2b^2(x_i+1)<2a^2(y_i-0.5)$上半部分

下一个中点:$2b^2(x_i+1)>2a^2(y_i-0.5)$下半部分



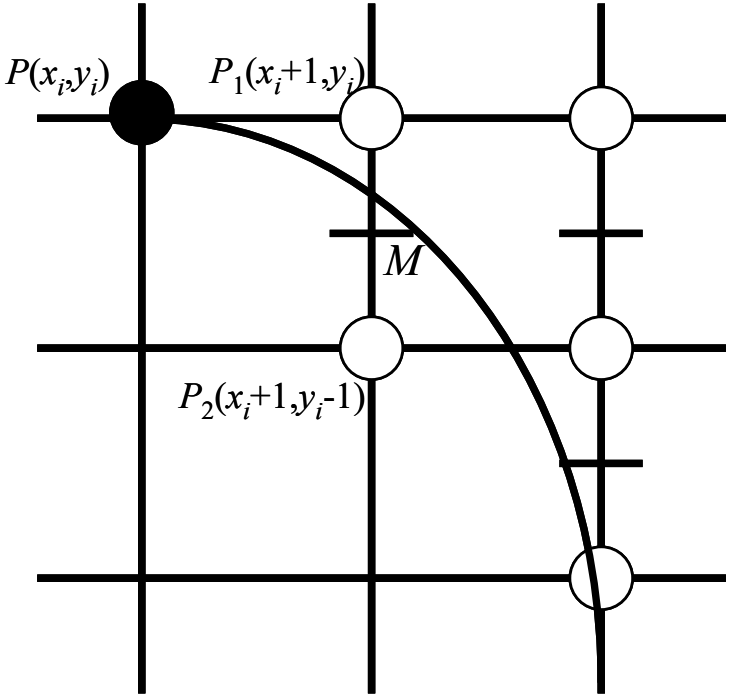
上半部分:$(x_i, y_i), (x_i + 1, y_i - 0.5)$

$$d_1 = F(x_i + 1, y_i - 0.5) = b^2(x_i + 1)^2 + a^2(y_i - 0.5)^2 - a^2b^2$$

$d_1 < 0$,中点在椭圆内，取正右方象素

$$d_1' = F(x_i + 2, y_i - 0.5) = b^2(x_i + 2)^2 + a^2(y_i - 0.5)^2 - a^2b^2$$

$$= d_1 + b^2(2x_i + 3)$$



上半部分:$(x_i, y_i), (x_i + 1, y_i - 0.5)$

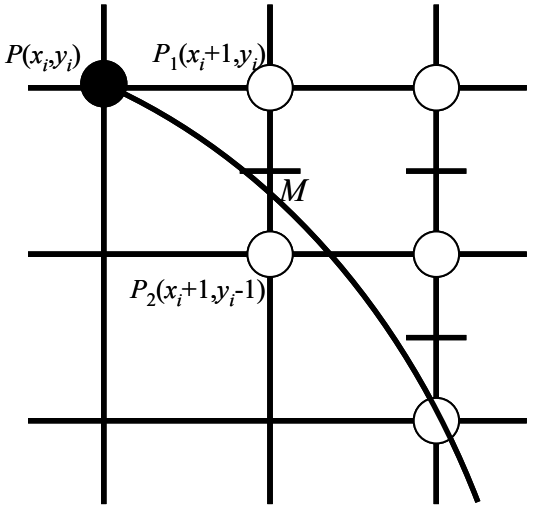
$$d_1 = F(x_i + 1, y_i - 0.5) = b^2(x_i + 1)^2 + a^2(y_i - 0.5)^2 - a^2b^2$$

$d_1 > 0$,中点在椭圆外，取右下方象素

$$d_1' = F(x_i + 2, y_i - 1.5) = b^2(x_i + 2)^2 + a^2(y_i - 1.5)^2 - a^2b^2$$

$$= d_1 + b^2(2x_i + 3) + a^2(-2y_i + 2)$$

d_1初值 $d_1^0 = F(1, b - 0.5) = b^2 + a^2(-b + 0.25)$



下半部分:(x_i, y_i),下一个点为$(x_i, y_i - 1)$或者$(x_i + 1, y_i - 1)$

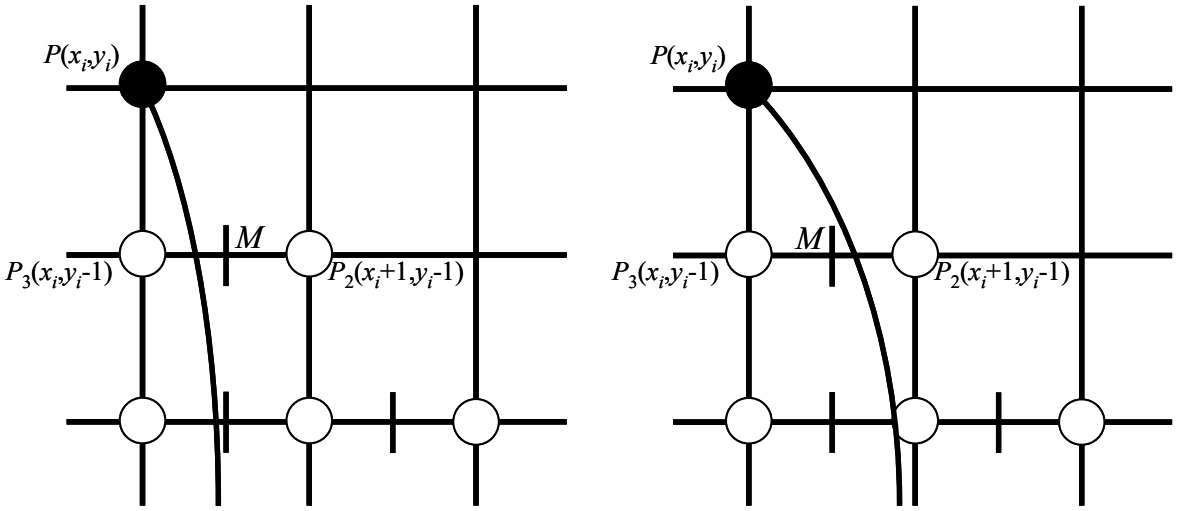
$$M = (x_i + 0.5, y_i - 1)$$

$$d_2 = F(x_i + 0.5, y_i - 1) = b^2(x_i + 0.5)^2 + a^2(y_i - 1)^2 - a^2b^2$$

$d_2 < 0$,中点在椭圆内，取右下方象素

$$d_2' = F(x_i + 1.5, y_i - 2) = b^2(x_i + 1.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$

$$= d_2 + b^2(2x_i + 2) + a^2(-2y_i + 3)$$



下半部分:(x_i, y_i),下一个点为$(x_i, y_i - 1)$或者$(x_i + 1, y_i - 1)$

$$d_2 = F(x_i + 0.5, y_i - 1) = b^2(x_i + 0.5)^2 + a^2(y_i - 1)^2 - a^2b^2$$

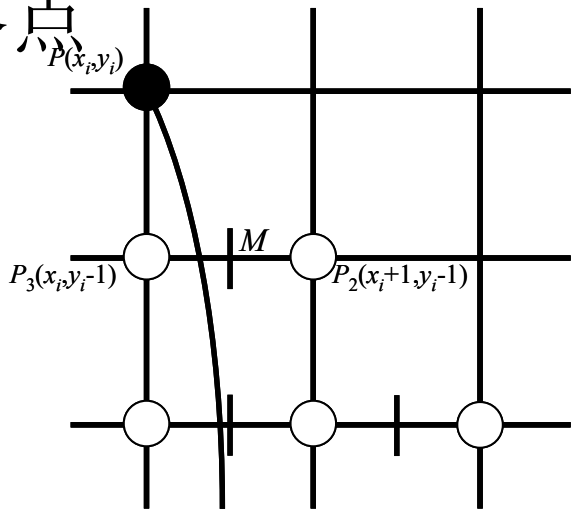
$d_2 \geq 0$,中点在椭圆外，取正下方象素

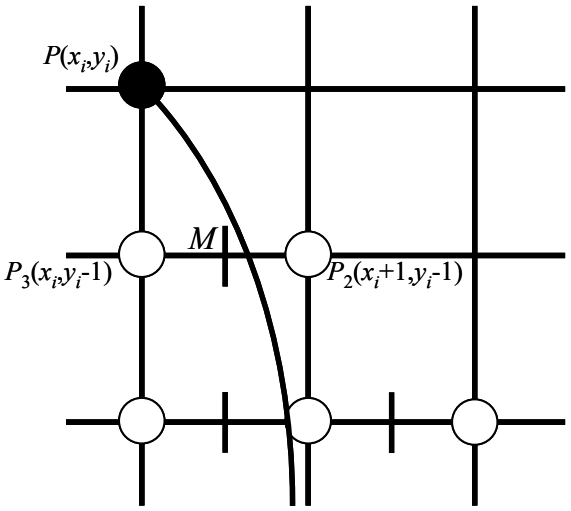
$$d_2' = F(x_i + 0.5, y_i - 2) = b^2(x_i + 0.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$

$$= d_2 + a^2(-2y_i + 3)$$

d_2初值$d_2^0 = F(x, y) = (b(x + 0.5))^2 + (a(y - 1))^2 - a^2b^2$

(x, y)为交界点






```
void MidpointEllipse(int a,int b){
    int x,y;
    double d1,d2;
    x=0;y=b;
    d1=b*b+a*a*(-b+0.25);
    SetPixel(x,y);
    while(b*b*(x+1)<a*a*(y-0.5)) {
        if(d1<0) {
            d1+=b*b*(2*x+3);x++;
        }else{
            d1+=(b*b*(2*x+3)+a*a*(-2*y+2));x++;y--;
        }
        SetPixel(x,y);
    }
    d2=b*b*(x+0.5)*(x+0.5)+a*a*(y-1)*(y-1)-a*a*b*b;
    while(y>0){
        if(d2<0){
            d2+=b*b*(2*x+2)+a*a*(-2*y+3);x++;y--;
        }else{
            d2+=a*a*(-2*y+3);y--;
        }
        SetPixel(x,y);
    }
}
```

第四节 区域填充

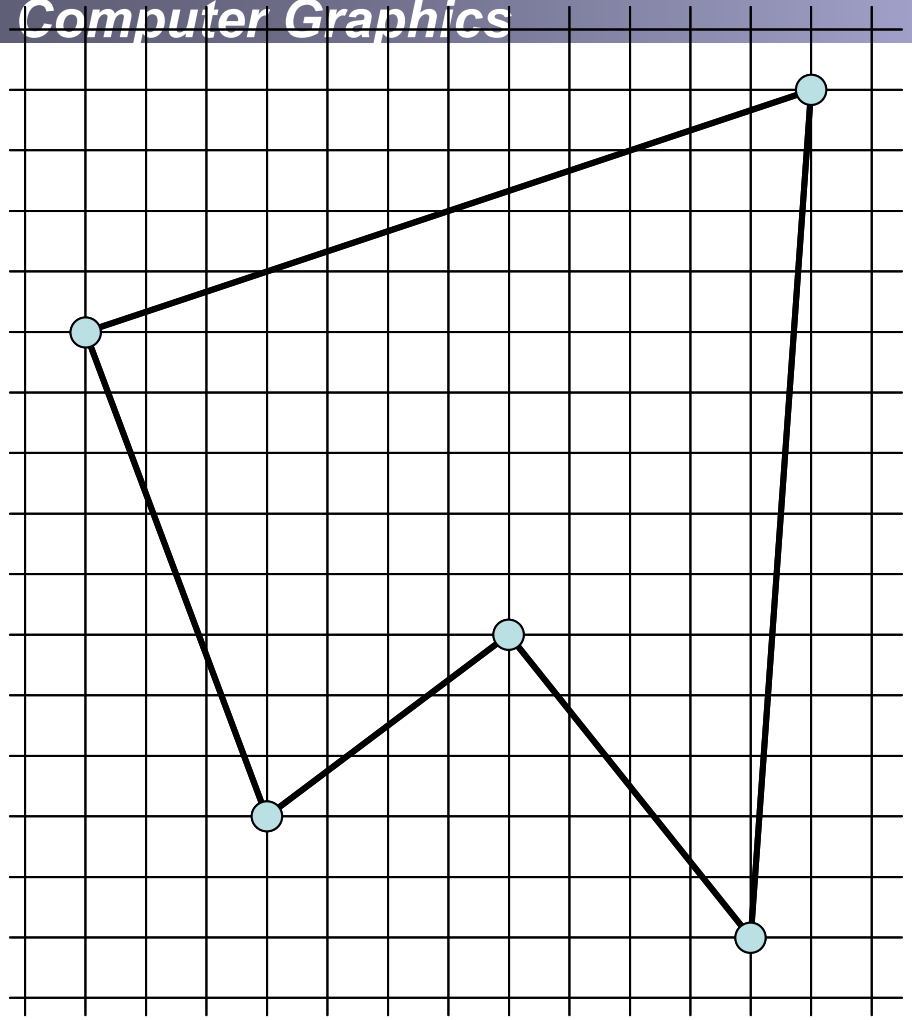
- 种子填充算法
- 扫描线填充算法
- 多边形扫描转换算法
- 边填充算法
- 图案填充

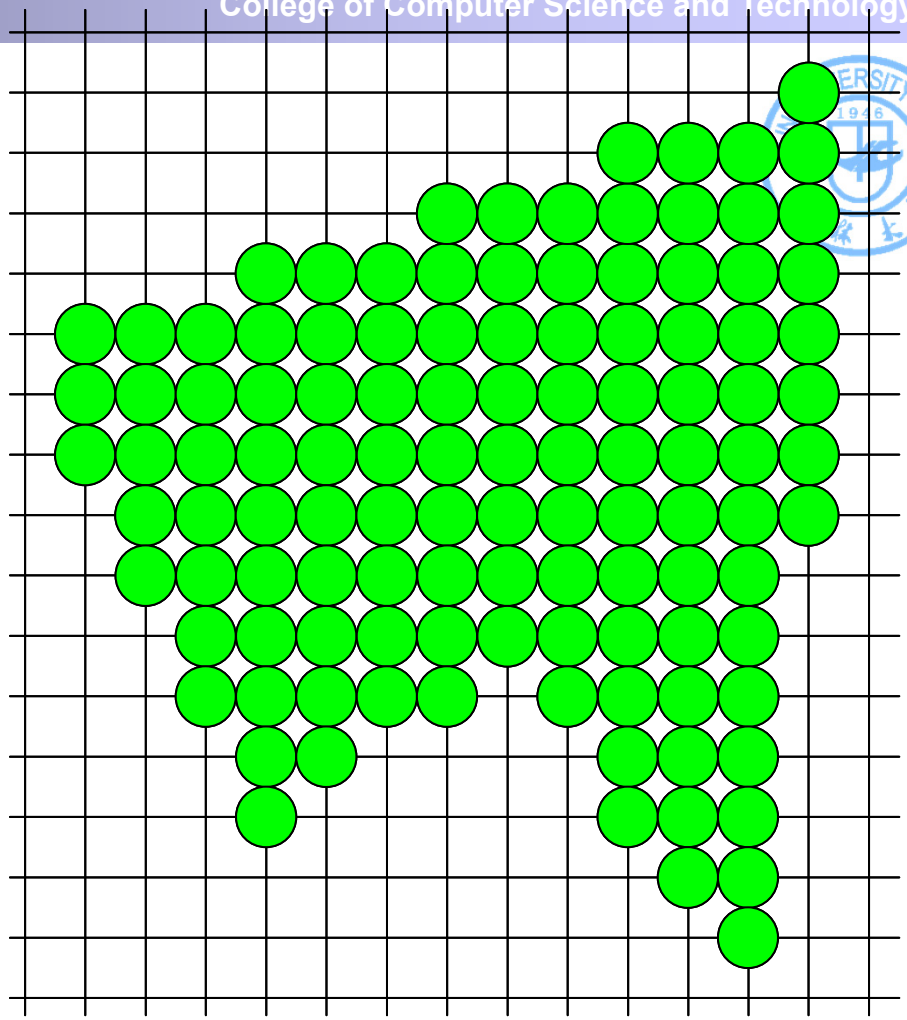
一、种子填充算法

1. 概念

区域：指光栅网格上的一组像素。

区域填充：把某确定的像素值送入到区域内部的所有像素中。





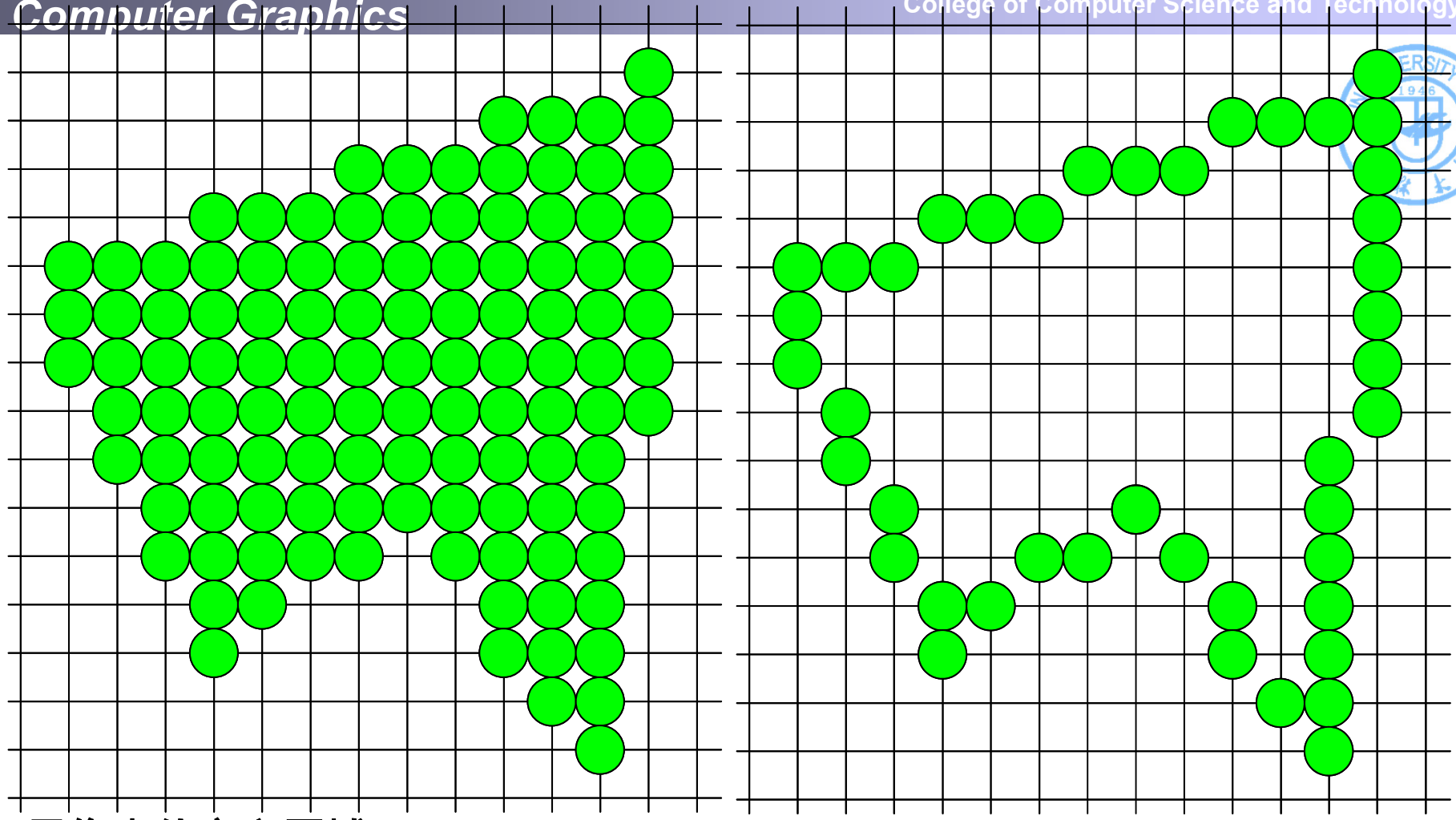
2.区域填充方法分为两大类。

(1)区域由多边形围成，区域由多边形的顶点序列来定义；

优点：占用内存少，缺点：需要区分内外侧

(2)是通过像素的值来定义区域的内部

优点：形状可任意复杂，可直接着色　缺点：几何意义不直观



3.用像素值定义区域

（1）内定义区域(oldvalue)指出区域内部所具有的像素值,（内点表示）

（2）边界定义区域(boundaryvalue)，指出区域边界所具有的像素值。此时区域边界上所有像素具有某个边界boundaryvalue。区域的边界应该是封闭的，指明区域的内部和外部。

以像素为基础的区域填充主要是依据区域的连通性进行。

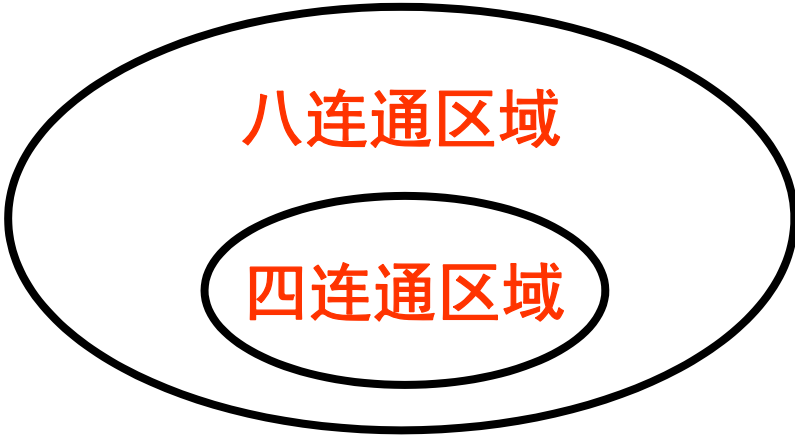
4. 区域的连通性

（1）四连通：从区域中的一个像素出发，经连续地向上下左右四个相邻像素的移动，就可以到达区域内的任意另一个像素。

（2）八连通：如果除了要经上下左右的移动，还要经左上、右上、左下和右下的移动，才能由一个像素走到区域中另外任意一个像素.

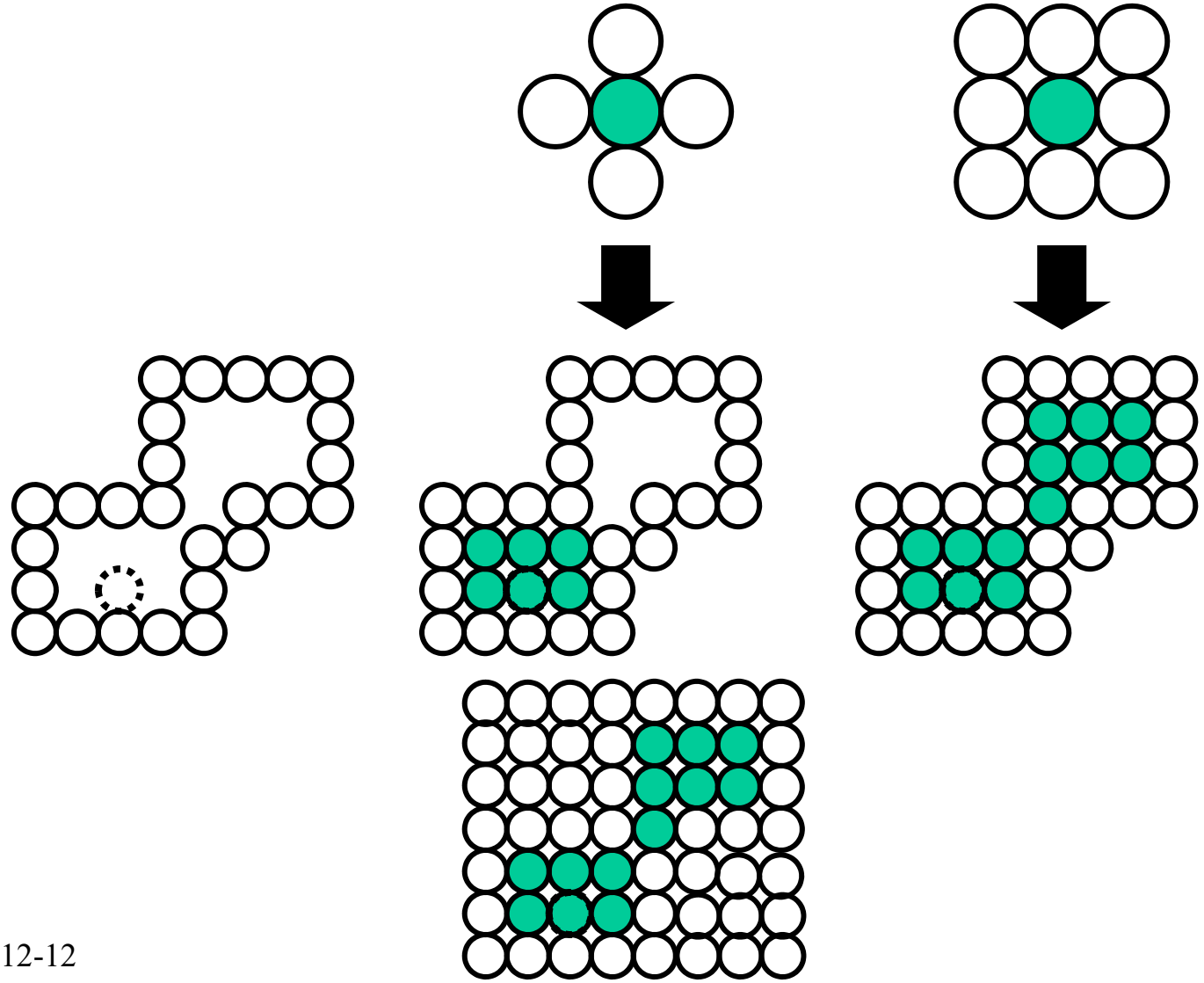
四连通区域必定是八连通区域，反之不一定。

左上	上	右上
左	P	右
左下	下	右下



4. 区域的连通性

（1）四连通 （2）八连通



5. 种子填充算法

利用区域的连通性进行区域填充的要素：

- 明确定义区域，
- 事先给定一个区域内部像素，称为种子。

做区域填充时，要对光栅网格进行遍历，找出由种子出发能达到而又不穿过边界的所有像素。

这种利用连通性的填充，

优点：不受区域不规则性的影响，

缺点：需要事先知道一个内部像素。

（1）四连通内定义区域填充算法：

```
void   Floodfill(int x,int y,COLORREF oldvalue,COLORREF
newvalue){
/*(x,y)为种子  oldvalue是原值  newvalue是新值，应不等于原
值。*/
{        //是否在区域内并且尚未被访问过
        if (GetPixel(x,y) == oldvalue){
     {
                SetPixel(x,y,newvalue); //赋值为新值
                Floodfill(x,y-1,oldvalue,newvalue);//向上扩散
                Floodfill(x,y+1,oldvalue,newvalue); //向下扩散
                Floodfill(x-1,y,oldvalue,newvalue); //向左扩散
                Floodfill(x+1,y,oldvalue,newvalue); //向右扩散
        }
}
```

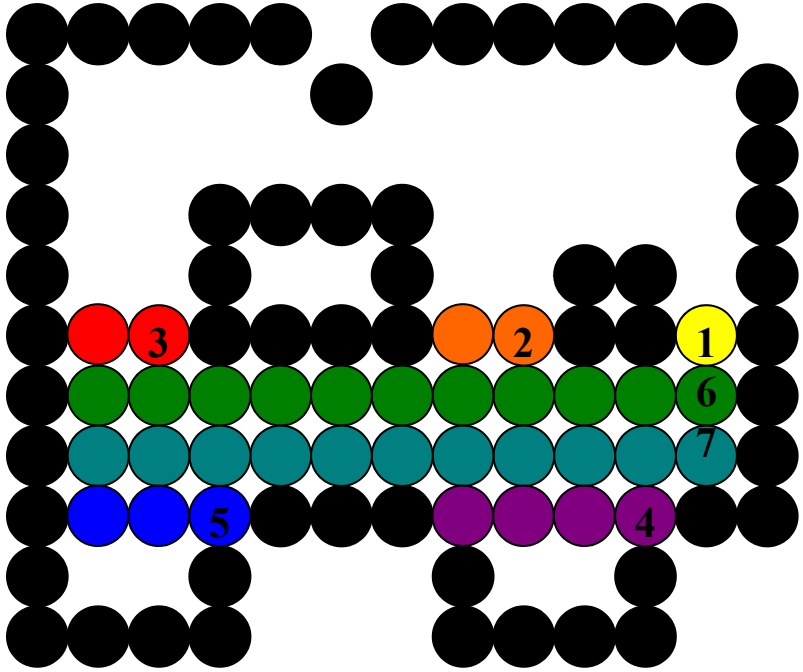
优点：算法简单 缺点：重复多

（2）四连通边界定义区域填充算法：

```
void   Boundaryfill(int x,int y,COLORREF
boundaryvalue,COLORREF newvalue)
/*(x,y) 为种子位置
boundaryvalue是边界像素值
newvalue是区域内像素新值，未填充前区域内不应有值为
newvalue的像素。*/
{       if( GetPixel(x,y)!=boundaryvalue
        && GetPixel(x,y)!=newvalue) //未达边界且未访问过
        {       SetPixel(x,y,newvalue);//赋以新值
                //向四个方向扩散。
                Boundaryfill(x,y-1,boundaryvalue,newvalue);
                Boundaryfill(x,y+1,boundaryvalue,newvalue);
                Boundaryfill(x-1,y,boundaryvalue,newvalue);
                Boundaryfill(x+1,y,boundaryvalue,newvalue);
        }
} //算法简单，多层递归，存储空间有限—>堆栈溢出
```

二、扫描线填充算法

- 扫描线种子填充算法(适用于边界定义的四连通区域)
- 像素段：将区域内由边界点限定的同一行内相连 接的不具有新值newvalue的一组像素称为一个像素段，像素段用它最右边的像素来标识。



算法的步骤如下：

1.[初始化]将堆栈置为空，将给定的种子点(x,y)压入堆栈

2.[出栈]如果堆栈为空，算法结束，否则取栈顶元素(x,y) 作为种子点

3.[区段填充]从种子点(x,y)开始沿当前扫描线向左右两个方向逐个像素进行填充，直到遇到边界点为止

4.[定范围]用xl和xr分别表示在步骤3中填充的区段的左右两个端点的横坐标

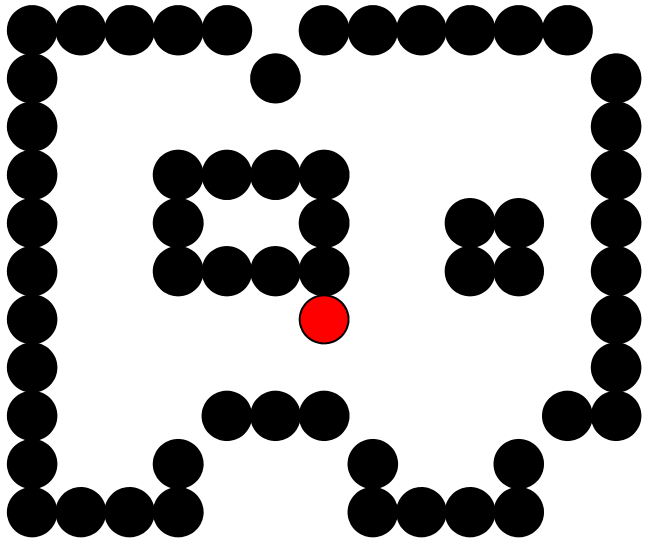
5.[进栈]对当前扫描线上下相邻的两条扫描线从右向左的确定位于[xl,xr]区域内的像素段。

如果区域内的像素未填充且不是边界，则取像素段的右端点作为种子点，压入堆栈，再转到步骤2，

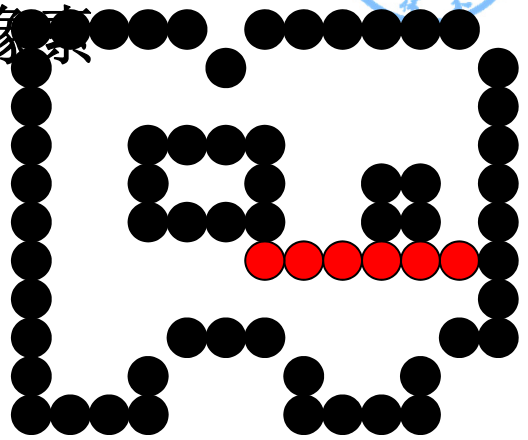
否则转到步骤2。

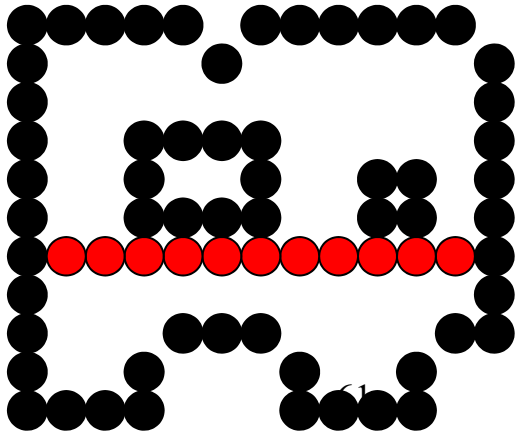
下面我们用伪C语言写出扫描线填充算法。

```
void ScanlineSeedfill(int x,int y,COLORREF
                      boundaryvalue,COLORREF newvalue)
{
    int x0,xl,xr,y0,xid;
    int flag;
    Stack s;
    Point p;
    s.push(Point(x,y));//种子像素入栈
    while(!s.isempty())
    {
        p=s.pop(); //栈顶像素出栈
        x=p.x;y=p.y;
        SetPixel(x ,y ,newvalue);
```

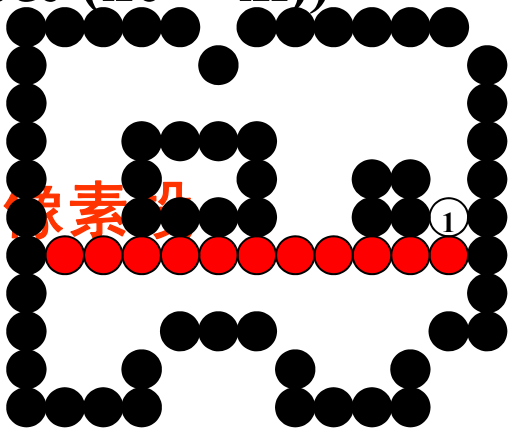


```
x0= x+1;
while(GetPixel(x0,y)!=boundaryvalue
&& GetPixel(x0,y)!=newvalue)//填充右方像素
{
        SetPixel(x0 ,y ,newvalue);
        x0++;
}
xr=x0-1;//最右边像素
x0= x-1;
while(GetPixel(x0,y)!=boundaryvalue
&& GetPixel(x0,y)!=newvalue)//填充左方像素
{
        SetPixel(x0 ,y ,newvalue);
        x0--;
}
xl=x0+1;//最左边像素
```

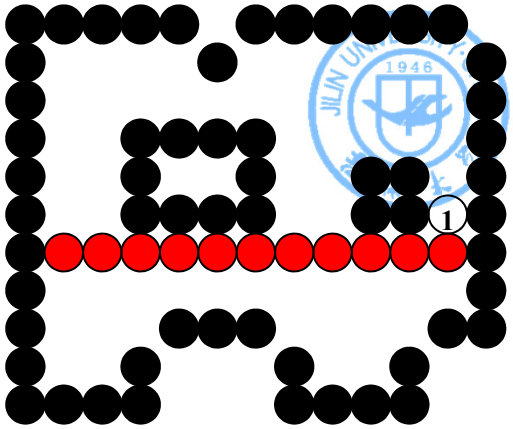


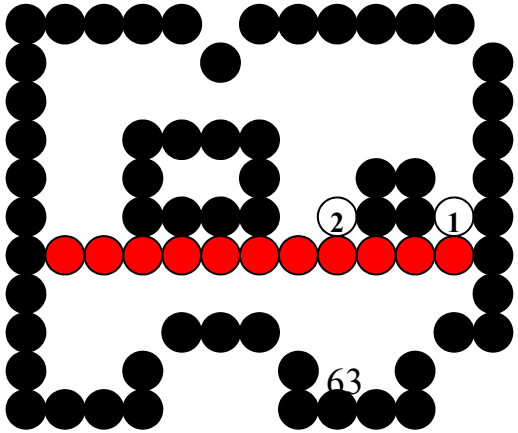


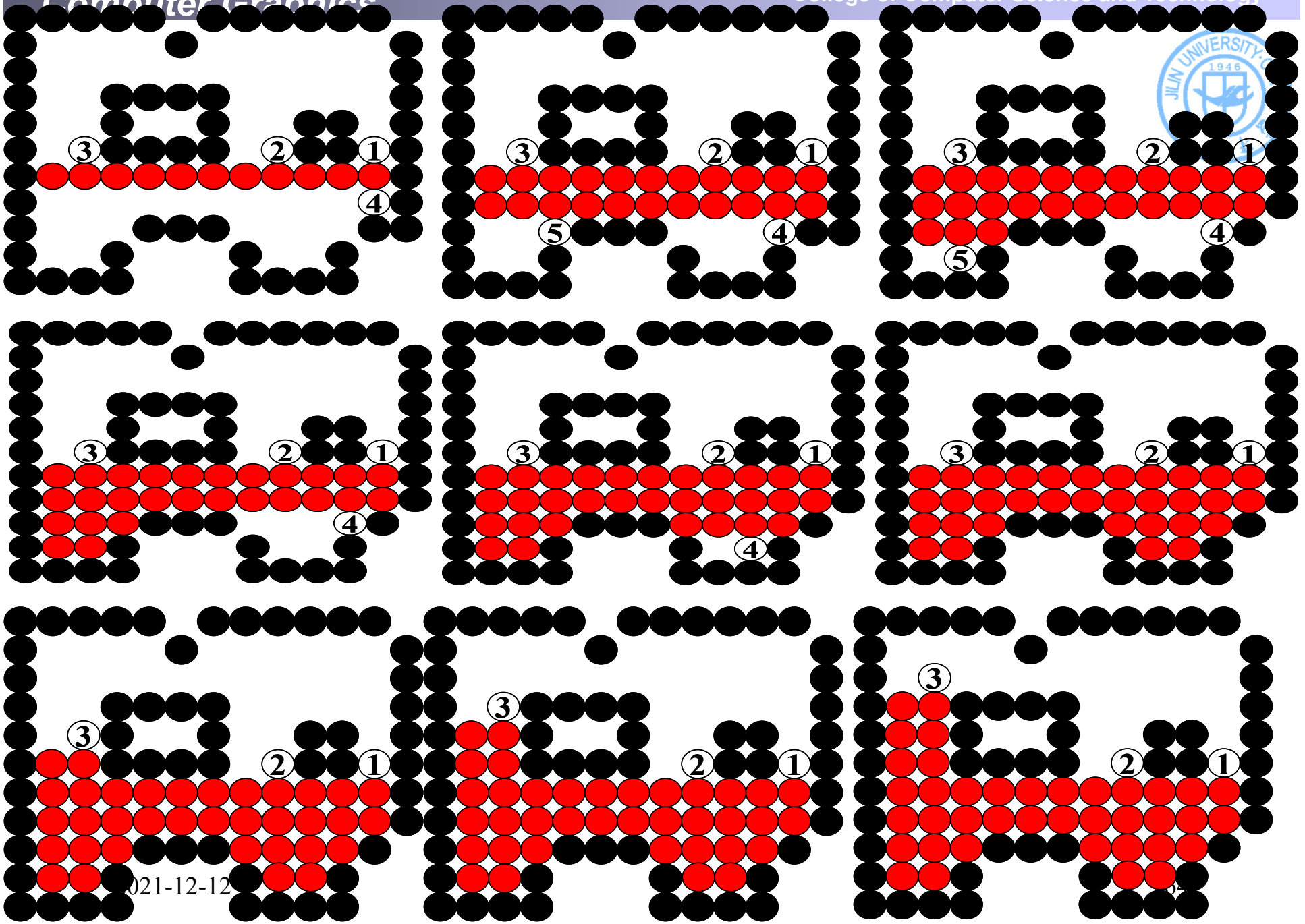
```
//检查上一条扫描线和下一条扫描线，
//若存在非边界且未填充的像素，
//则选取代表各连续区间的种子像素入栈。
y0=y;
for(int i=1;i>=-1;i-=2)//从最右像素开始寻找上下两行的像素段
{       x0=xr;
        y=y0+i;
        while(x0>=xl)
        {       flag=0;//是否找到像素段的标识，是为1，否为0
                while((GetPixel(x0,y)!=boundaryvalue)&&
                        (GetPixel(x0,y)!=newvalue) && (x0>=xl))
                {
                        if(flag==0)
                        {       flag=1;//标志找到新的像素段
                                xid=x0;
                        }
                        x0--;
                }
```

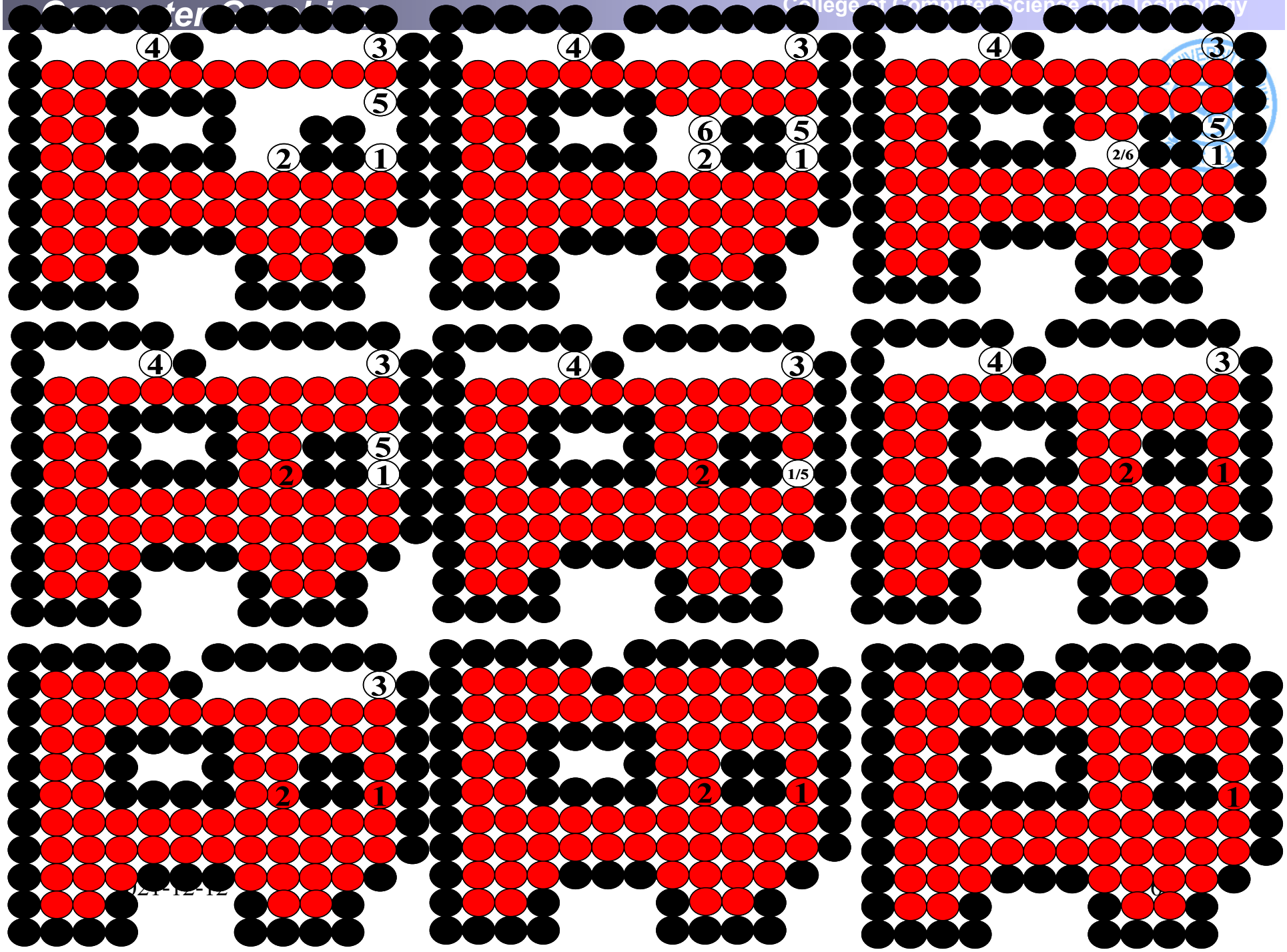


```
//将最右侧可填充像素压入栈中
                    if(flag==1)
                    {
                            s.push(Point(xid,y));
                            flag=0;
                    }
//检查当前填充行是否被中断，若被中断，寻找左方第一个可填
  充像素
//判断当前点是否为边界点//判断当前点是否为已填充点
                    while((GetPixel(x0,y)==boundaryvalue
                                    ||(GetPixel(x0,y)==newvalue))
                                    &&(x0>=xl))
                            x0--;
                    }//while(x0>=xl)
            }//for(int i=1;i>=-1;i-=2)
        }//while(!s.isempty())
}
```



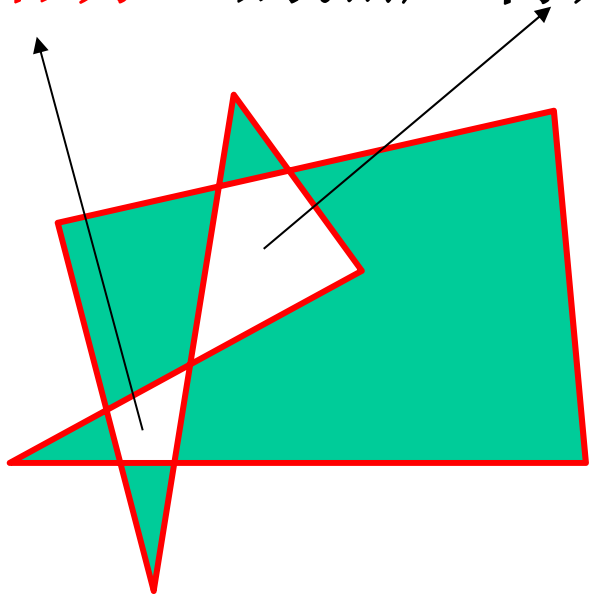



3
2
1
4
3
2
1
5
4
3
2
1
5
4
3
2
1
4
3
2
1
4
3
2
1
3
2
1
3
2
1
3
2
1

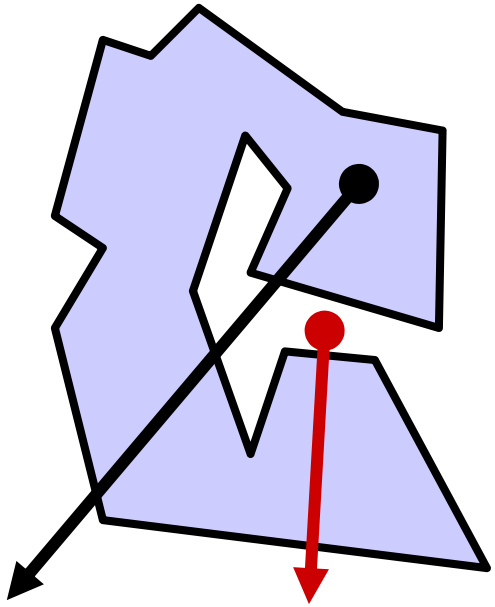

4
3
5
2
1
4
3
6
5
2
1
4
3
5
2/6
1
4
3
5
2
1
4
3
2
1/5
4
3
2
1
3
2
1
2
1
1

三、多边形的扫描转换算法

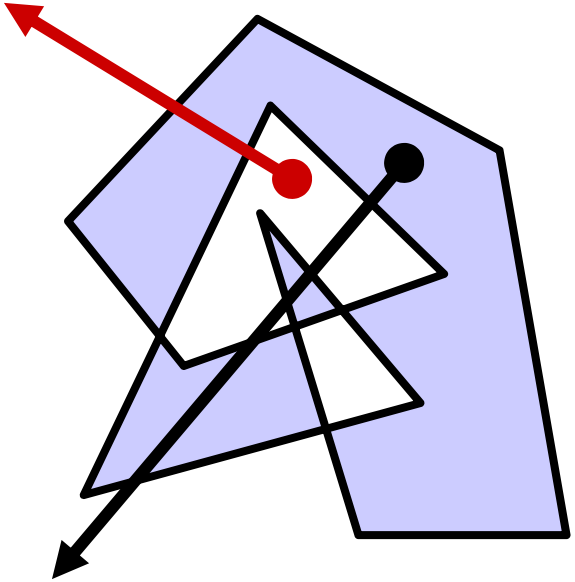
“奇偶”性质：即一条直线与任意封闭的曲线相交时，总是从第一个交点进入内部，再从第二个交点退出，以下交替的进入退出，即奇数次进入，偶数次退出。当然可能有一些“相切”的点应特殊处理。



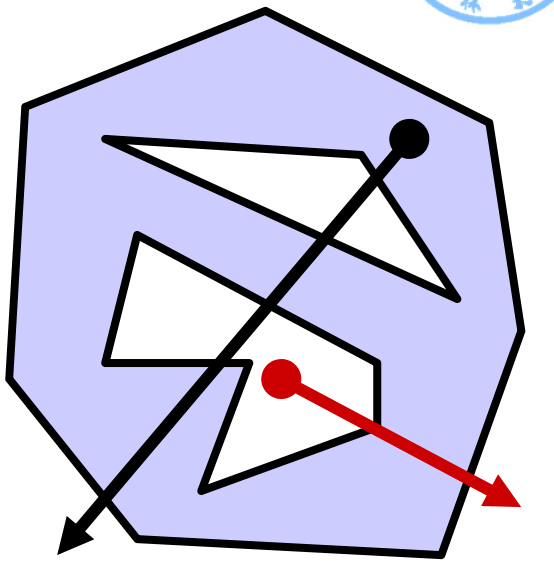
- 奇偶规则



凹多边形



自相交多边形



带有空洞的多边形

三、多边形的扫描转换算法

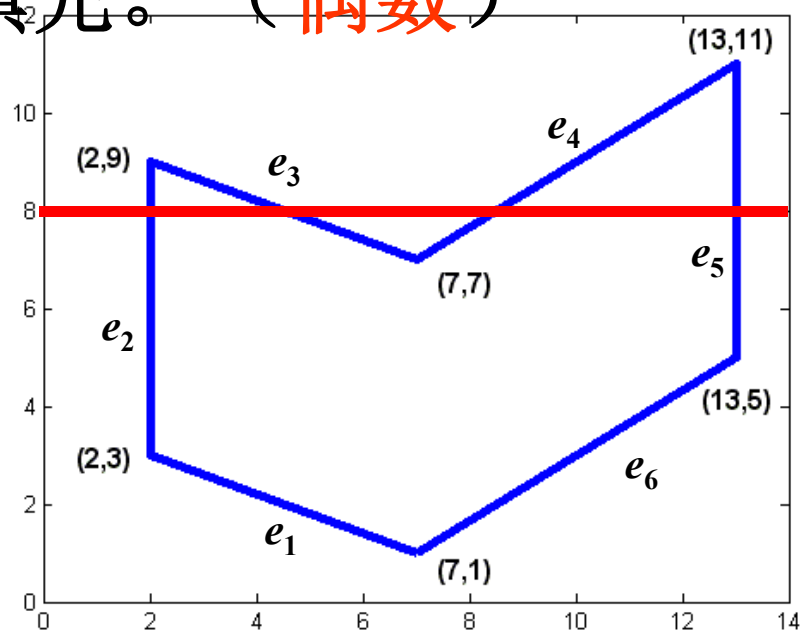
扫描转换算法步骤：

对于每一条扫描线

step1．找出扫描线与多边形边界线的所有交点；

step2．按x坐标增加顺序对交点排序；

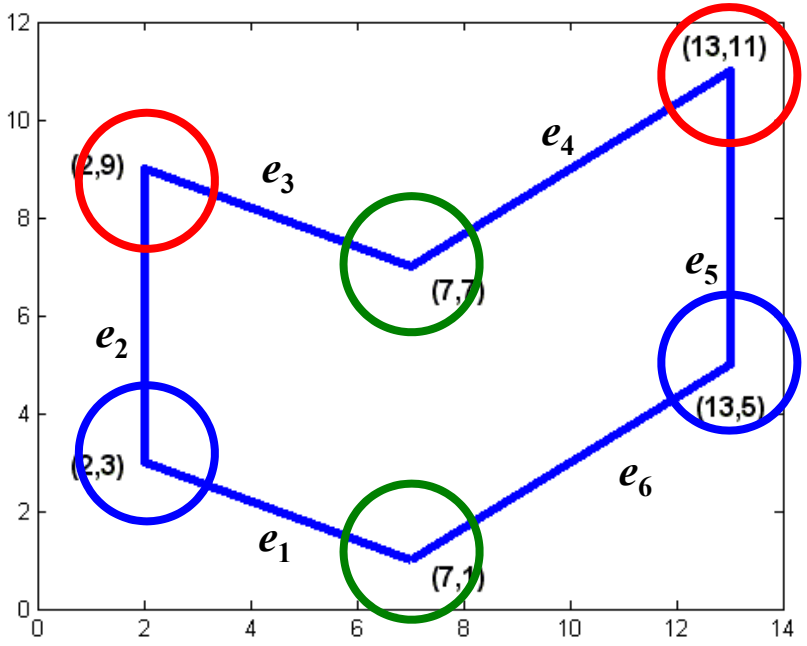
step3．在交点对之间进行填充。（偶数）



三、多边形的扫描转换算法

局部极大：如果一个顶点的前面和后面各有一些相邻顶点的y坐标，都小于该顶点的y坐标，则这个顶点是局部极大。

局部极小：如果一个顶点的前面和后面各有一些相邻顶点的y坐标，都大于该顶点的y坐标，则这个顶点是局部极小。



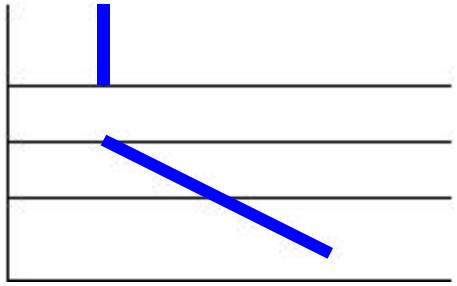
三、多边形的扫描转换算法

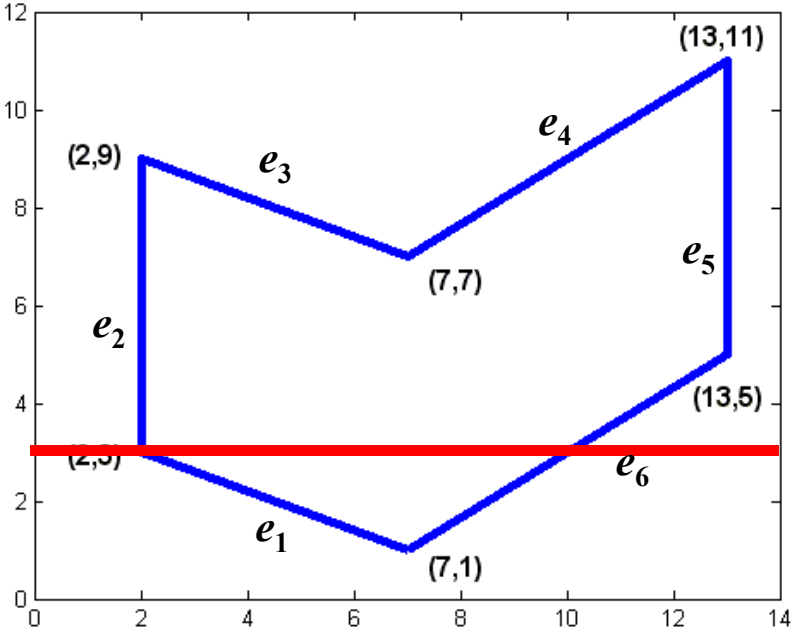
问题一：交点个数是奇数

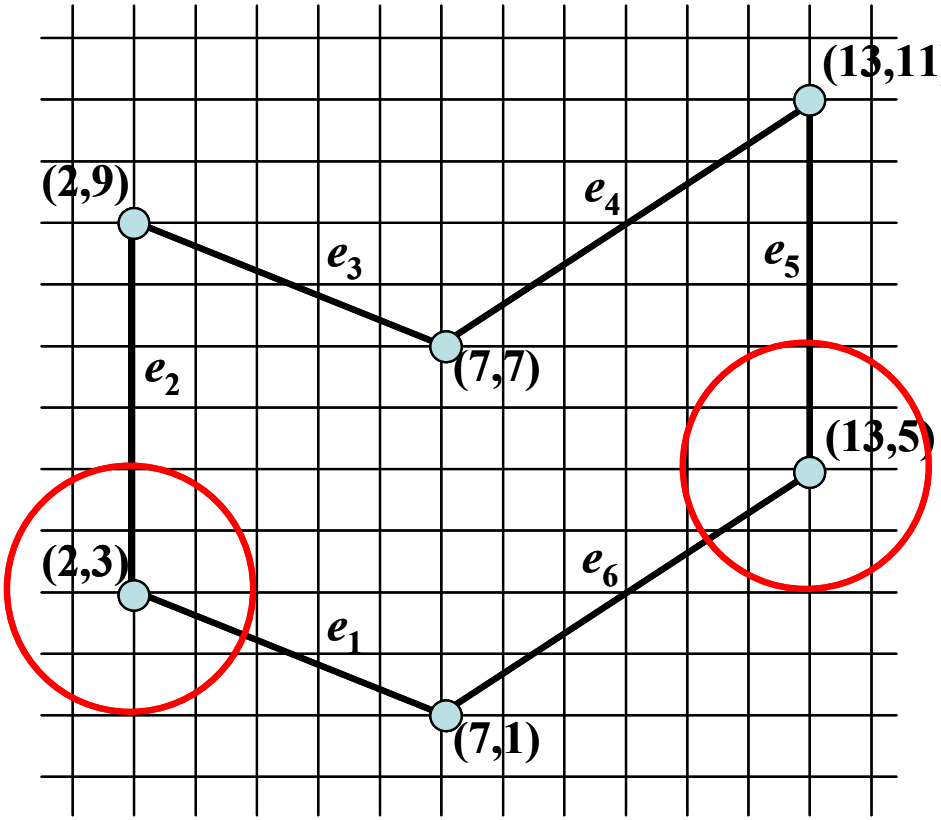
奇数个交点的特殊处理：

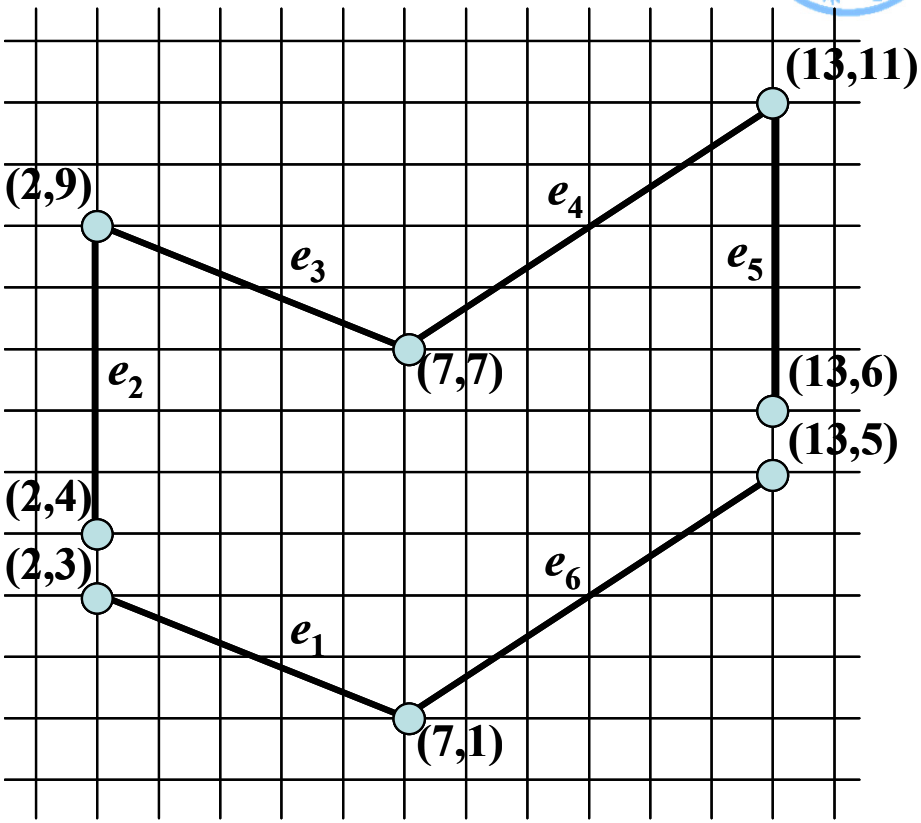
发生奇数个交点的情况出现在扫描线穿过非局部极值点，该非局部极值点看做是一个顶点，将其上面的边缩短一个单位

对非极值点的简便处理






(13,11)
(2,9)
e_4
e_3
e_5
e_2
(7,7)
(13,5)
(2,3)
e_6
e_1
(7,1)

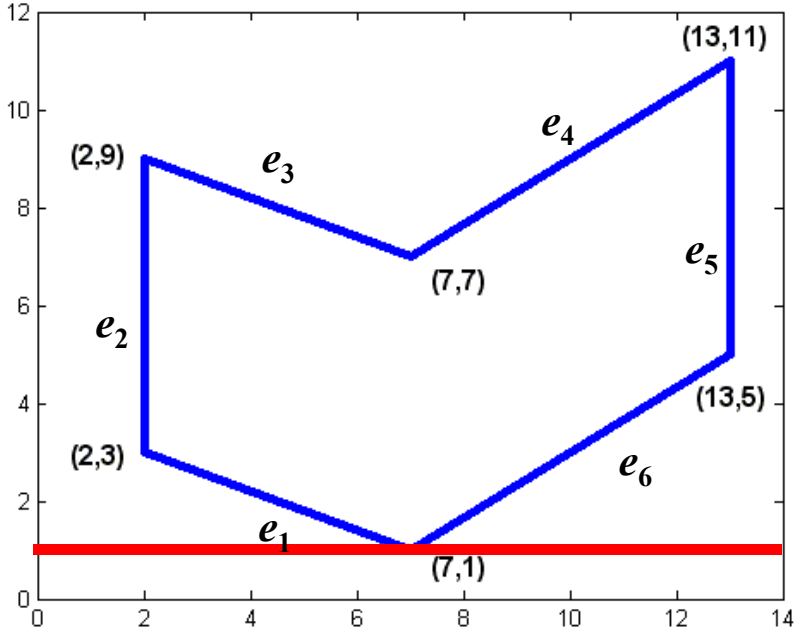

(13,11)
(2,9)
e_4
e_3
e_5
e_2
(7,7)
(13,6)
(13,5)
(2,4)
(2,3)
e_6
e_1
(7,1)

三、多边形的扫描转换算法

问题二：“相切”是否满足奇偶规则?

“相切”点的特殊处理：不做处理

与平行的扫描线“相切”的切点一定是局部极大或局部极小。边的“切点”一定是偶数个，满足奇偶规则，不做处理。



顶点处理

当顶点表现为是局部极大或局部极小时，就看做是二个，否则看做一个。

- 局部极大或局部极小点，交点看做是二个
- 非局部极值点，交点看做一个

step1．如何计算扫描线与多边形边界线的所有交点？

若扫描线y_i与多边形边界线交点x的坐标是x_i，则对下一条扫描线y_{i+1}，它与那条边界线的交点的x坐标x_{i+1}，可如下求出：

$$m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i},\ y_{i+1} = y_i + 1 \Rightarrow x_{i+1} = x_i + \frac{1}{m}$$

边表ET：记录多边形的所有边，按边的下端点y坐标递增排序

“吊桶”：记录一条边的信息

y_{max}	x_{min}	1/m	next

y_{max}:边的上端点(具有较大y坐标的端点)的y坐标

x_{min}:边的下端点(具有较小的y坐标的端点)的x坐标

1/m:斜率的倒数

next:指向下一条边的指针

活跃边：与当前扫描线相交的边

活跃边表AET：存贮当前扫描线相交的各边的表。

y_{max}	x	1/m	next

y_{max}:当前边的上端点的y坐标

x: 当前边与扫描线交点的x坐标

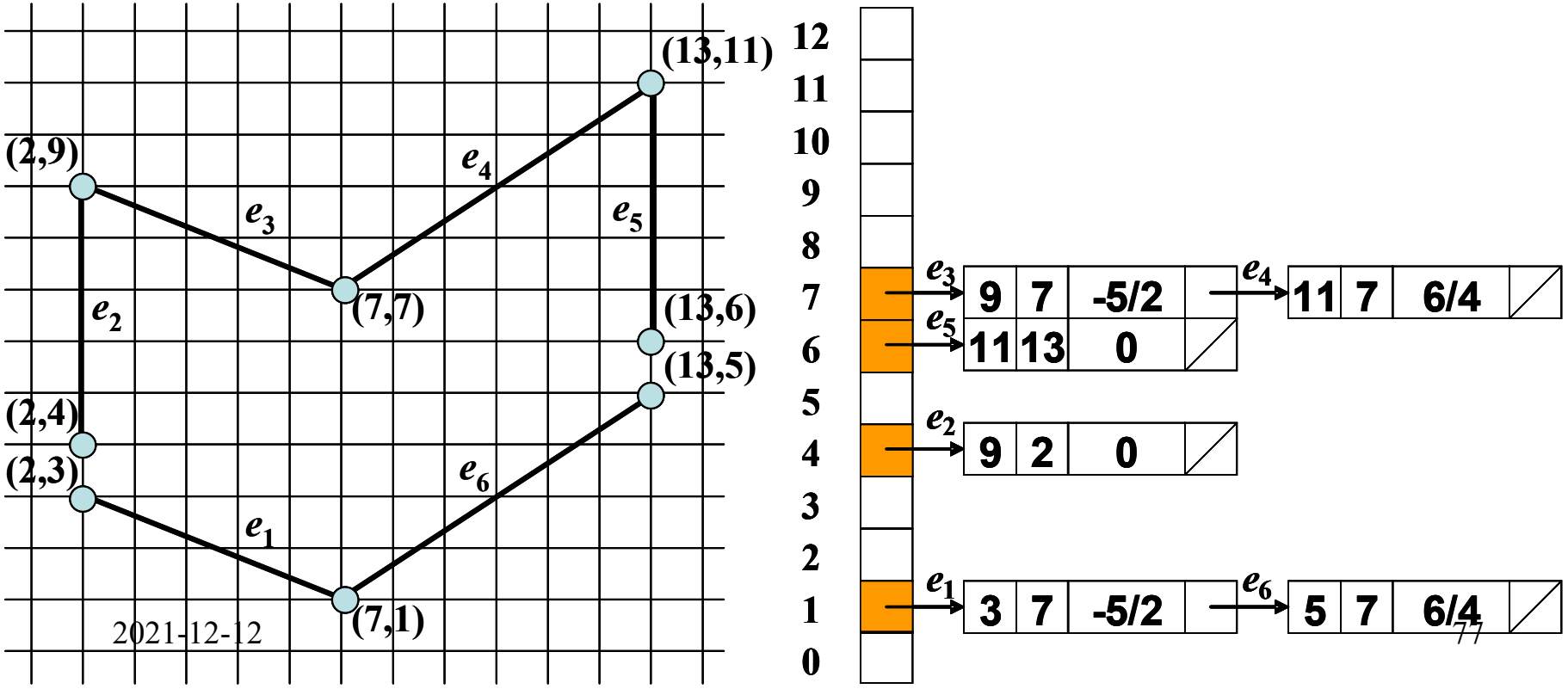
1/m:当前边的斜率的倒数

next:指向下一条边的指针

每次离开一条扫描线进入下一条之前，将表中有但与下一条扫描线不相交的边清除出表，将与下一条扫描线相交而表中没有的边加入表中。

1. 边的上端点y坐标ymax。
2. 边的下端点的x坐标xmin。
3. 斜率的倒数，即1/m。

y_{max}	x_{min}	1/m	next



Polygonfill(EdgeTableET,COLORREFcolor)

{

1.y=置y为边表ET中各登记项对应的y 坐标中最小的值;

2.活跃边表AET初始化为空表;

3.若AET非空或ET非空，则做下列步骤，否则算法结束

{

3.1.将ET中登记项y对应的各“吊桶”合并到表AET中，将AET中各吊桶按x坐标递增排序;

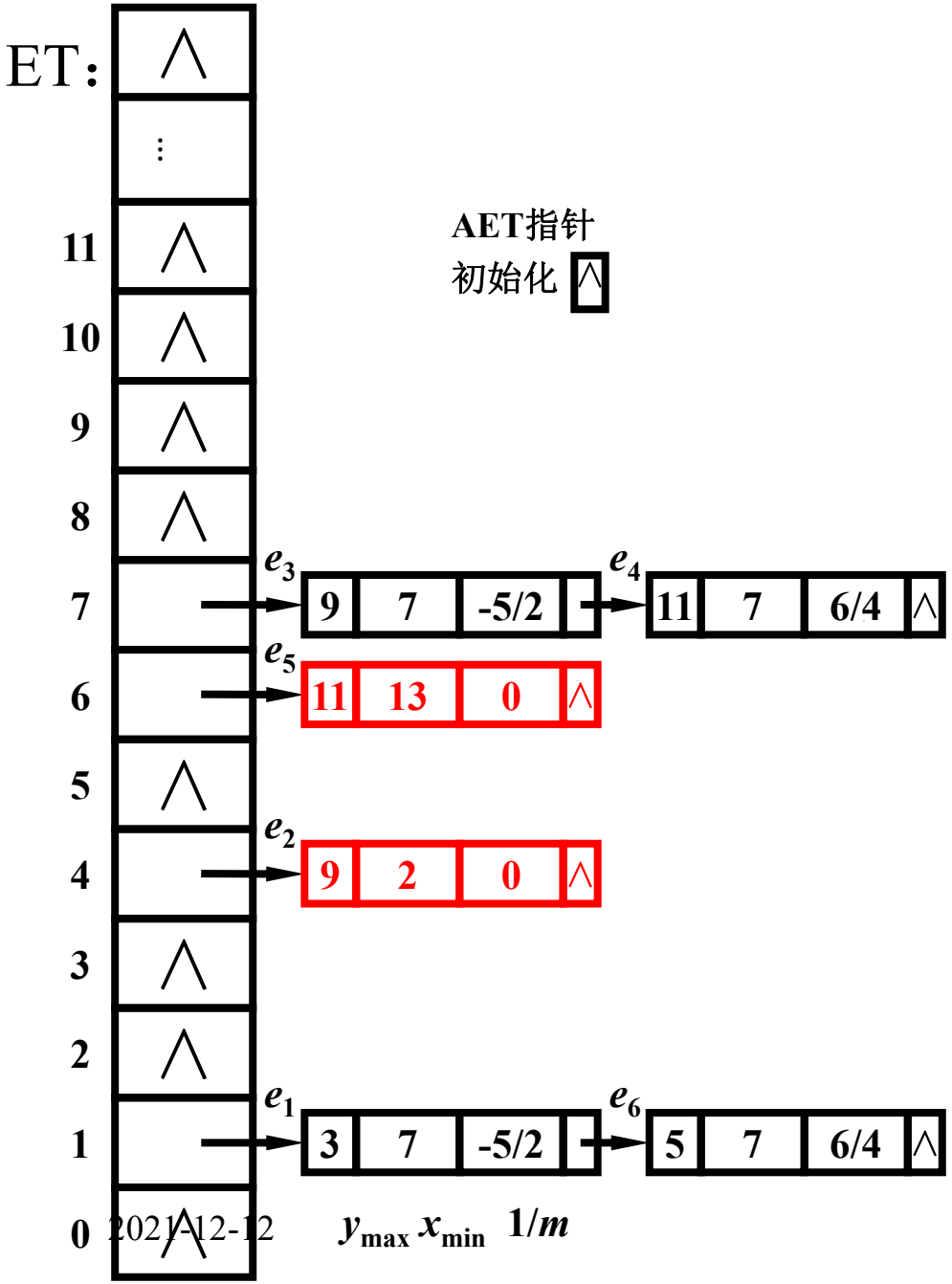
3.2.在扫描线y上，按照AET表提供的x坐标对, 用 color 实施填充;

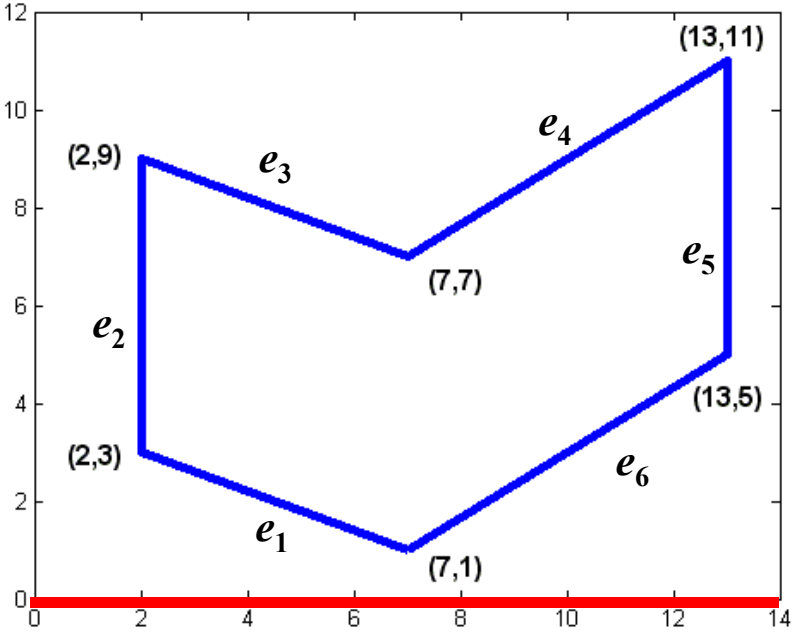
3.3.将AET表中有$y=y_{max}$的各项清除出表;

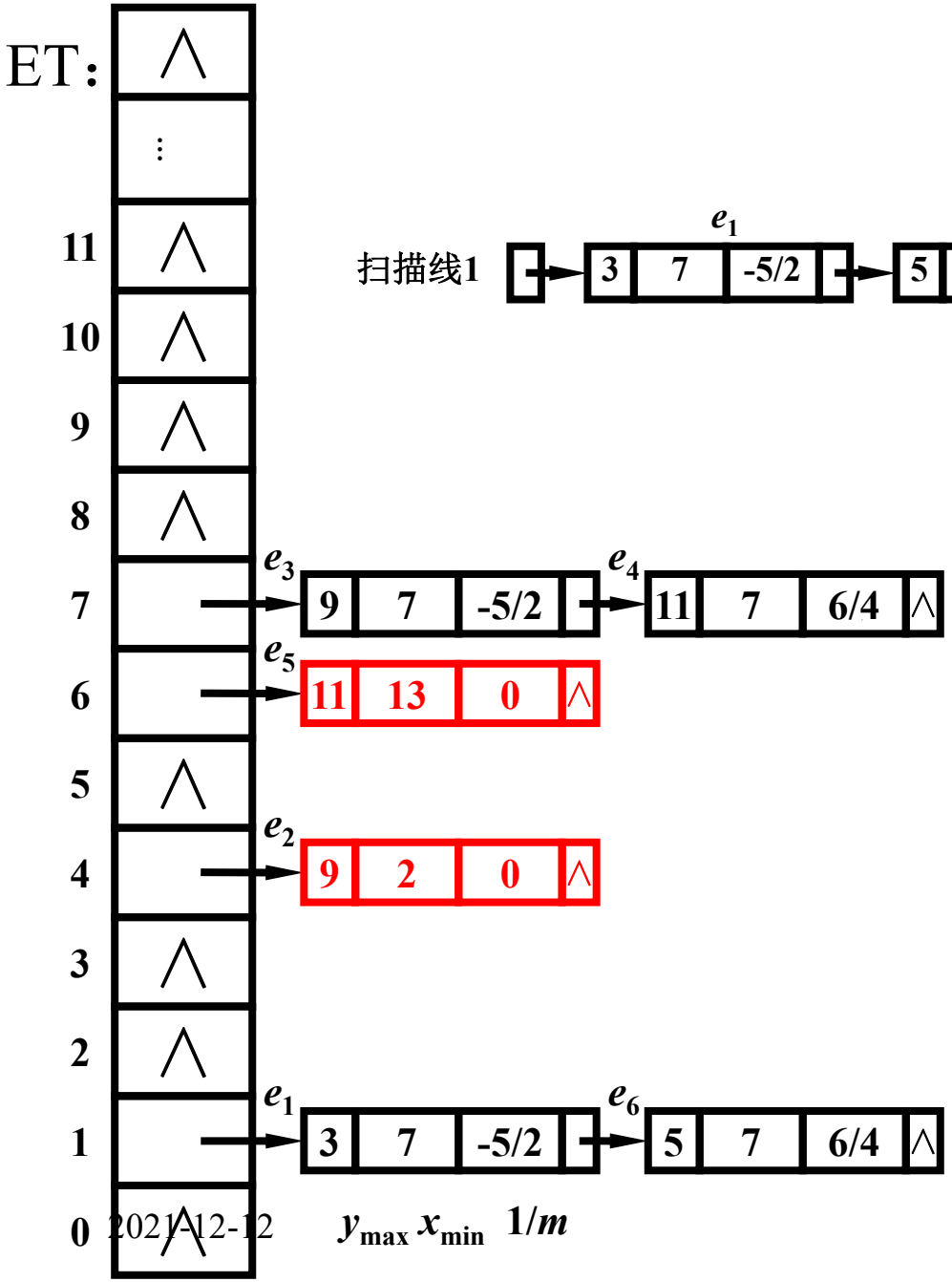
3.4.对AET中留下的各项，分别将x换为x+1/m，求出AET中各边与下一条扫描线交点的x坐标;

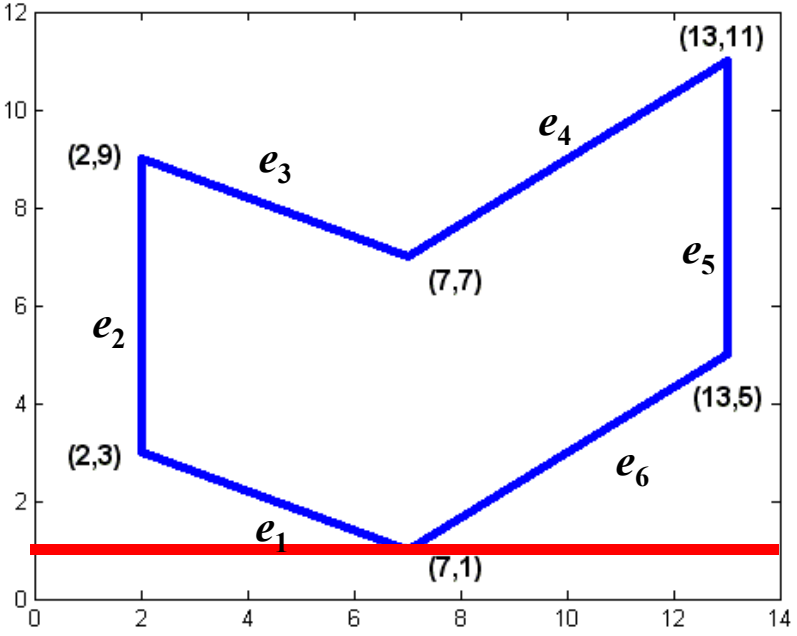
3.5.y=y＋1，返回步骤3处理下一条扫描线。

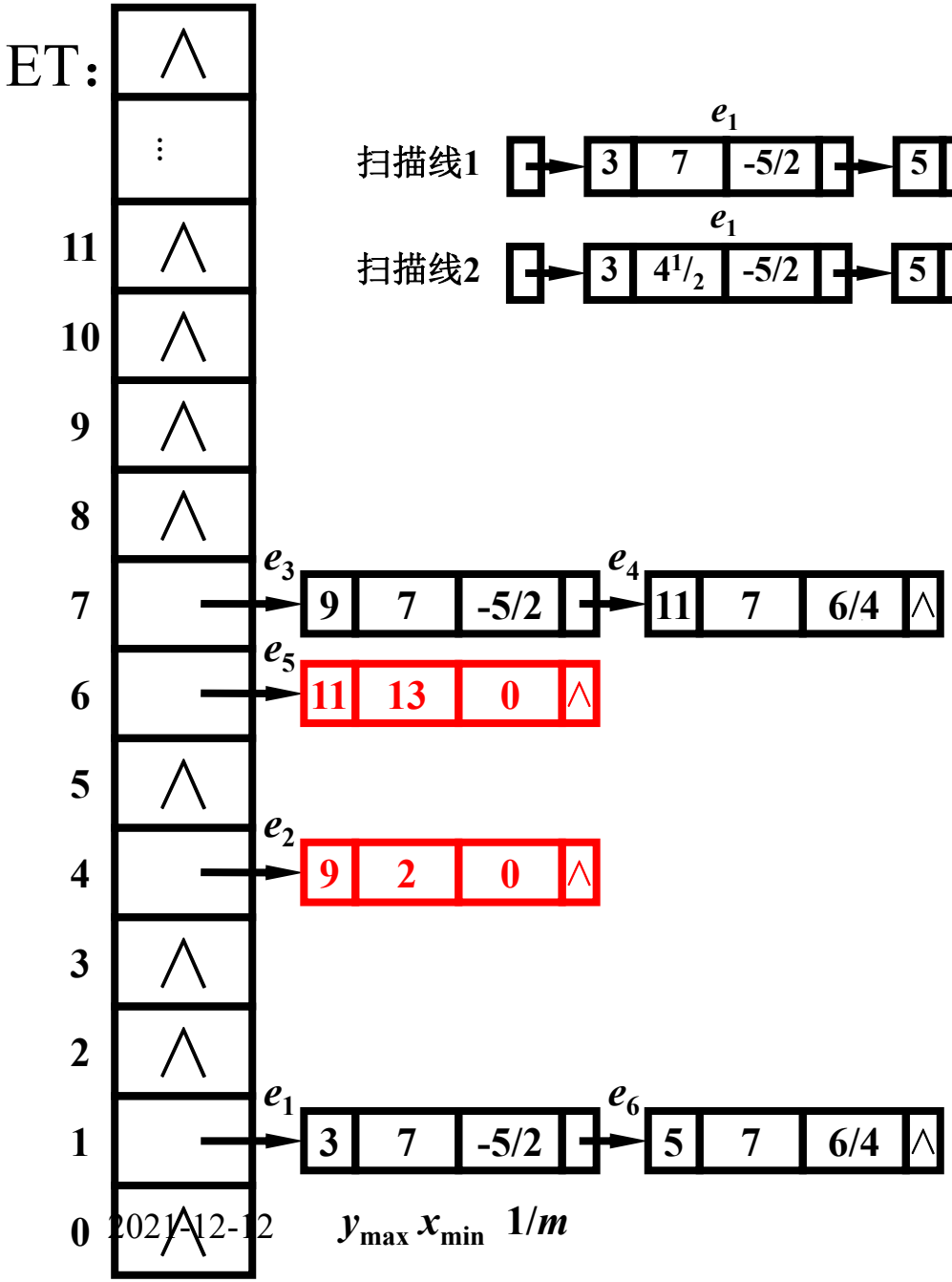
}}


ET:
11
10
9
8
7
6
5
4
3
2
1
0
AET指针
初始化
e_3
9 7 -5/2
e_4
11 7 6/4
e_5
11 13 0
e_2
9 2 0
e_1
3 7 -5/2
e_6
5 7 6/4
y_{max} x_{min} $1/m$

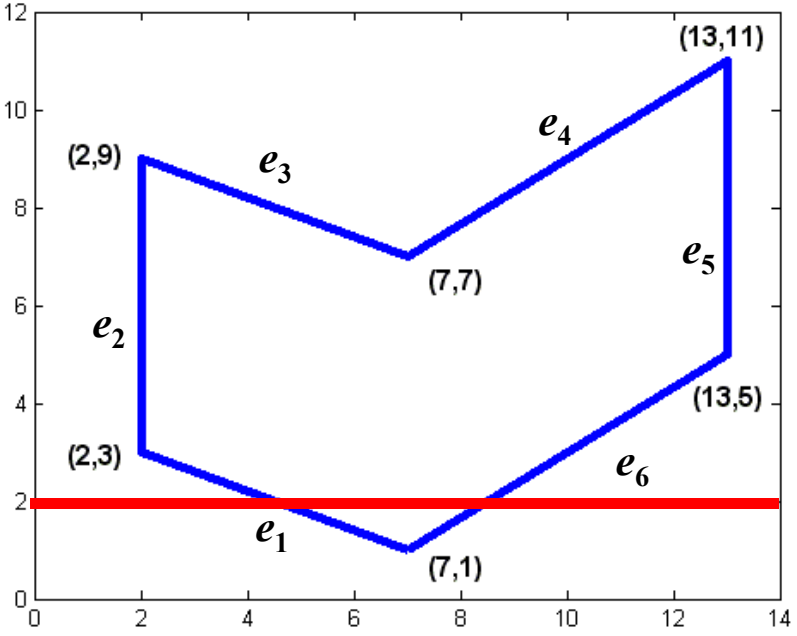

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0
2
4
6
8
10
12
14

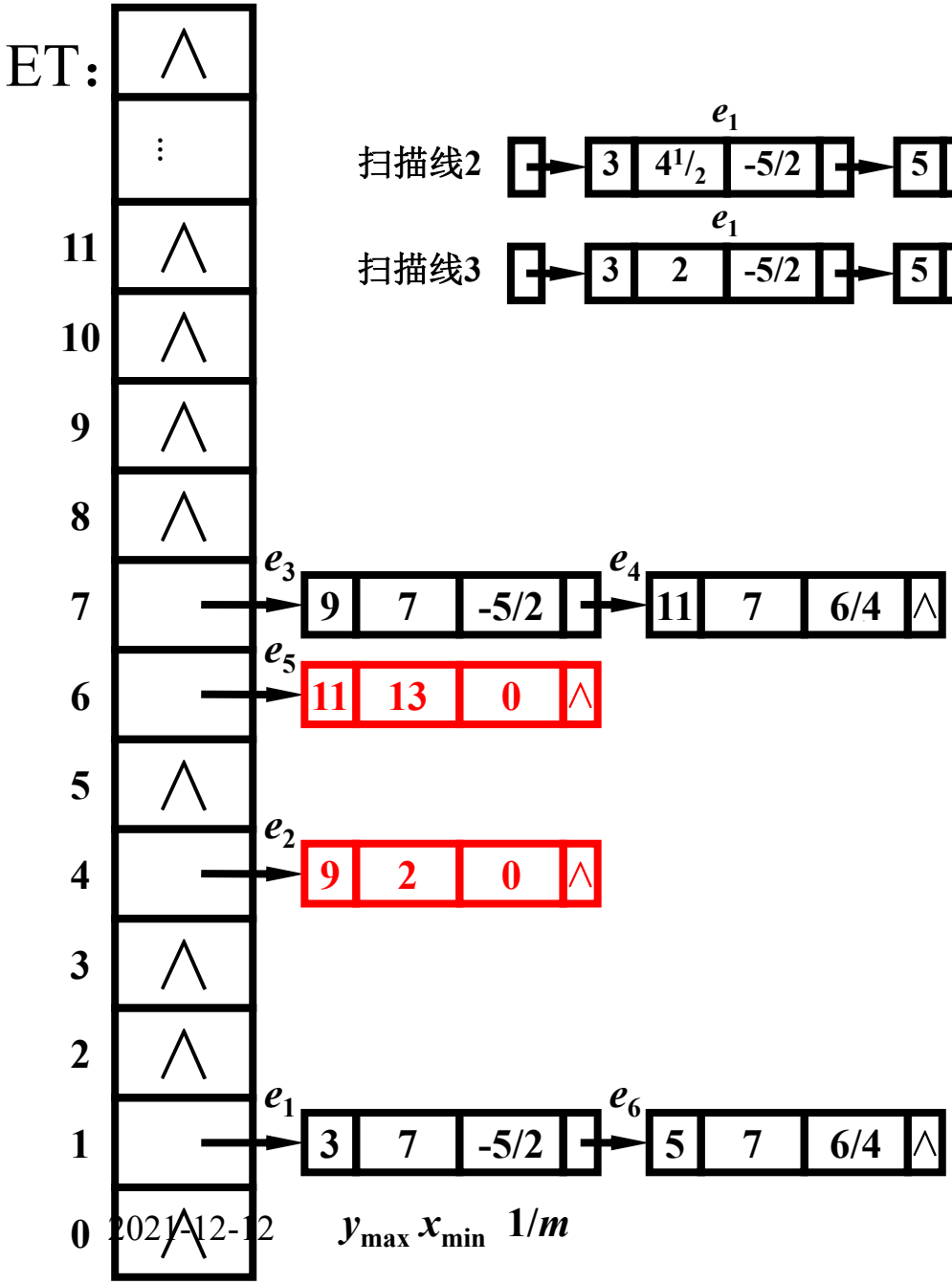

ET:
11
10
9
8
7
6
5
4
3
2
1
0
扫描线1
e_1
e_6
3 7 -5/2
5 7 6/4
e_3
e_4
9 7 -5/2
11 7 6/4
e_5
11 13 0
e_2
9 2 0
e_1
e_6
3 7 -5/2
5 7 6/4
y_{max} x_{min} $1/m$

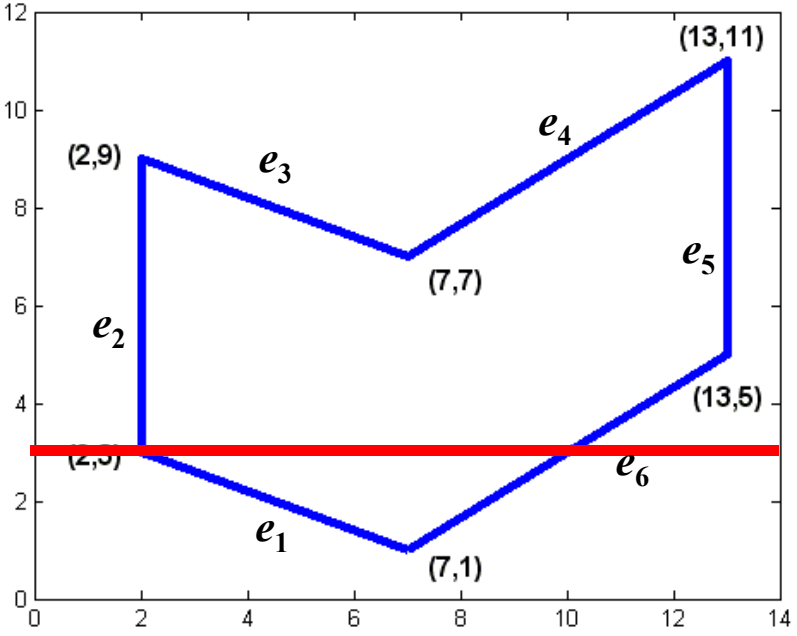

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0
2
4
6
8
10
12
14

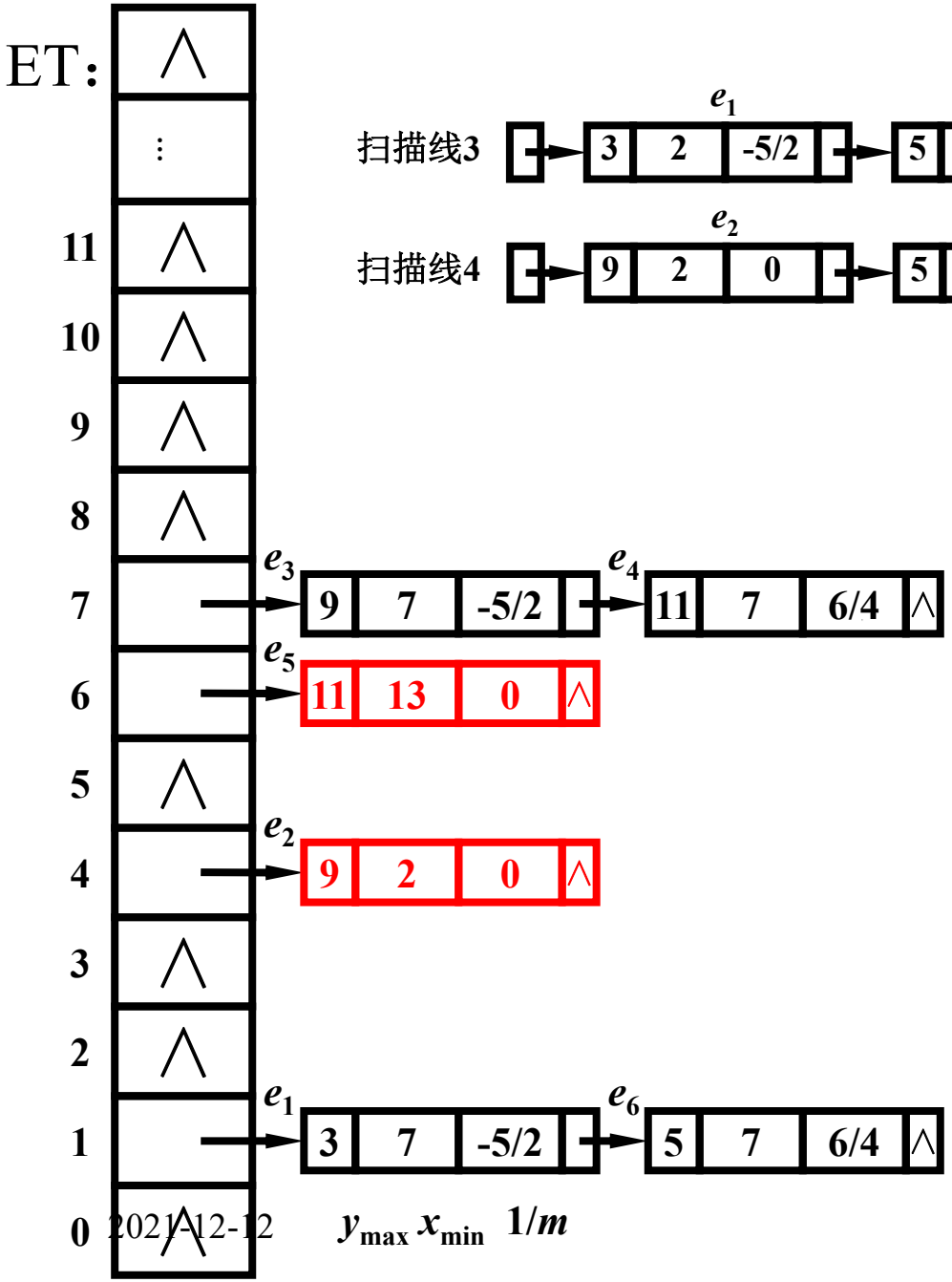

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2
e_4 11 7 6/4
e_5 11 13 0
e_2 9 2 0
e_1 3 7 -5/2
e_6 5 7 6/4
y_{max} x_{min} $1/m$

扫描线1
e_1 3 7 -5/2
e_6 5 7 6/4
扫描线2
e_1 3 $4^1/_2$ -5/2
e_6 5 $8^1/_2$ 6/4

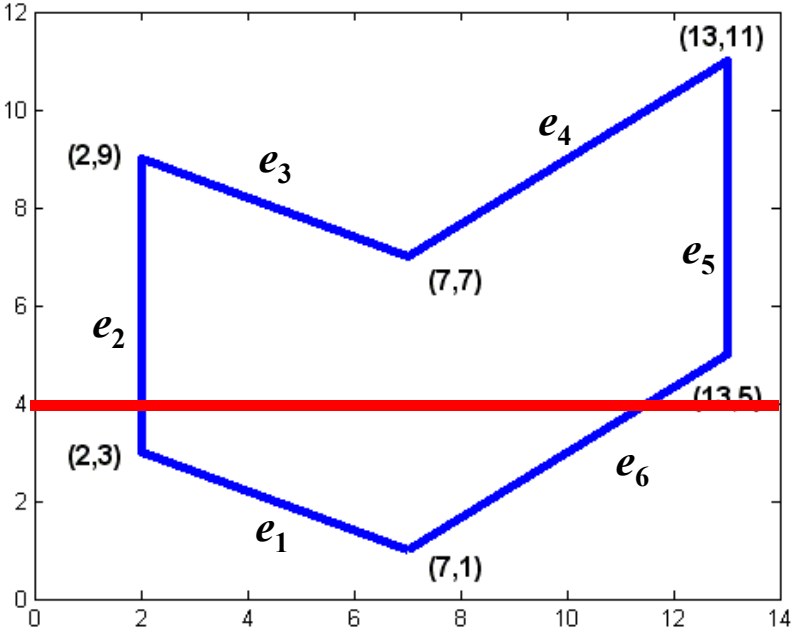

(13,11)
(2,9)
(7,7)
(13,5)
(2,3)
(7,1)
e_1
e_2
e_3
e_4
e_5
e_6

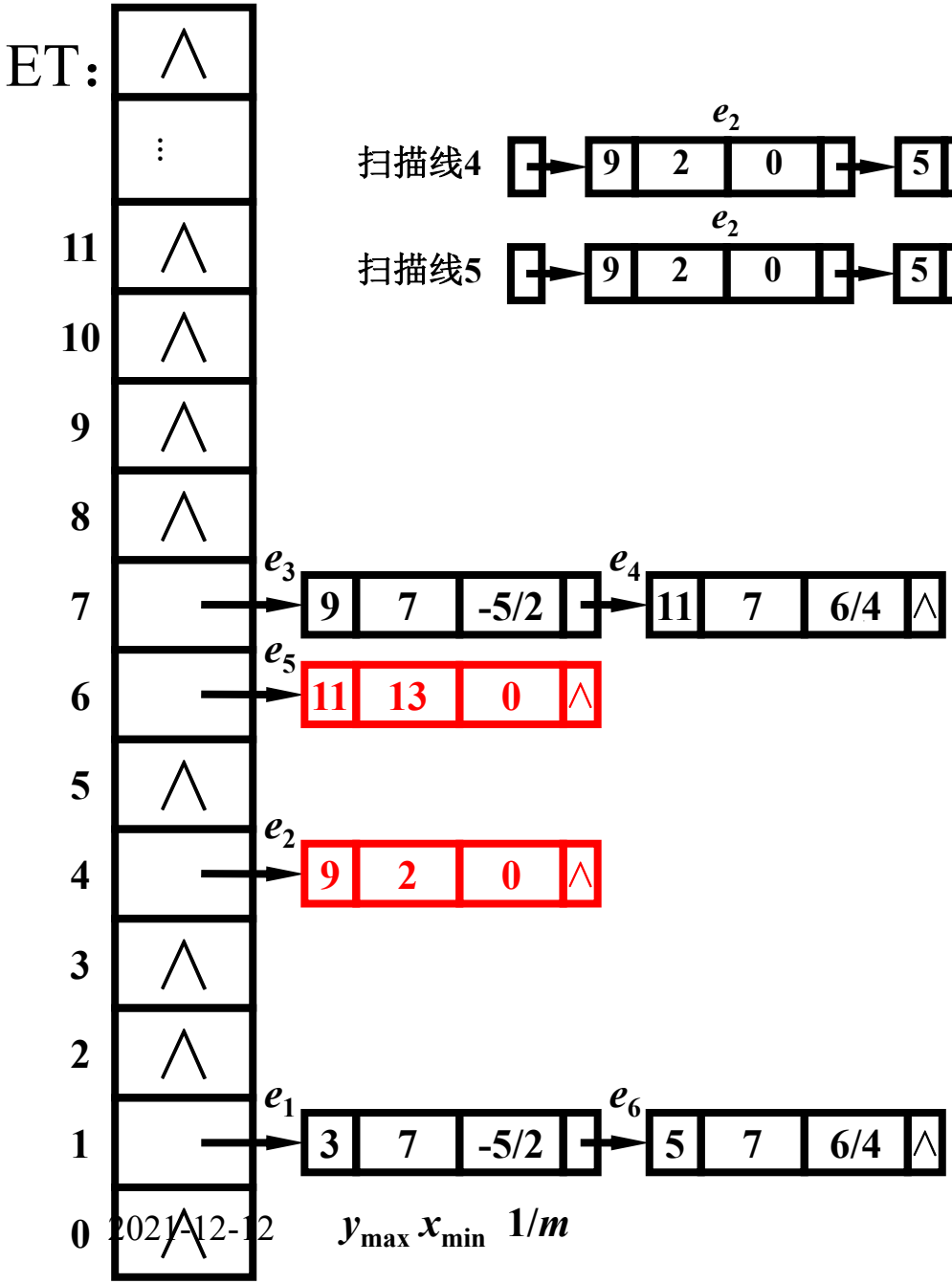

ET:
11
10
9
8
7
6
5
4
3
2
1
0
扫描线2
扫描线3
e_1
e_6
3 $4^1/_2$ -5/2
5 $8^1/_2$ 6/4
3 2 -5/2
5 10 6/4
e_3
e_4
9 7 -5/2
11 7 6/4
e_5
11 13 0
e_2
9 2 0
3 7 -5/2
5 7 6/4
y_{max} x_{min} $1/m$


(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)

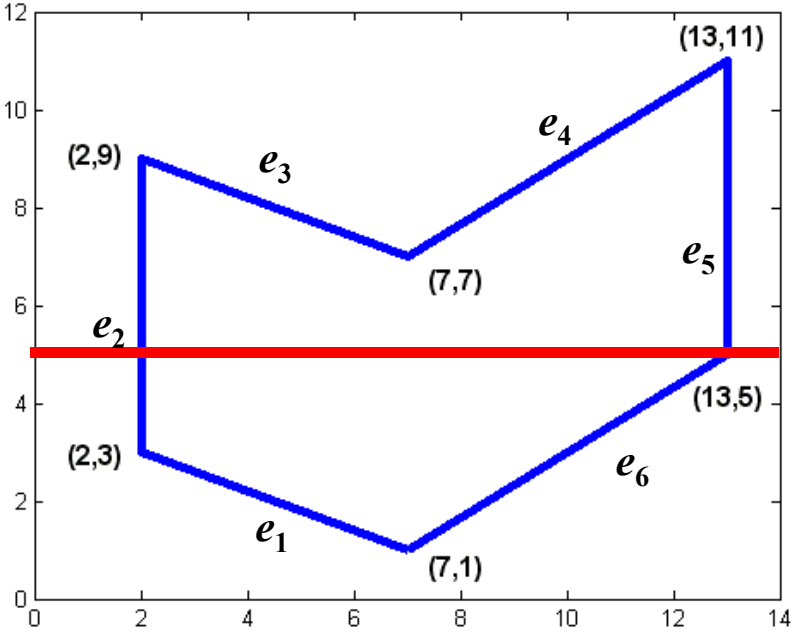

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2
e_4 11 7 6/4
e_5 11 13 0
e_2 9 2 0
e_1 3 7 -5/2
e_6 5 7 6/4
y_{max} x_{min} $1/m$

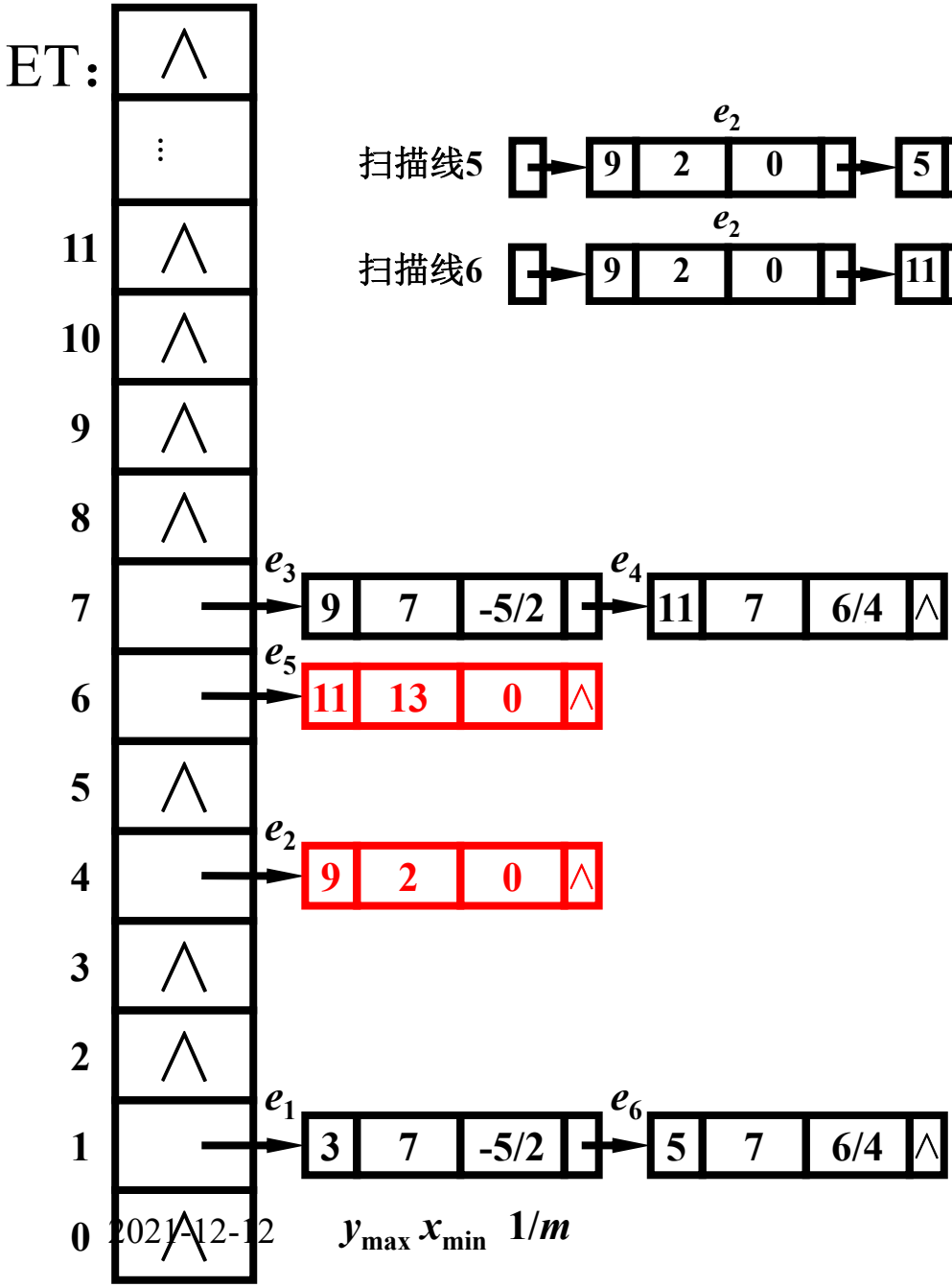
扫描线3
e_1 3 2 -5/2
e_6 5 10 6/4
扫描线4
e_2 9 2 0
e_6 5 $11^1/_2$ 6/4


(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0
2
4
6
8
10
12
14

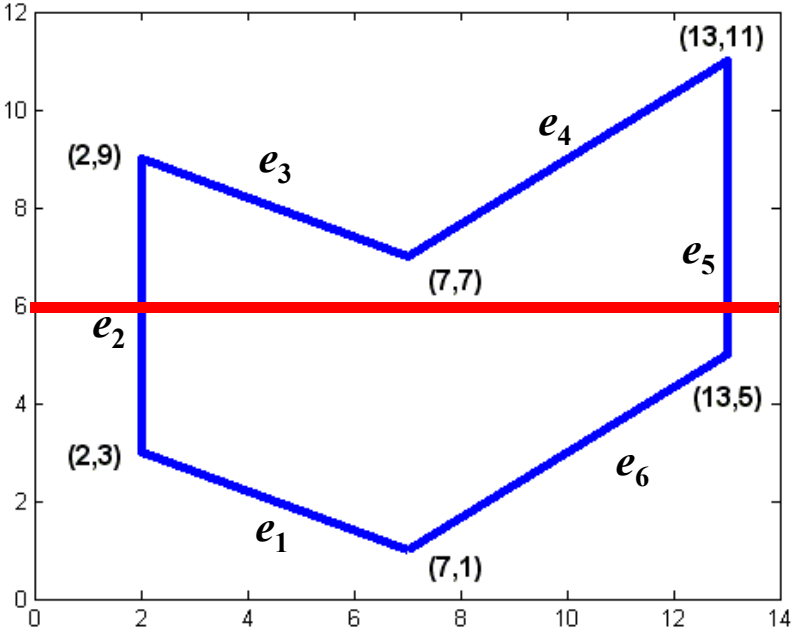

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2
e_4 11 7 6/4
e_5 11 13 0
e_2 9 2 0
e_1 3 7 -5/2
e_6 5 7 6/4
y_{max} x_{min} $1/m$

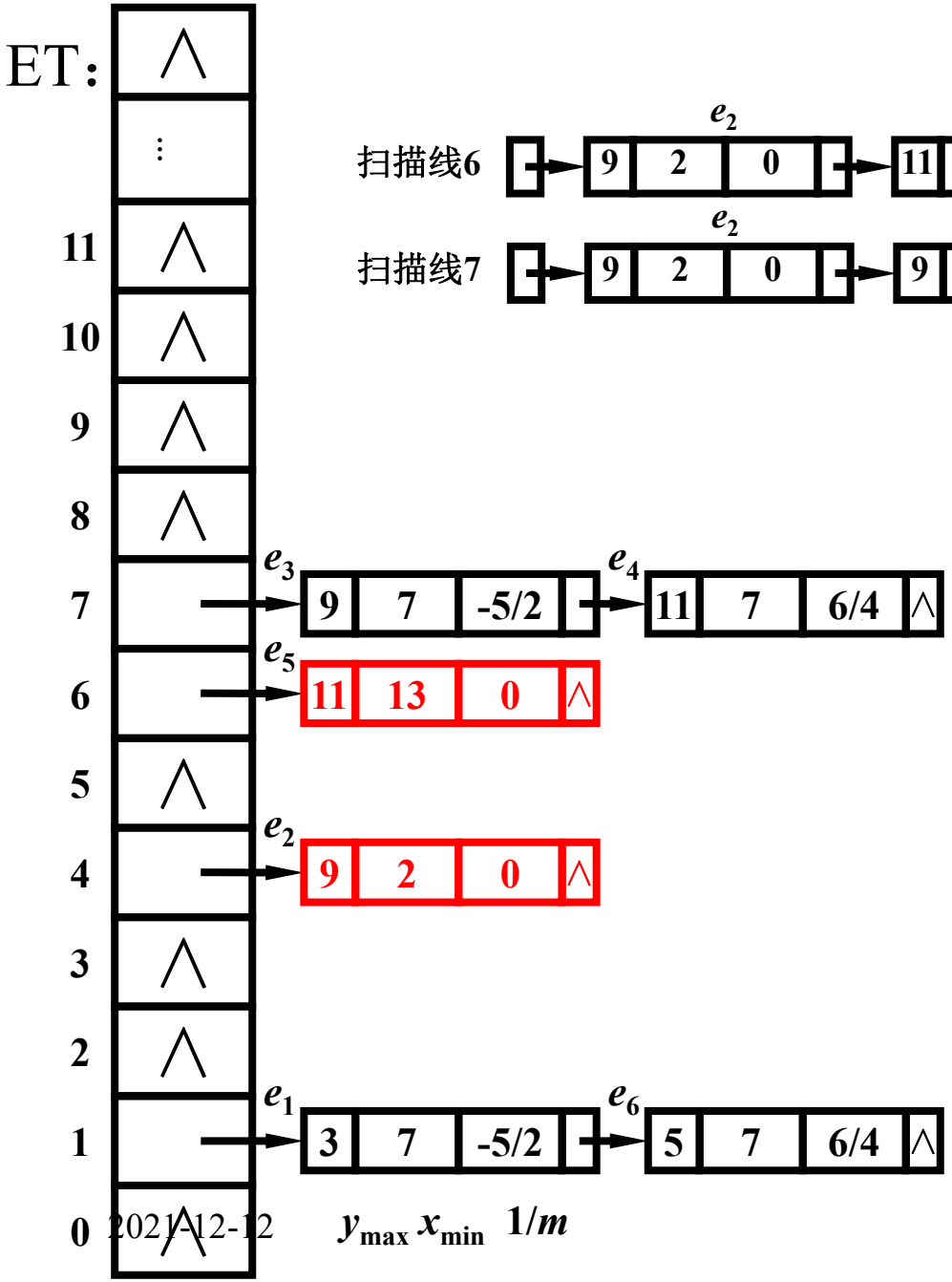
扫描线4
e_2 9 2 0
e_6 5 $11^1/_2$ 6/4
扫描线5
e_2 9 2 0
e_6 5 13 6/4

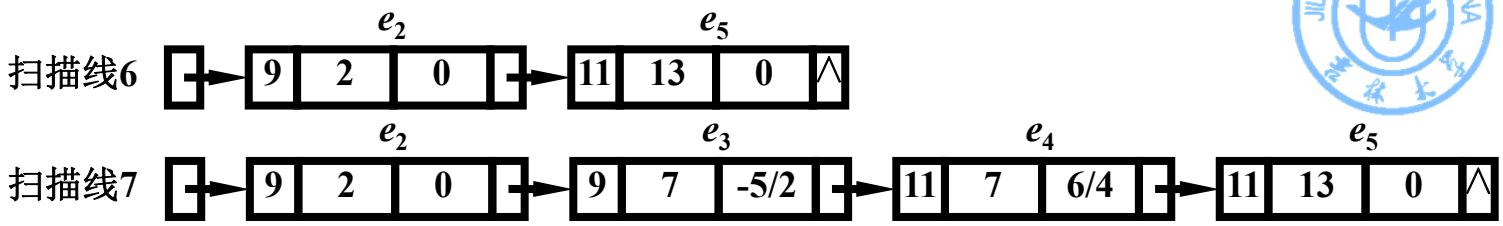

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)

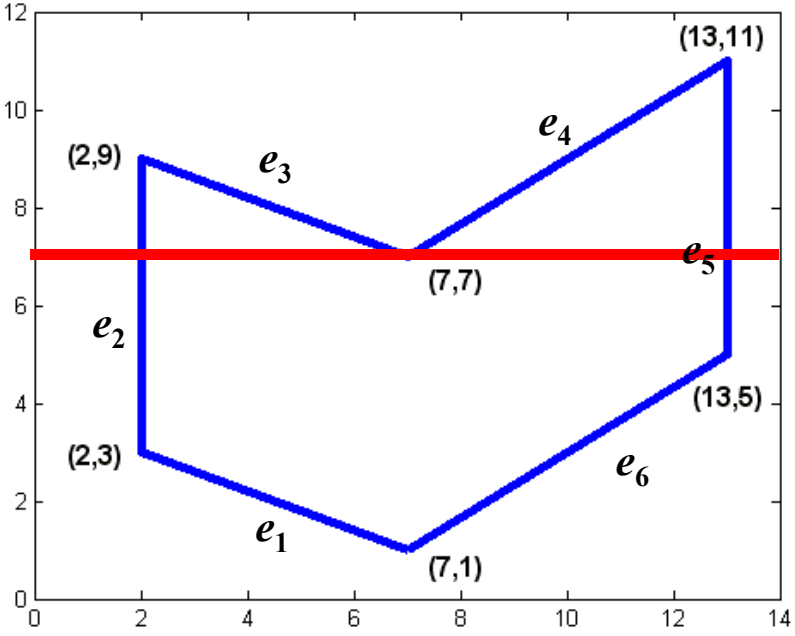

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2 e_4 11 7 6/4 ∧
e_5 11 13 0 ∧
e_2 9 2 0 ∧
e_1 3 7 -5/2 e_6 5 7 6/4 ∧
y_{max} x_{min} $1/m$

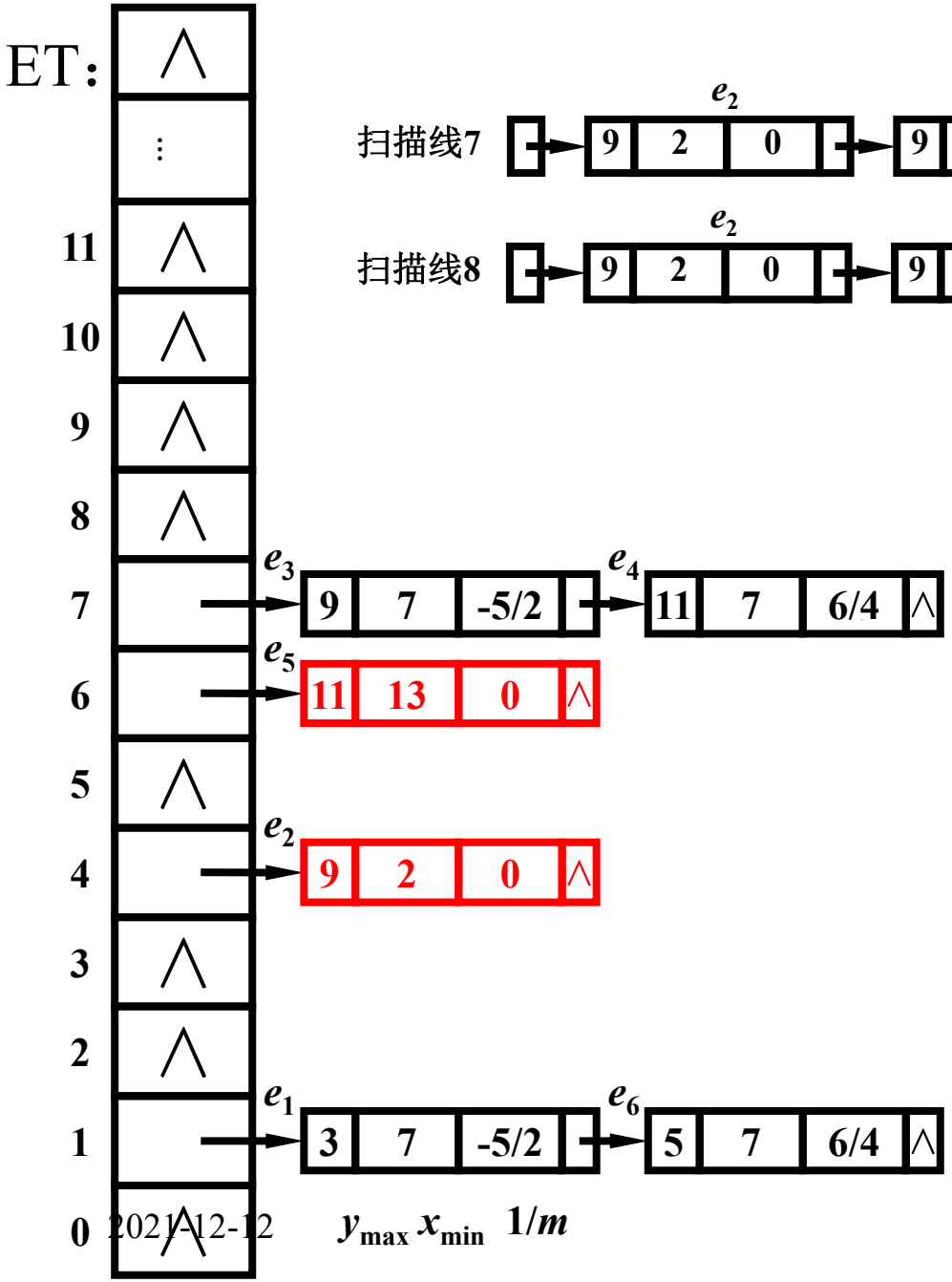
扫描线5
e_2 9 2 0 e_6 5 13 6/4 ∧
扫描线6
e_2 9 2 0 e_5 11 13 0 ∧

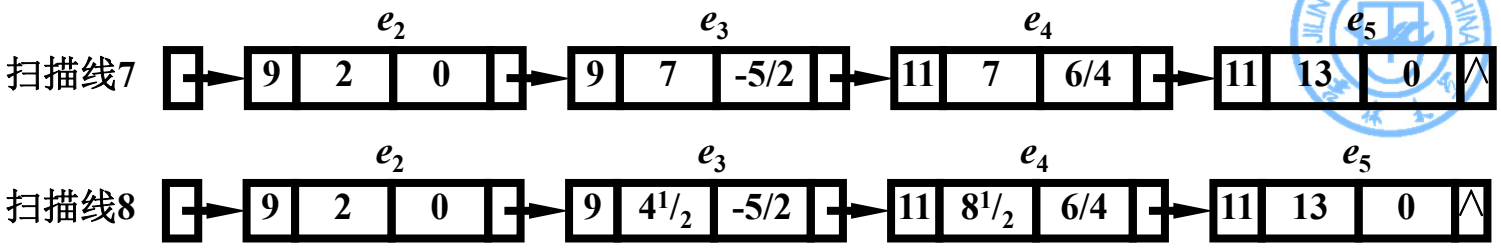

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0 2 4 6 8 10 12 14
0 2 4 6 8 10 12

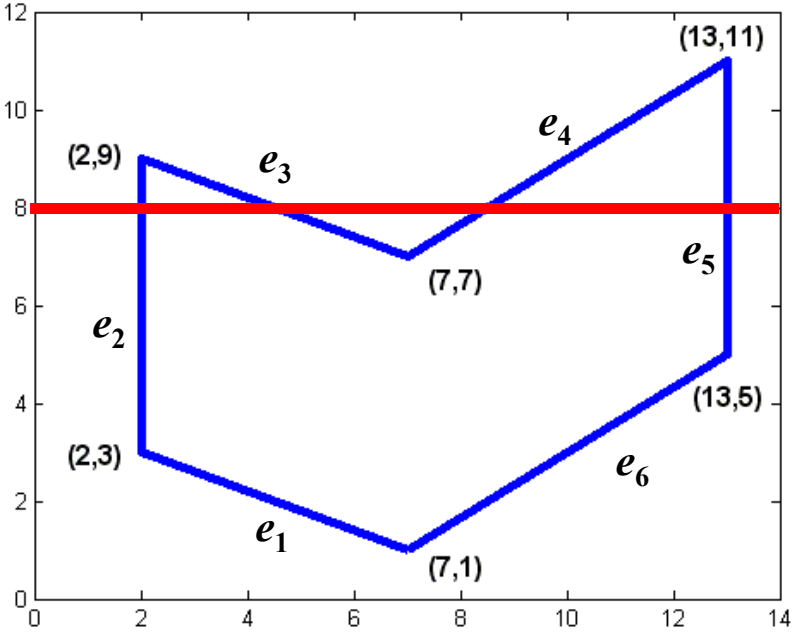

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2 e_4 11 7 6/4 ∧
e_5 11 13 0 ∧
e_2 9 2 0 ∧
e_1 3 7 -5/2 e_6 5 7 6/4 ∧
y_{max} x_{min} $1/m$

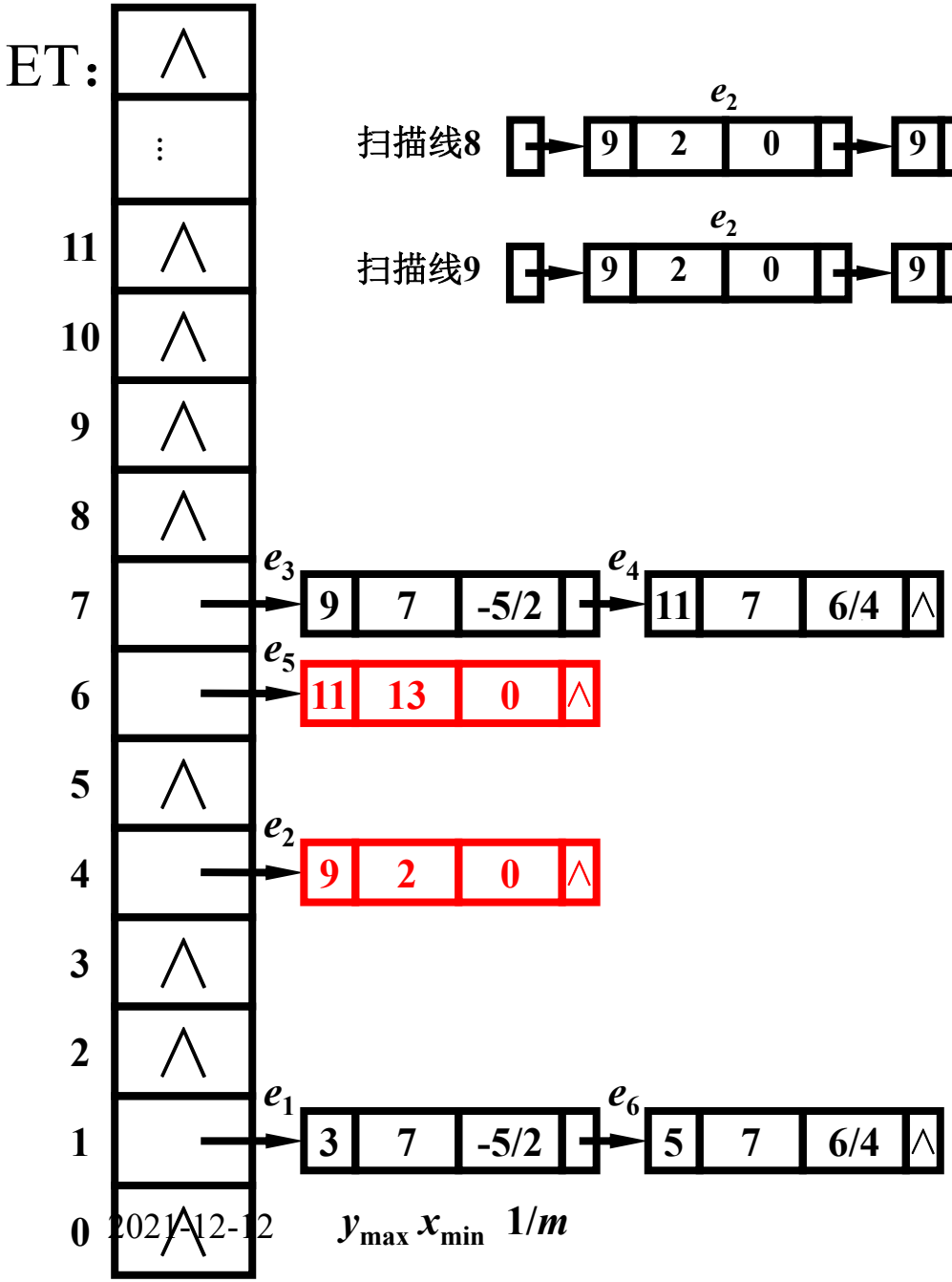

扫描线6
e_2 9 2 0 e_5 11 13 0 ∧
扫描线7
e_2 9 2 0 e_3 9 7 -5/2 e_4 11 7 6/4 e_5 11 13 0 ∧

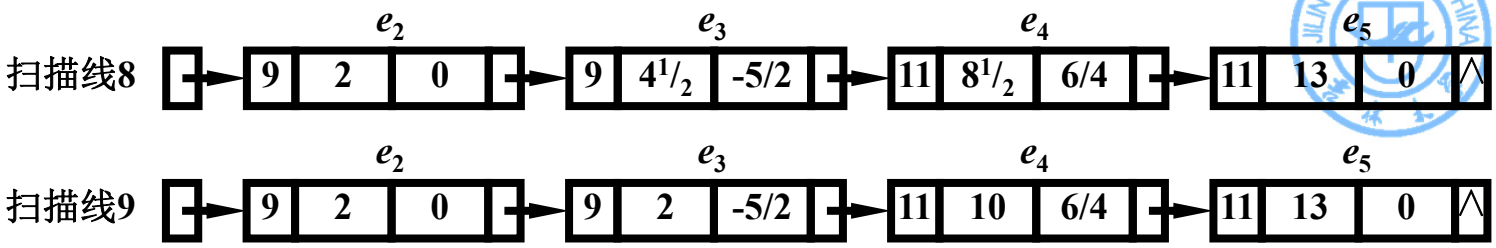

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)

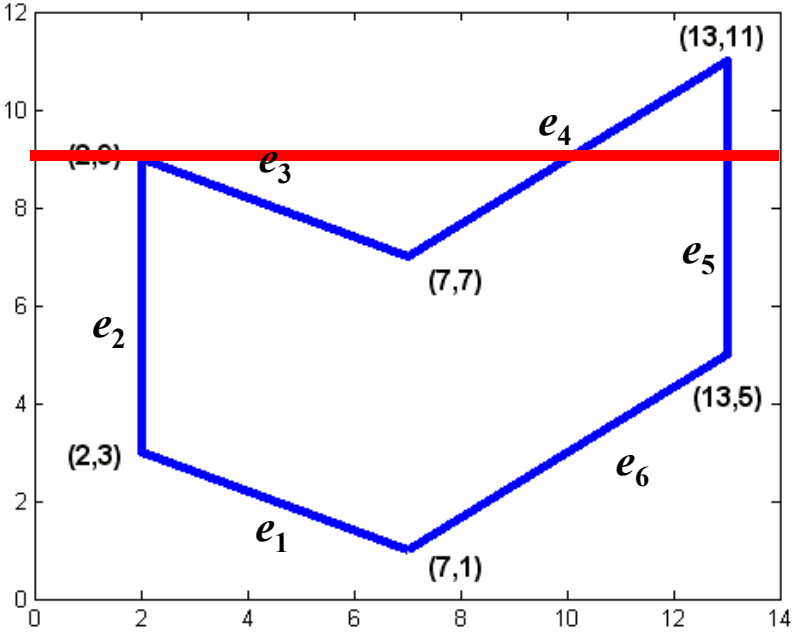

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2 e_4 11 7 6/4 ∧
e_5 11 13 0 ∧
e_2 9 2 0 ∧
e_1 3 7 -5/2 e_6 5 7 6/4 ∧
y_{max} x_{min} $1/m$

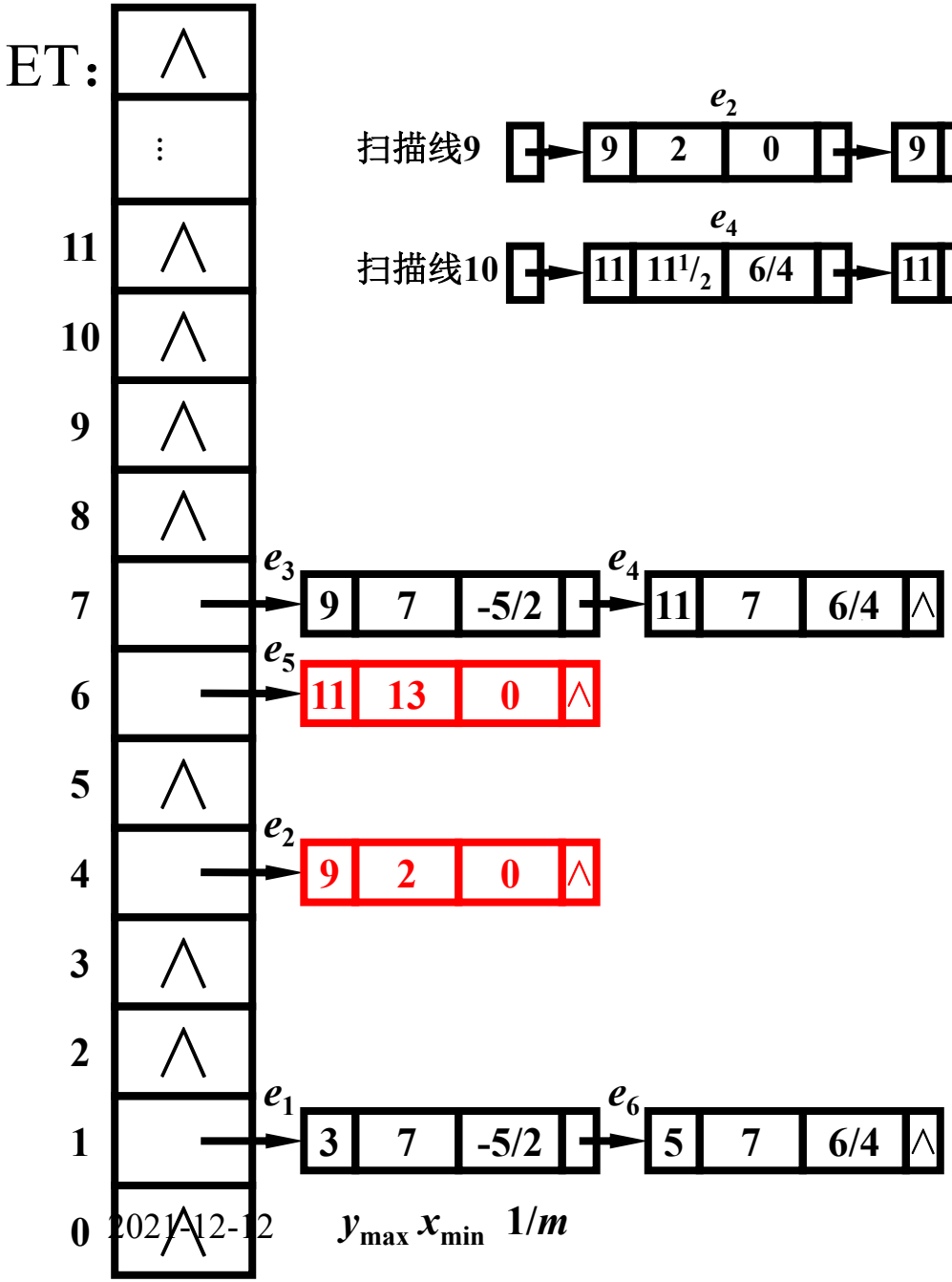

扫描线7
e_2 9 2 0
e_3 9 7 -5/2
e_4 11 7 6/4
e_5 11 13 0 ∧
扫描线8
e_2 9 2 0
e_3 9 $4^1/_2$ -5/2
e_4 11 $8^1/_2$ 6/4
e_5 11 13 0 ∧

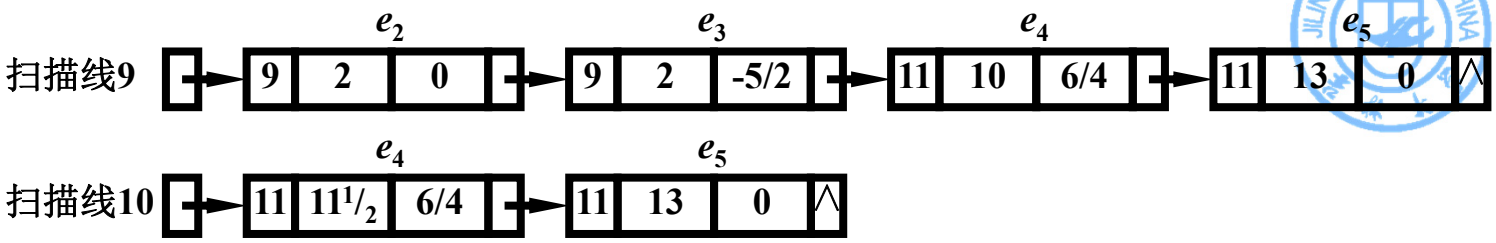

(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)

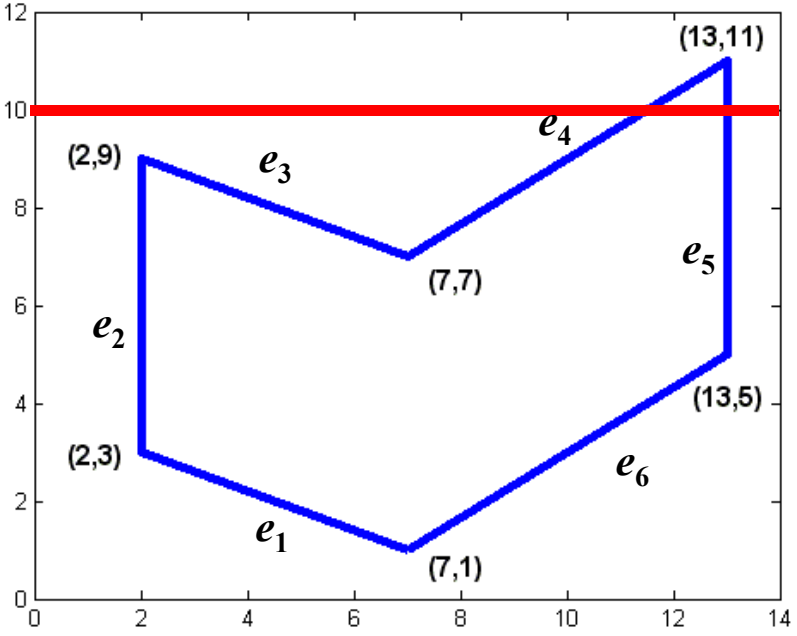

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2 e_4 11 7 6/4 ∧
e_5 11 13 0 ∧
e_2 9 2 0 ∧
e_1 3 7 -5/2 e_6 5 7 6/4 ∧
y_{max} x_{min} $1/m$

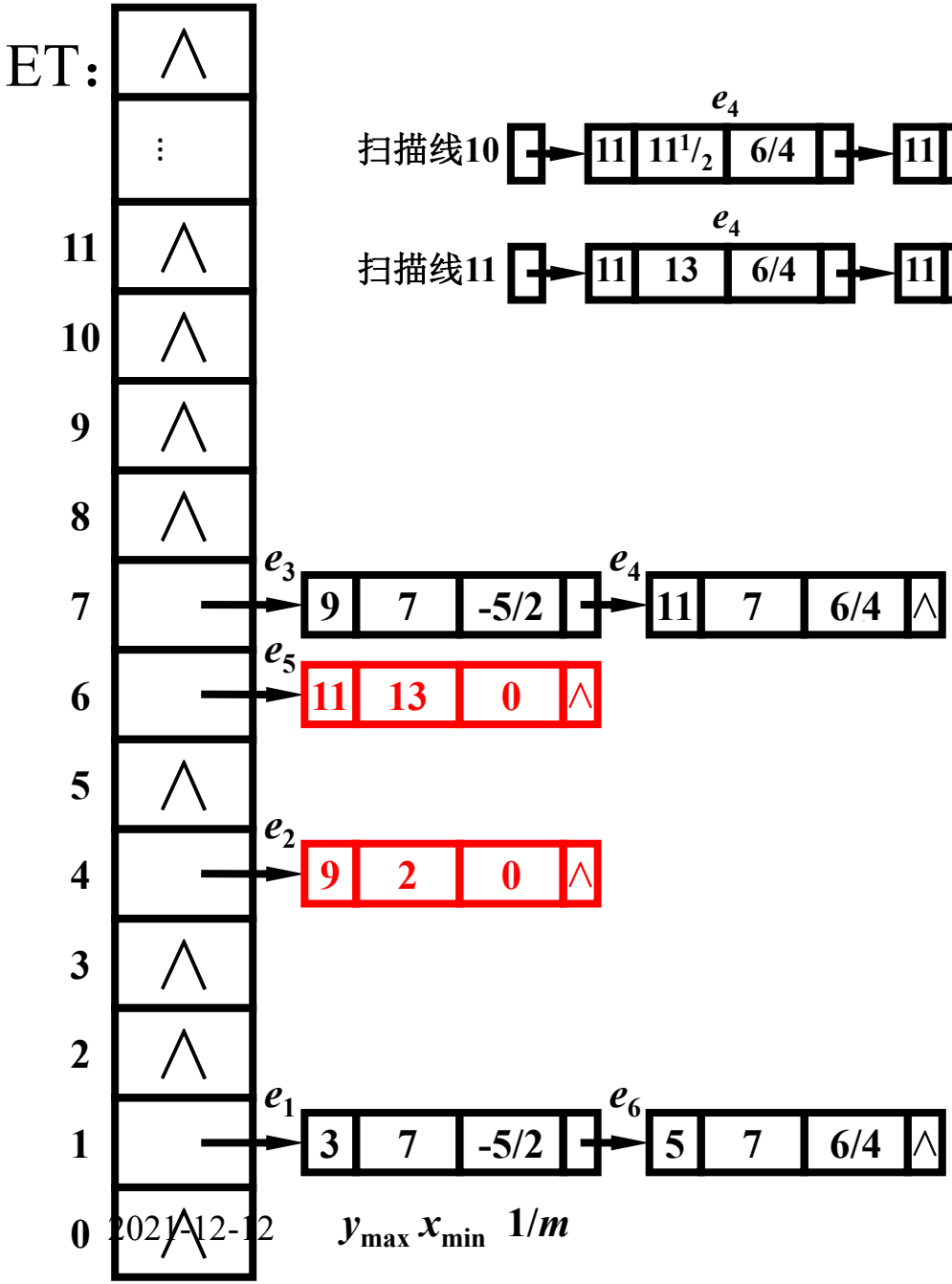

扫描线8
e_2 9 2 0
e_3 9 $4^1/_2$ -5/2
e_4 11 $8^1/_2$ 6/4
e_5 11 13 0 ∧
扫描线9
e_2 9 2 0
e_3 9 2 -5/2
e_4 11 10 6/4
e_5 11 13 0 ∧


(13,11)
e_4
(2,9)
e_3
(7,7)
e_5
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0
2
4
6
8
10
12
14

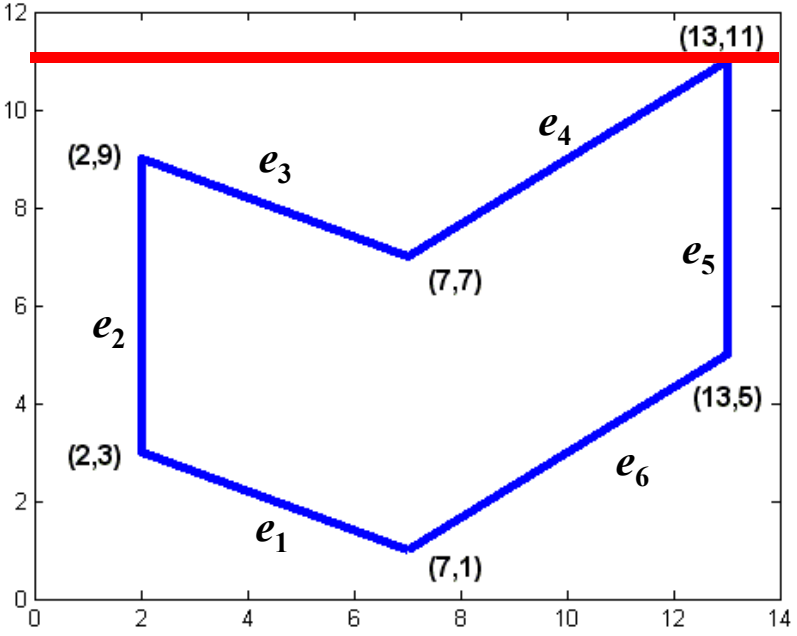

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2 e_4 11 7 6/4 ∧
e_5 11 13 0 ∧
e_2 9 2 0 ∧
e_1 3 7 -5/2 e_6 5 7 6/4 ∧
y_{max} x_{min} $1/m$

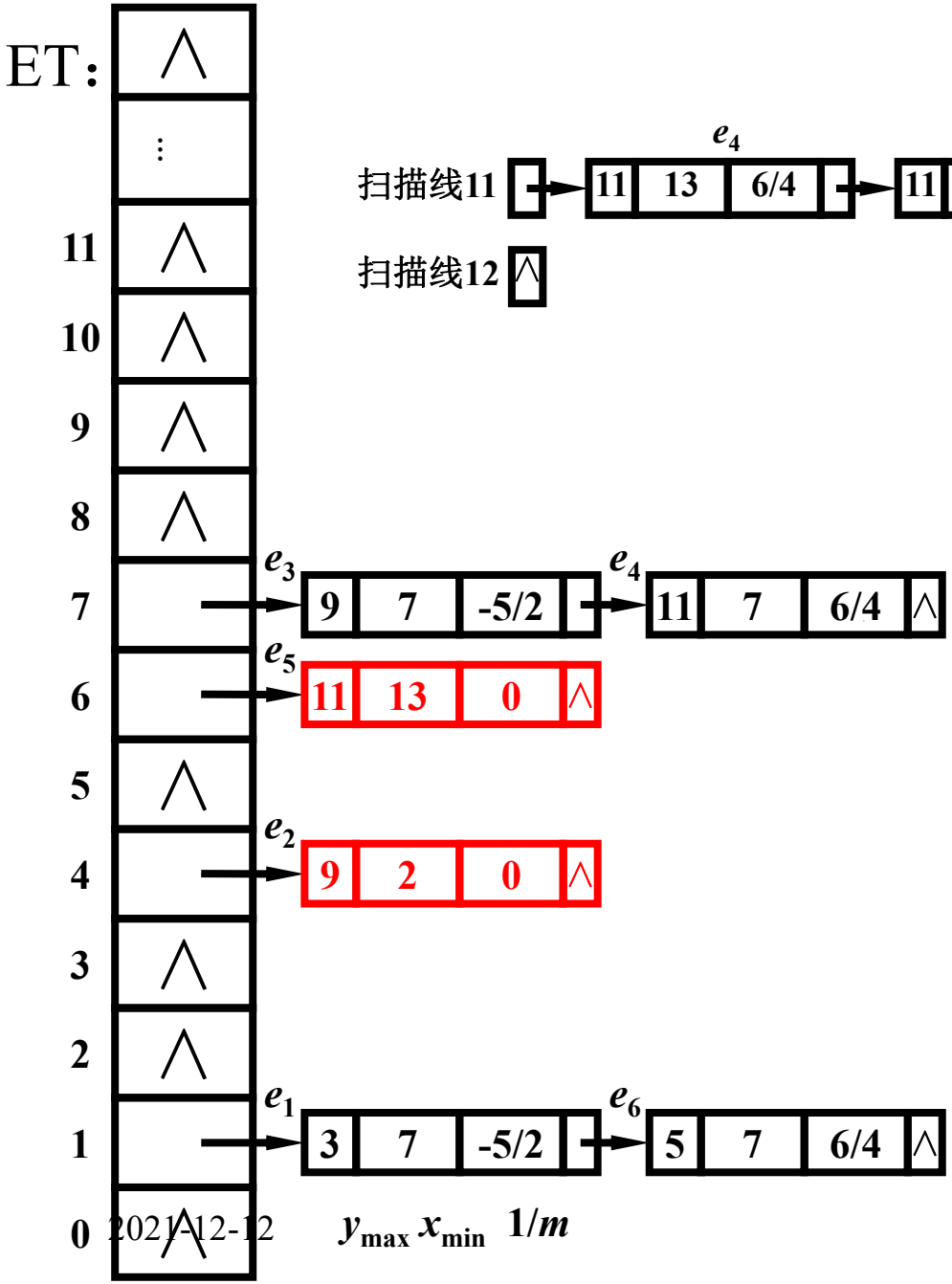

扫描线9
e_2 9 2 0
e_3 9 2 -5/2
e_4 11 10 6/4
e_5 11 13 0 ∧
扫描线10
e_4 11 $11^1/_2$ 6/4
e_5 11 13 0 ∧

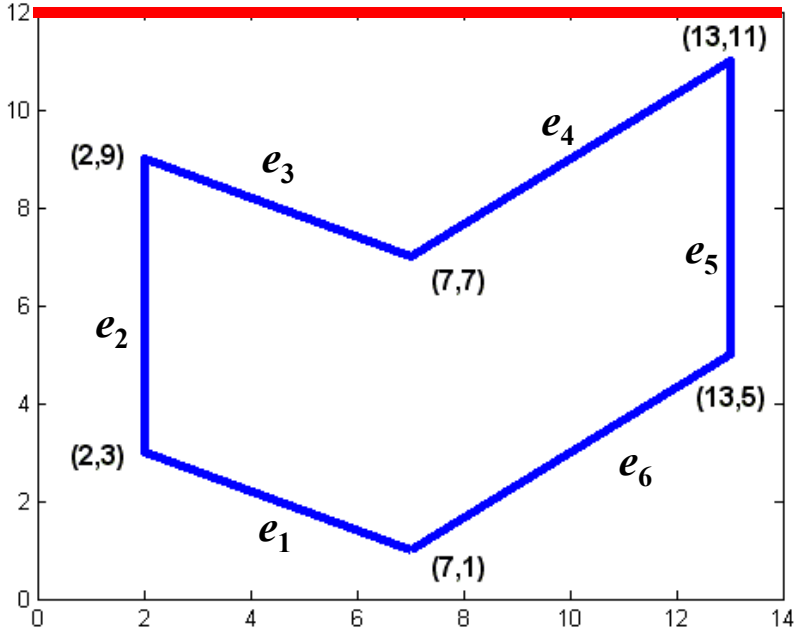

(13,11)
(2,9)
(7,7)
(13,5)
(2,3)
(7,1)
e_1
e_2
e_3
e_4
e_5
e_6
0
2
4
6
8
10
12
14

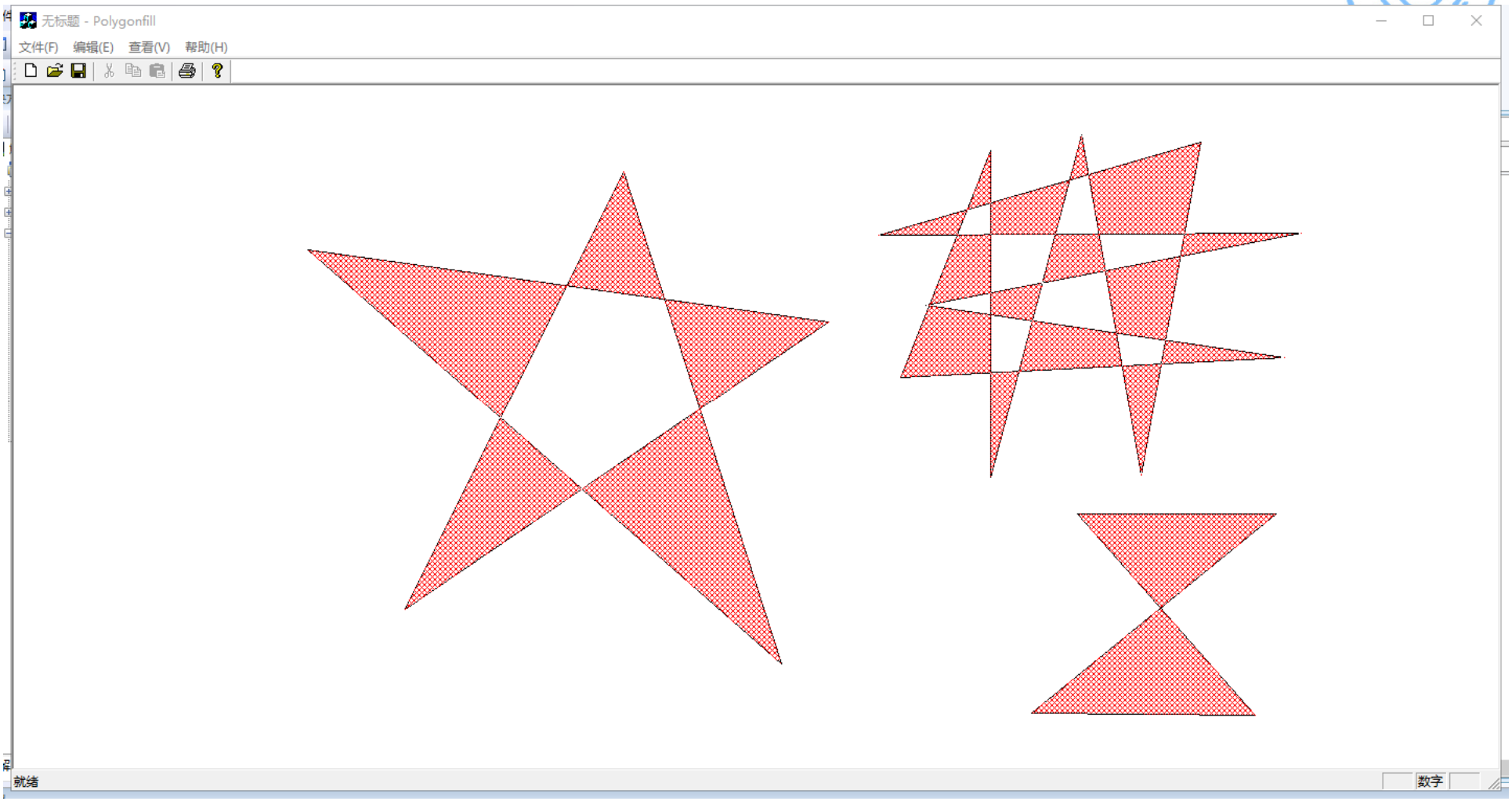

ET:
11
10
9
8
7
6
5
4
3
2
1
0
e_3 9 7 -5/2
e_4 11 7 6/4
e_5 11 13 0
e_2 9 2 0
e_1 3 7 -5/2
e_6 5 7 6/4
y_{max} x_{min} $1/m$

扫描线10
e_4 11 $11^1/_2$ 6/4
e_5 11 13 0
扫描线11
e_4 11 13 6/4
e_5 11 13 0


(13,11)
(2,9)
e_3
e_4
e_5
(7,7)
e_2
(13,5)
(2,3)
e_6
e_1
(7,1)
0
2
4
6
8
10
12
14

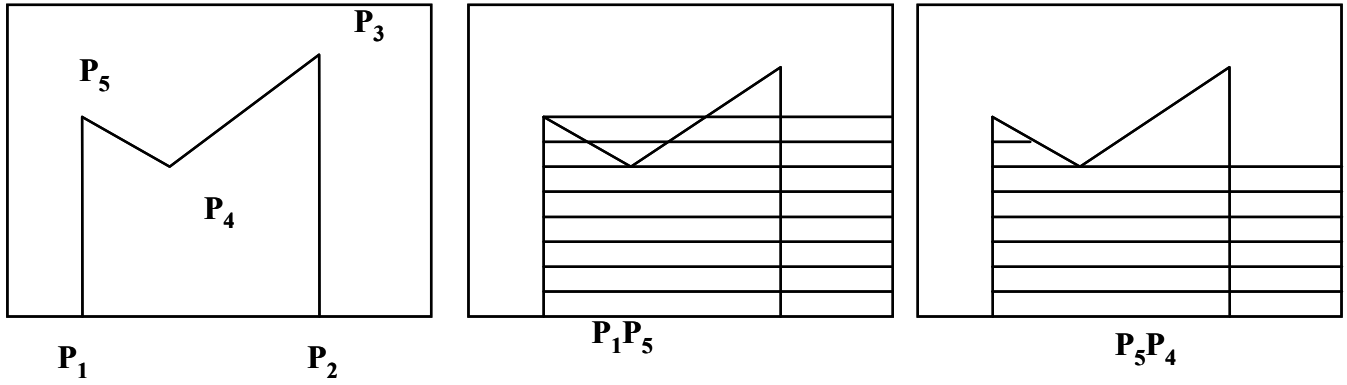

ET:
11
10
9
8
7
6
5
4
3
2
1
0
扫描线11
扫描线12
e_4
e_5
11 13 6/4
11 13 0
e_3
e_4
9 7 -5/2
11 7 6/4
e_5
11 13 0
e_2
9 2 0
e_1
e_6
3 7 -5/2
5 7 6/4
y_{max} x_{min} $1/m$


(13,11)
(2,9)
(7,7)
(13,5)
(2,3)
(7,1)
e_1
e_2
e_3
e_4
e_5
e_6
0
2
4
6
8
10
12
14


无标题 - Polygonfill
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪
数字

四、边填充算法(正负相消法)

基本思想：对于每一条扫描线和每条多边形边的交点 (x_s, y_s)，都将该扫描线上交点右方的所有像素取补，并对多边形的每条边按一定顺序(逆时针、顺时针均可)做此处理



待填充区域 D $R(x_1 \le x \le x_2, y_1 \le y \le y_2) \supset$ D

$$MASK(x_1 \cdots x_2, y_1 \cdots y_2)$$

$$MASK(x, y) = false \quad (x, y) \in R$$

对区域的每一条边与扫描线的每一个交点 (x_s, y_s)

当 $x \ge x_s$ 做 $MASK(x, y_s) = \overline{MASK(x, y_s)}$

沿D的边界经历一周后，只要 $(x, y) \in D$

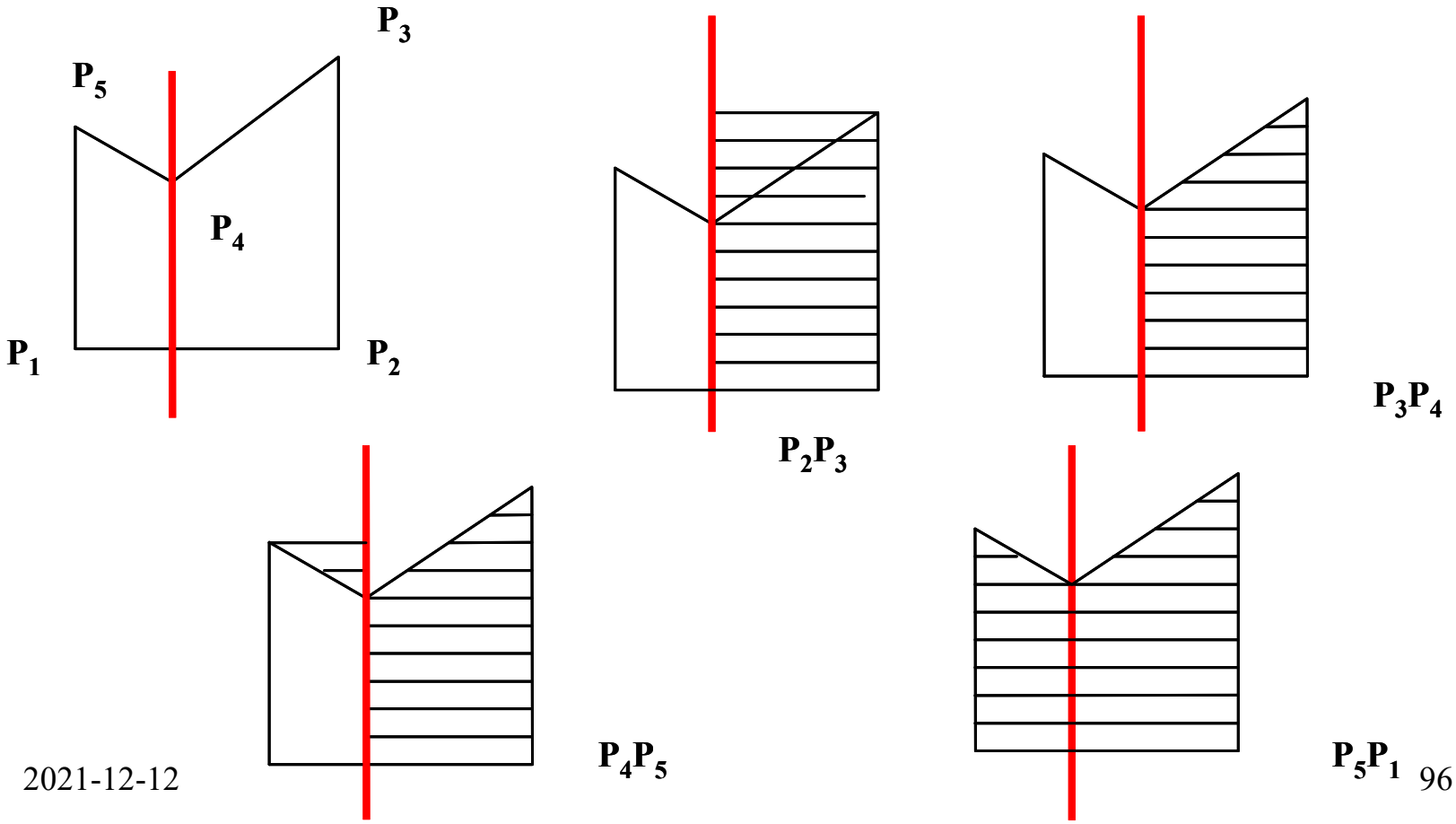
则 $MASK(x, y) = true$ 否则 $MASK(x, y) = false$

```
void EdgeFill(PointArray& ptArray,COLORREF color){
    for(y=y1;y<=y2;y++)
        for(x=x1;x<=x2;x++)
            MASK[y][x]=false;
    for(区域D内的每一条边PiPi+1){
        xs=x[i];  dxs=(x[i+1]- x[i])/(y[i+1]- y[i]);
        dys=abs(y[i+1]- y[i])/(y[i+1]- y[i]);
        for(ys=y[i]; ys!= y[i+1];ys+=dys){
            Ixs =int(xs +0.5);
            for(x= Ixs;x<=x2;x++)
                MASK[ys][x] =!MASK[ys][x];
            xs= xs + dys* dxs;
        }
    }
    for(y=y1;y<=y2;y++)
        for(x=x1;x<=x2;x++)
            if(MASK[y][x])
                SetPixel(x,y,color);
}
```

栅栏(一条与扫描线垂直的直线)

栅栏填充基本思想：

对于每条扫描线与多边形的交点，将交点与栅栏之间的扫描线上的像素取补.



边标志算法(轮廓填充算法)

对每个像素访问一次

边标志算法分为两个步骤：

第一步:边界像素打标志；

第二步:填充内部像素。

边标志算法的程序如下：

```
void EdgeMarkFill(PointArray& ptArray,COLORREF color){
    for(y=y1;y<=y2;y++)
        for(x=x1;x<=x2;x++)
            MASK[y][x]=false; //形成轮廓线
    for(区域D内的每一条边PiPi+1){
        xs=x[i];dxs=(x[i+1]-x[i])/(y[i+1]- y[i]);
        dys=abs(y[i+1]- y[i])/(y[i+1]- y[i]);
        for(ys=y[i]; ys!= y[i+1];ys+=dys){
            Ixs =int(xs +0.5);
            MASK[ys][Ixs]=!MASK[ys][Ixs];
            xs= xs +dys*dxs;
        }
    }
```


```
        for(y=y1;y<=y2;y++){//按轮廓线填充
                inside=false;
                for(x=x1;x<=x2;x++){
                        if(MASK[y][x])
                                inside=!inside;
                        if(inside)
                                SetPixel(x,y,color);
                }
        }
}
```

五、图案填充

图案填充的主要问题是要决定图案区域和填充区域的相互位置关系

第一种方法是采用将图案粘贴在多边形的顶点位置，填充图案的视觉效果也是一致的。

第二种方法是采用绝对定位，即认为整个屏幕被图案覆盖，则标准的粘贴位置是屏幕的原点。其视觉效果可能是有差异的。

设图案通常是由较小的m×n像素阵列pattern[m][n]组成的，图案原点对应屏幕的坐标原点(对窗口处理而言，图案原点对应窗口坐标系的原点)，这样就可以用模运算x mod n和y mod m来获取x，y像素位置所对应的图案像素。


```
void PatternFill(int x, int y, COLORREF* pattern, int m, int n){
        SetPixel(x,y,pattern[y%m][x%n]);
}
void ShadowFill(int x, int y, COLORREF color) {// 斜影线填充
        const int p=5;
        if((x-y)%p==0)SetPixel(x, y,color);
}
void ShadowFill(int x, int y, COLORREF color) { // 对角交叉影线填充
        const int p=5;
        if(((x+y)%p==0)|| ((x-y)%p==0))SetPixel(x, y,color);
}
```

m	Δx	Δy	Δp(p≥0)	Δp(p<0)
0≤m≤1	>0	>0	$\Delta p = 2dy - 2dx$	$\Delta p = 2dy$
	<0	<0	$\Delta p = -2dy + 2dx$	$\Delta p = -2dy$
-1≤m≤0	>0	<0	$\Delta p = 2dy - 2dx$	$\Delta p = -2dy$
	<0	>0	$\Delta p = -2dy + 2dx$	$\Delta p = 2dy$
m>1	>0	>0	$\Delta p = 2dx - 2dy$	$\Delta p = 2dx$
	<0	<0	$\Delta p = -2dx + 2dy$	$\Delta p = -2dx$
m<-1	>0	<0	$\Delta p = 2dx - 2dy$	$\Delta p = -2dx$
	<0	>0	$\Delta p = -2dx + 2dy$	$\Delta p = 2dx$


```
void BresenhamLine(int x1,int y1,int x2,int y2){
        int x,y,dx,dy,p,xmin,ymin,xmax,ymax,lx,ly,deltax,deltay;
        dx=x2-x1;   dy=y2-y1;
        xmin=min(x1,x2);        xmax=max(x1,x2);
        ymin=min(y1,y2);        ymax=max(y1,y2);
        lx=xmax-xmin;           ly=ymax-ymin;
        deltax= (dx==0)?0:(dx>0?1:-1);      deltay= (dy==0)?0:(dy>0?1:-1);
        x=x1;       y=y1;
        if(lx>ly){  p=2*dy-dx;
                    for(;x!=x2;x+=deltax){
                                SetPixel(x,y,RGB(0,0,0));
                                if(p>=0){   y+=deltay;p+=2*(ly-lx);
                                }else{      p+=2*ly;
                                }
                    }
        }else{      p=2*dx-dy;
                    for(;y!=y2;y+=deltay){
                                SetPixel(x,y,RGB(0,0,0));
                                if(p>=0){   x+=deltax;p+=2*(lx-ly);
                                }else{      p+=2*lx;
                                }
                    }
        }
}
```

```
void CBresenhamView::BresenhamLine(CDC *pDC, int x1,int y1,int x2,int y2){
        int x,y,dx,dy,p,temp;
        int deltay,deltap1,deltap2,deltax1,deltax2,deltay1,deltay2;
        if(x1>x2){temp = x1;x1 = x2;x2 = temp;temp = y1;y1 = y2;y2 = temp;}
        x=x1;y=y1;
        if(y1<=y2){ deltay=1;
        }else{      deltay=-1;   y1 = -y1;y2 = -y2;}
        dx=x2-x1;dy=y2-y1;
        if(abs(dx)>abs(dy)){
                p=2*dy-dx;deltap1 = 2*(dy-dx);deltap2 = 2*dy;
                deltax1 = 1;deltax2 = 1;
                deltay1 = deltay;deltay2 = 0;
        }else{
                p=2*dx-dy;deltap1 = 2*(dx-dy);deltap2 = 2*dx;
                deltax1 = 1;deltax2 = 0;
                deltay1 = deltay;deltay2 = deltay;
        }
        for(int i=0;i<max(abs(dx),abs(dy));i++) {
                pDC->SetPixel(x,y,RGB(0,0,0));
                if(p>=0) {    x+=deltax1;y+=deltay1; p+=deltap1;
                }else {       x+=deltax2;y+=deltay2; p+=deltap2;
                }
        }
}
```