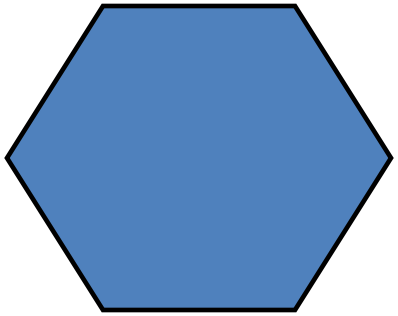
Computer Graphics
Polygon Filling
By
Dr. R. Srivastava

Polygon Filling

Types of filling

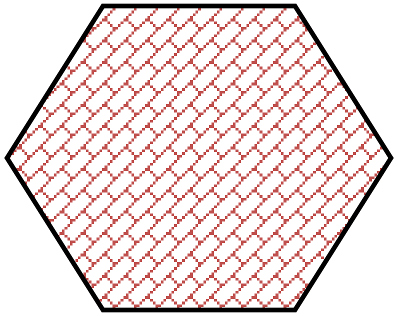
- **Solid-fill**

All the pixels inside the polygon's boundary are illuminated.



- **Pattern-fill**

the polygon is filled with an arbitrary predefined pattern.



Polygon Representation

The polygon can be represented by listing its n vertices in an ordered list.

$$\mathbf{P = \{(x_1, y_1), (x_2, y_2), \ldots\ldots\ldots, (x_n, y_n)\}}.$$

The polygon can be displayed by drawing a line between **(x_1, y_1)**, and **(x_2, y_2)**, then a line between **(x_2, y_2)**, and **(x_3, y_3)**, and so on until the end vertex. In order to close up the polygon, a line between **(x_n, y_n)**, and **(x_1, y_1)** must be drawn.

One problem with this representation is that if we wish to translate the polygon, it is necessary to apply the translation transformation to each vertex in order to obtain the translated polygon.

Polygon Representation

For objects described by many polygons with many vertices, this can be a time consuming process.

One method for reducing the computational time is to represent the polygon by the (**absolute**) **location** of its first vertex, and represent subsequent vertices as **relative positions** from the previous vertex. This enables us to translate the polygon simply by changing the coordinates of the first vertex.

Inside-Outside Tests

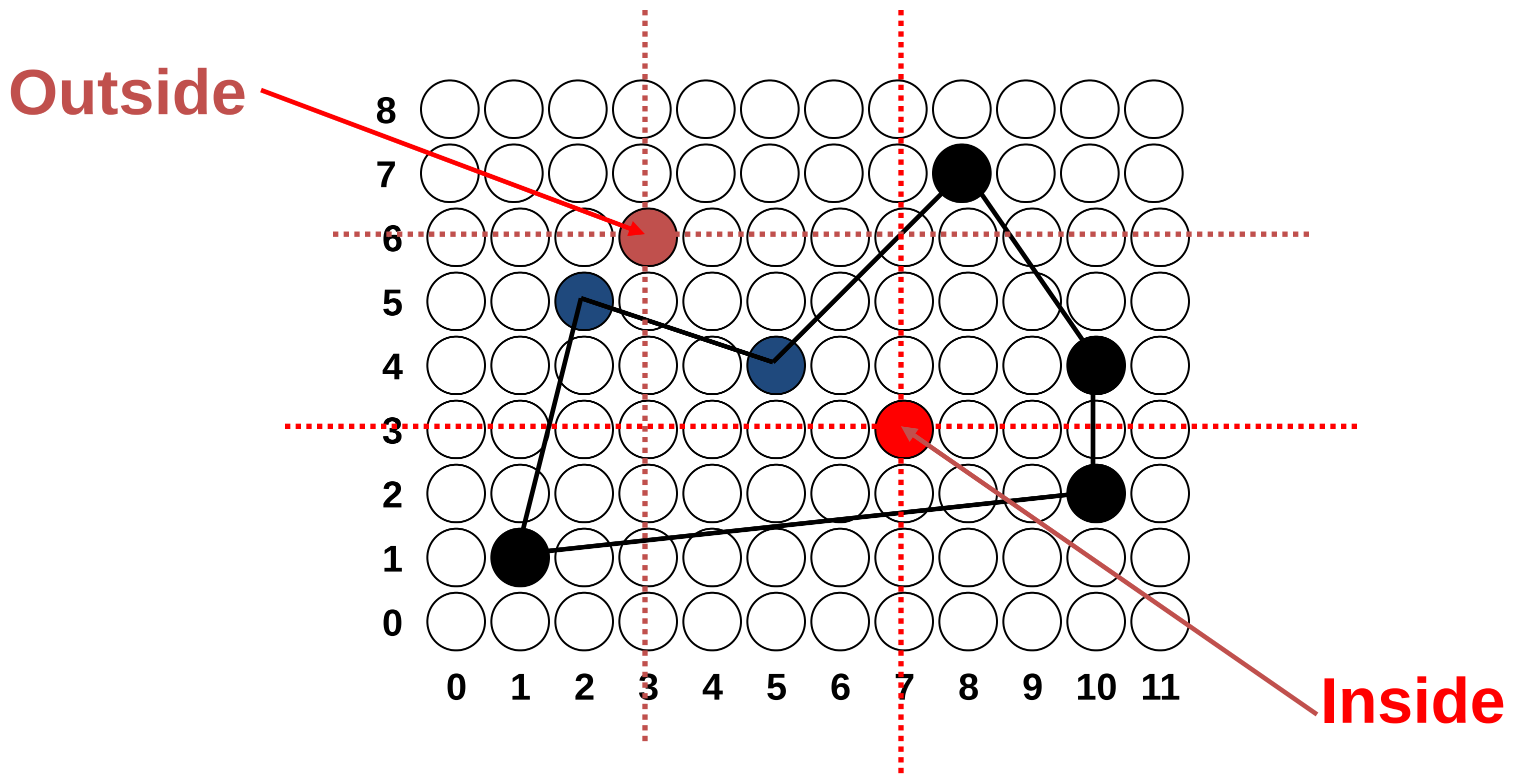
when filling polygons we should decide whether a particular point is interior or exterior to a polygon.

A rule called the **odd-parity** (or the **odd-even rule**) is applied to test whether a point is interior or not.

To apply this rule, we conceptually draw a line starting from the particular point and extending to a distance point outside the coordinate extends of the object in any direction such that **no polygon vertex intersects** with **the line**.

Inside-Outside Tests

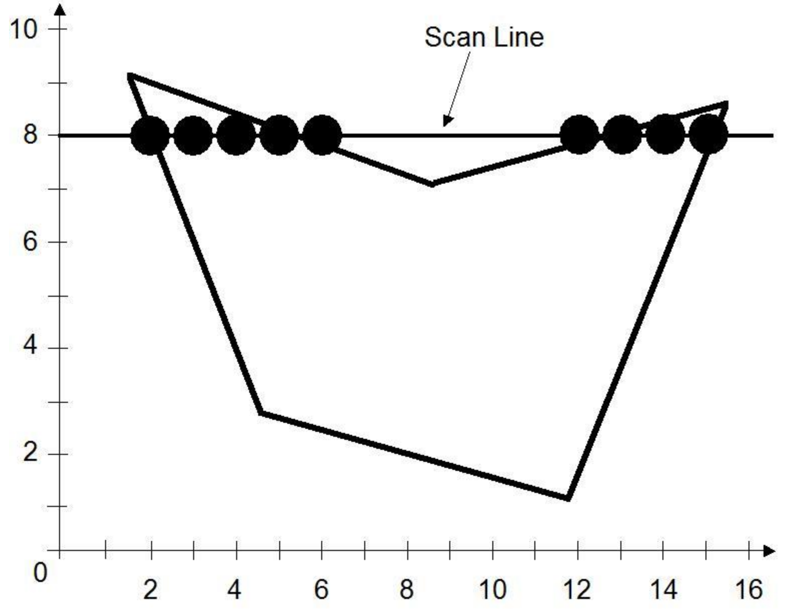
The point is considered to be **interior** if the number of intersections between the line and the polygon edges is **odd**. Otherwise, The point is exterior point.



The Scan-Line Polygon Fill Algorithm

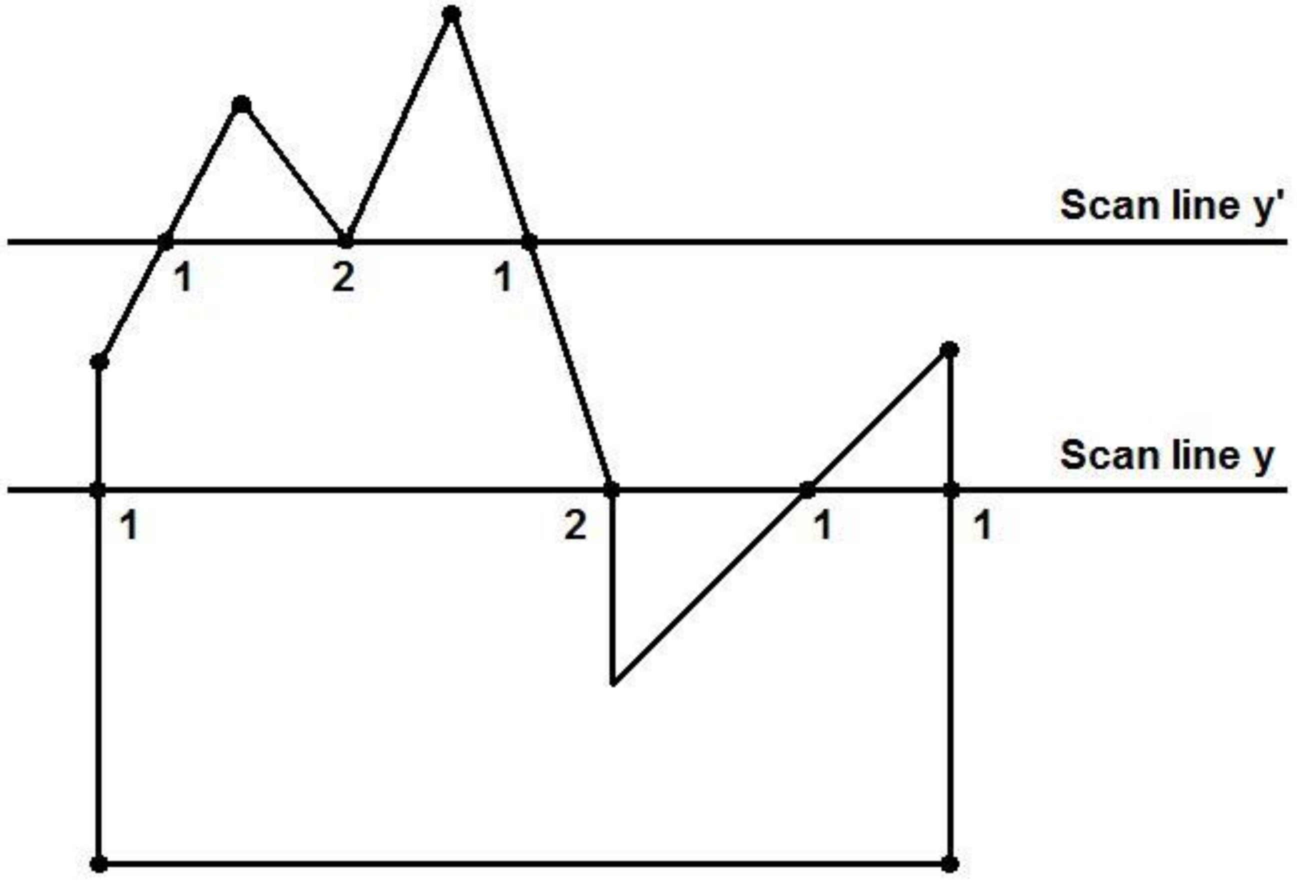
The scan-line polygon-filling algorithm involves

- the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,
- identifying which edges intersect the scan-line,
- and finally drawing the interior horizontal lines with the specified fill color. process.



The Scan-Line Polygon Fill Algorithm

Dealing with vertices

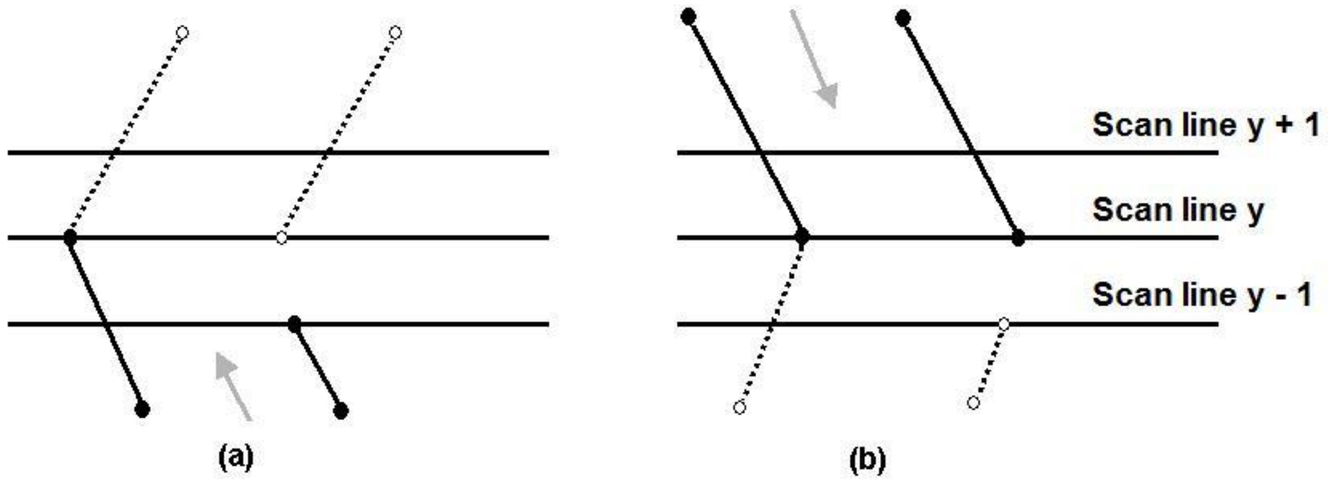


The Scan-Line Polygon Fill Algorithm

Dealing with vertices

- When the endpoint **y** coordinates of the two edges are **increasing**, the **y** value of the upper endpoint for the **current edge** is decreased by one (a)

- When the endpoint **y** values are **decreasing**, the **y** value of the **next edge** is decreased by one (b)



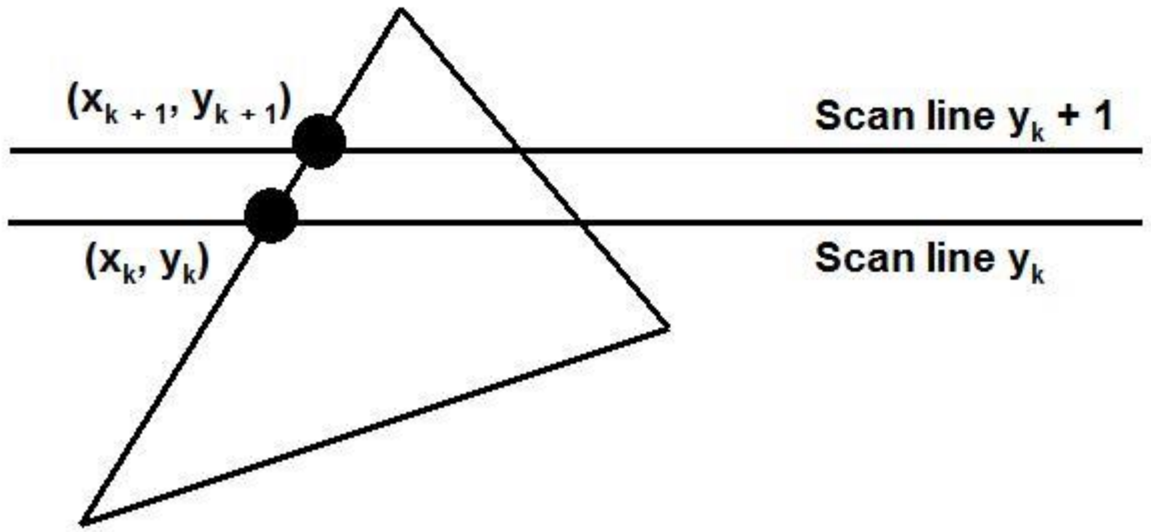
The Scan-Line Polygon Fill Algorithm

Determining Edge Intersections

$$m = (y_{k+1} - y_k) / (x_{k+1} - x_k)$$

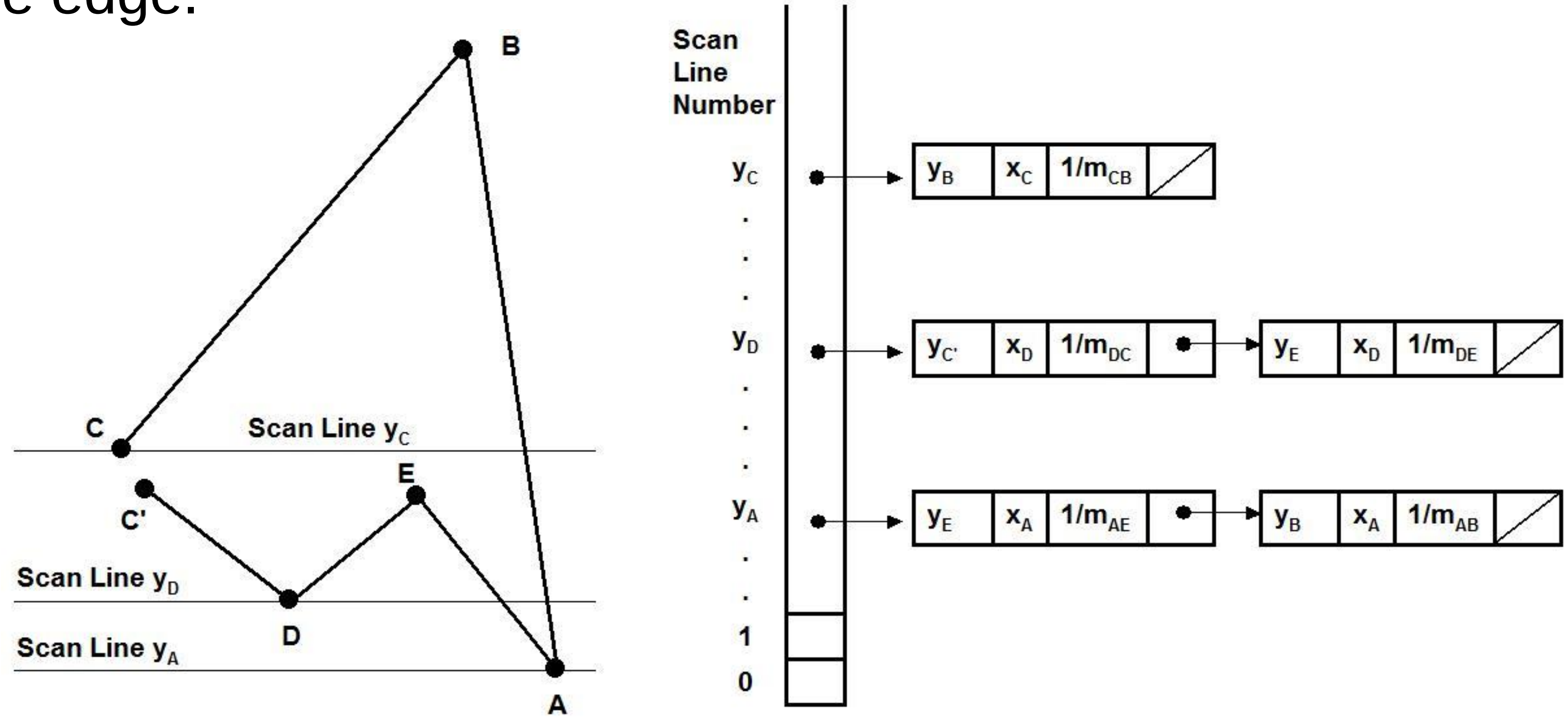
$$y_{k+1} - y_k = 1$$

$$x_{k+1} = x_k + 1/m$$



The Scan-Line Polygon Fill Algorithm

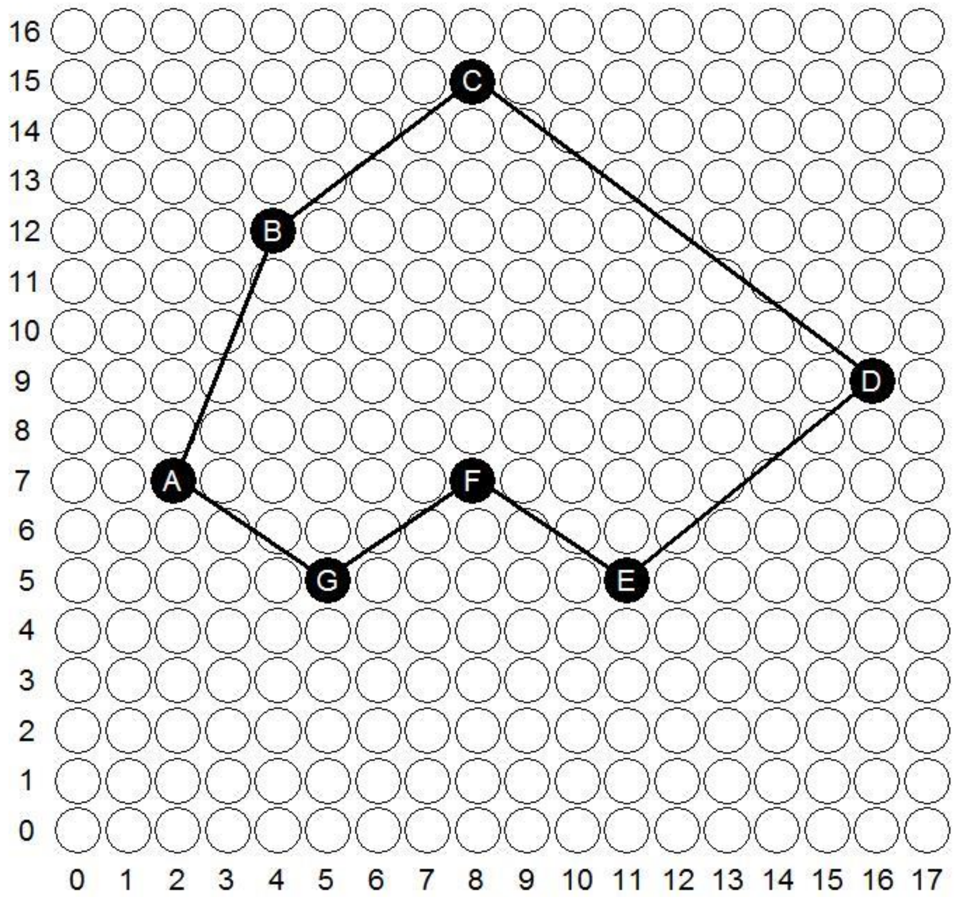
- Each **entry** in the table for a particular scan line contains the **maximum y** value for that edge, the **x-intercept** value (**at the lower vertex**) for the edge, and the **inverse slope** of the edge.

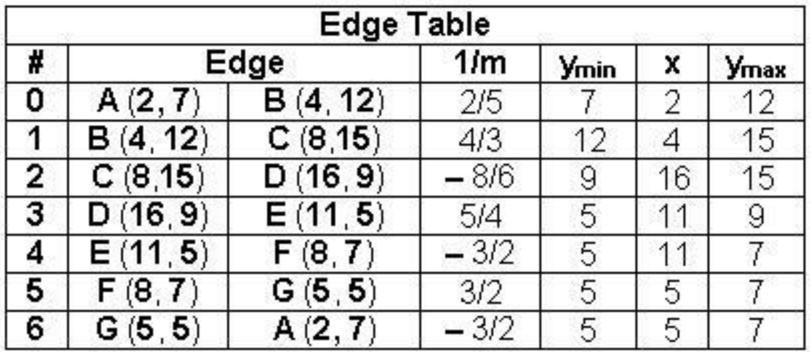


The Scan-Line Polygon Fill Algorithm

(Example) Polygon = {A, B, C, D, E, F, G}

Polygon = {(2, 7), (4, 12), (8,15), (16, 9), (11, 5), (8, 7), (5, 5)}



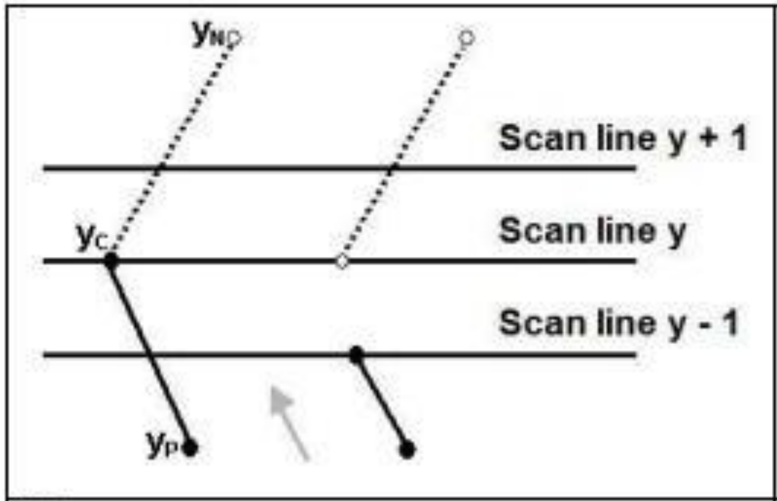
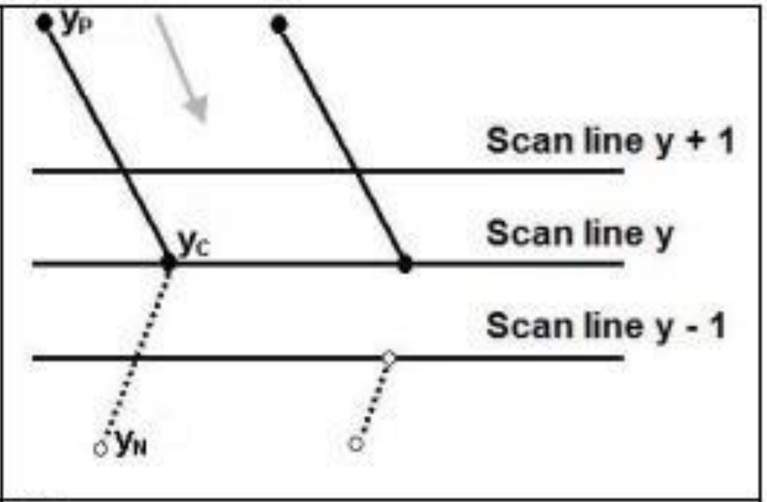


Edge Table

#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B (4, 12)	2/5	7	2	12
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D (16, 9)	E (11, 5)	5/4	5	11	9
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A (2, 7)	– 3/2	5	5	7

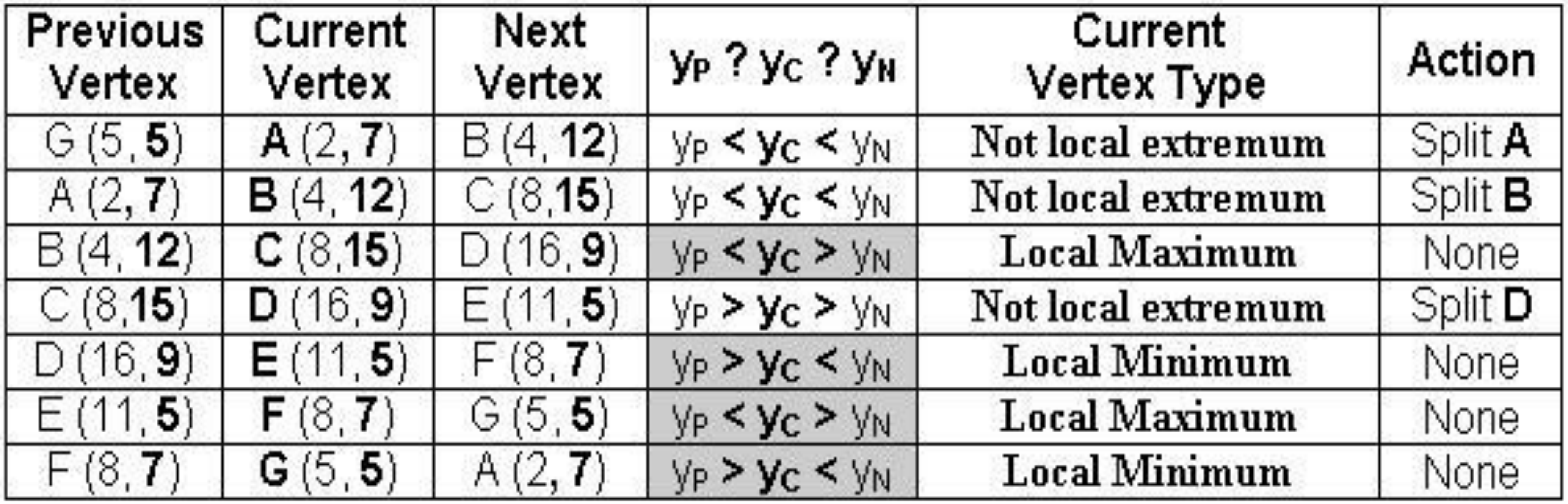
The Scan-Line Polygon Fill Algorithm

(Example)

 y_N Scan line y + 1 y_C Scan line y Scan line y - 1 y_P	 y_P Scan line y + 1 y_C Scan line y Scan line y - 1 y_N
If $y_P < y_C < y_N$ **Then** y_C is decreased by one. The new edges become $(x_P, y_P) \rightarrow (x'_C, y_C - 1)$ and $(x_C, y_C) \rightarrow (x_N, y_N)$	**If** $y_P > y_C > y_N$ **Then** y_C is decreased by one. The new edges become $(x_P, y_P) \rightarrow (x_C, y_C)$ and $(x'_C, y_C - 1) \rightarrow (x_N, y_N)$
$m = (y_P - y_C) / (x_P - x_C)$ $x'_C = x_P + (1/m)(y_C - 1 - y_P)$	$m = (y_N - y_C) / (x_N - x_C)$ $x'_C = x_N + (1/m)(y_C - 1 - y_N)$

The Scan-Line Polygon Fill Algorithm

(Example)



Previous Vertex	Current Vertex	Next Vertex	$y_P ? y_C ? y_N$	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	Not local extremum	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8, **15**)	$y_P < y_C < y_N$	Not local extremum	Split **B**
B (4, **12**)	**C** (8, **15**)	D (16, **9**)	$y_P < y_C > y_N$	Local Maximum	None
C (8, **15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	Not local extremum	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	Local Minimum	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	Local Maximum	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	Local Minimum	None

- **Vertex A** should be split into two vertices **A'** ($x_{A'}$, **6**) and **A**(**2, 7**)

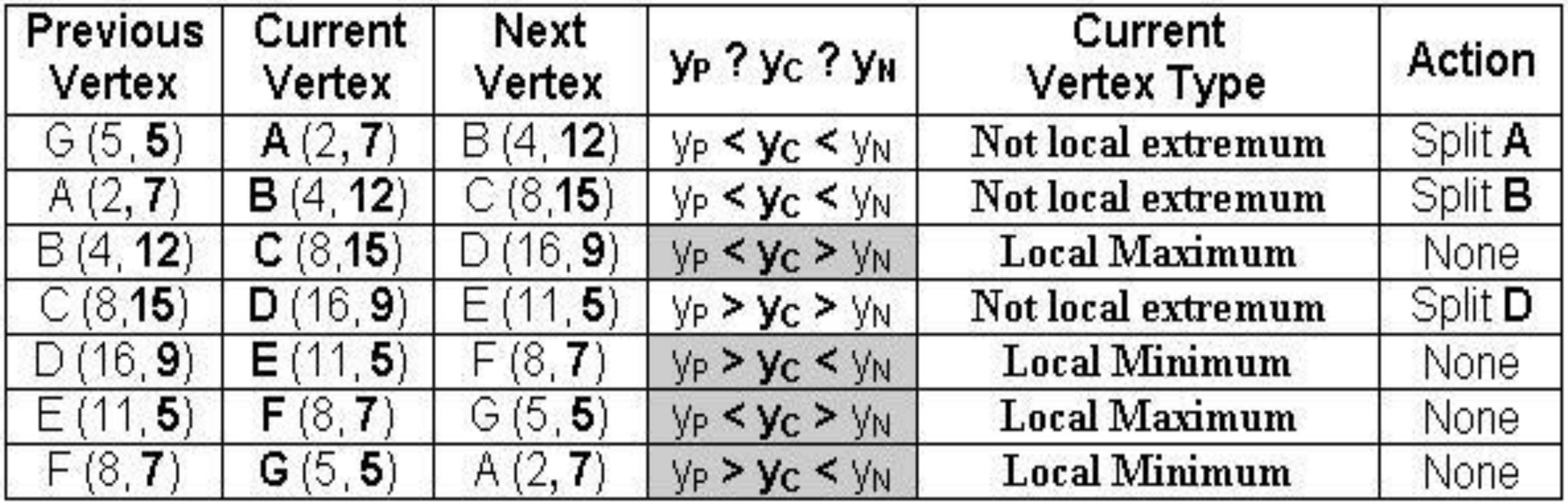
$$m = (5 - 7)/(5 - 2) = -2/3$$

$$x'_A = 5 + (-3/2)(7 - 1 - 5) = 7/2 = 3.5 \cong 4$$

The vertex **A** is split to **A'** (**4**, **6**) and **A**(**2, 7**)

The Scan-Line Polygon Fill Algorithm

(Example)



Previous Vertex	Current Vertex	Next Vertex	$y_P\ ?\ y_C\ ?\ y_N$	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	Not local extremum	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8,**15**)	$y_P < y_C < y_N$	Not local extremum	Split **B**
B (4, **12**)	**C** (8,**15**)	D (16, **9**)	$y_P < y_C > y_N$	Local Maximum	None
C (8,**15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	Not local extremum	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	Local Minimum	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	Local Maximum	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	Local Minimum	None

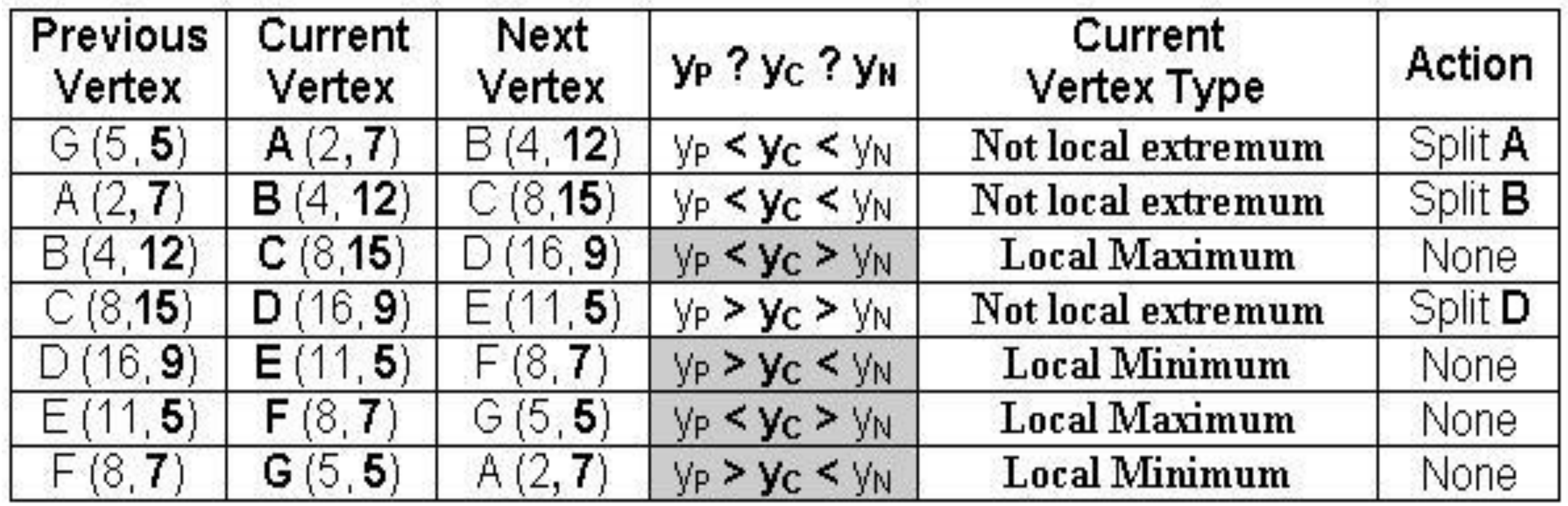
- **Vertex B** should be split into two vertices **B'** ($x_{B'}$, **11**) and **B**(**4, 12**)

$$m = (7 - 12)/(2 - 4) = 5/2$$

$$x'_A = 2 + (2/5)(12 - 1 - 7) = 18/5 = 3.6 \cong 4$$

The vertex **B** is split to **B'** (**4**, **11**) and **B(4, 12)**

The Scan-Line Polygon Fill Algorithm
(Example)



Previous Vertex	Current Vertex	Next Vertex	y_P ? y_C ? y_N	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	Not local extremum	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8, **15**)	$y_P < y_C < y_N$	Not local extremum	Split **B**
B (4, **12**)	**C** (8, **15**)	D (16, **9**)	$y_P < y_C > y_N$	Local Maximum	None
C (8, **15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	Not local extremum	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	Local Minimum	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	Local Maximum	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	Local Minimum	None

- **Vertex D** should be split into two vertices **D(16, 9)** and **D'** ($x_{D'}$, **8**)

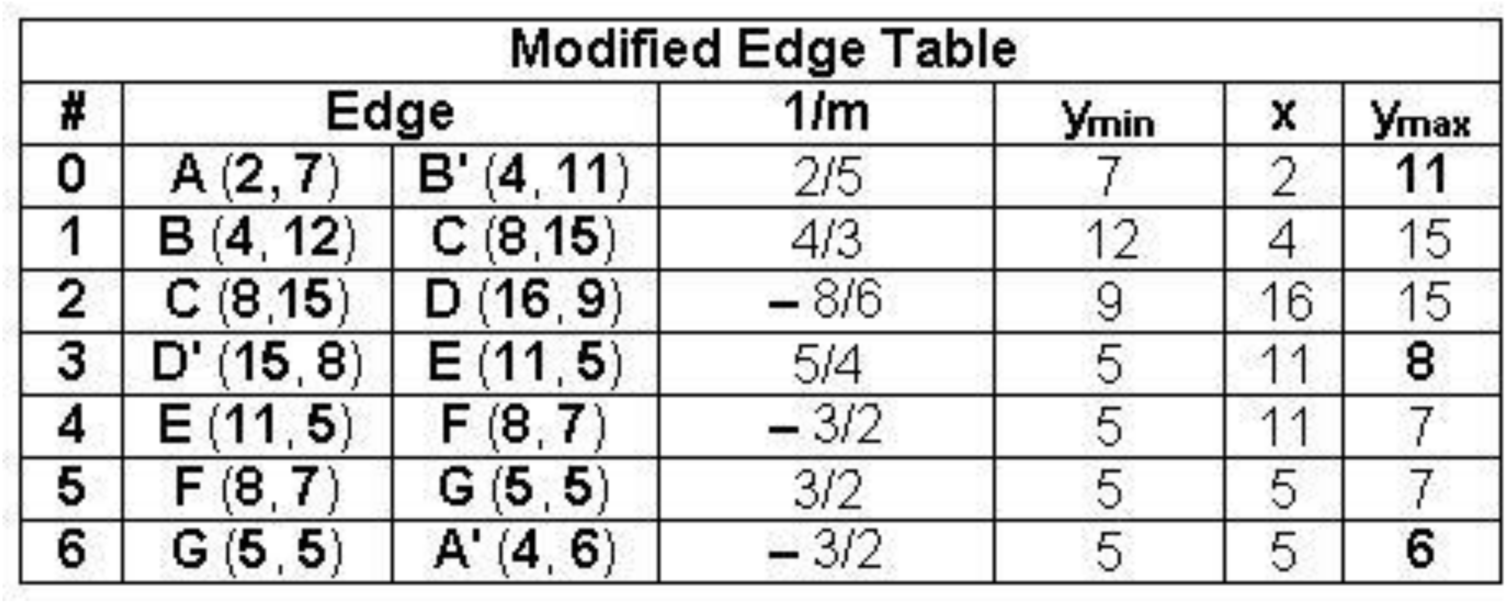
$$m = (5 - 9)/(11 - 16) = 4/5$$

$$x'_D = 11 + (5/4)(9 - 1 - 5) = 59/4 = 14.75 \cong 15$$

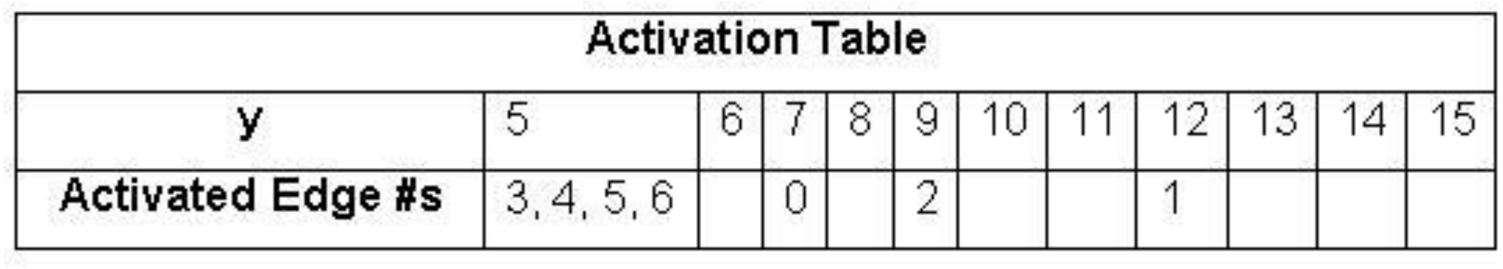
The vertex **D** is split to **D(16, 9)** and **D'** (**15**, **8**)

The Scan-Line Polygon Fill Algorithm

(Example)



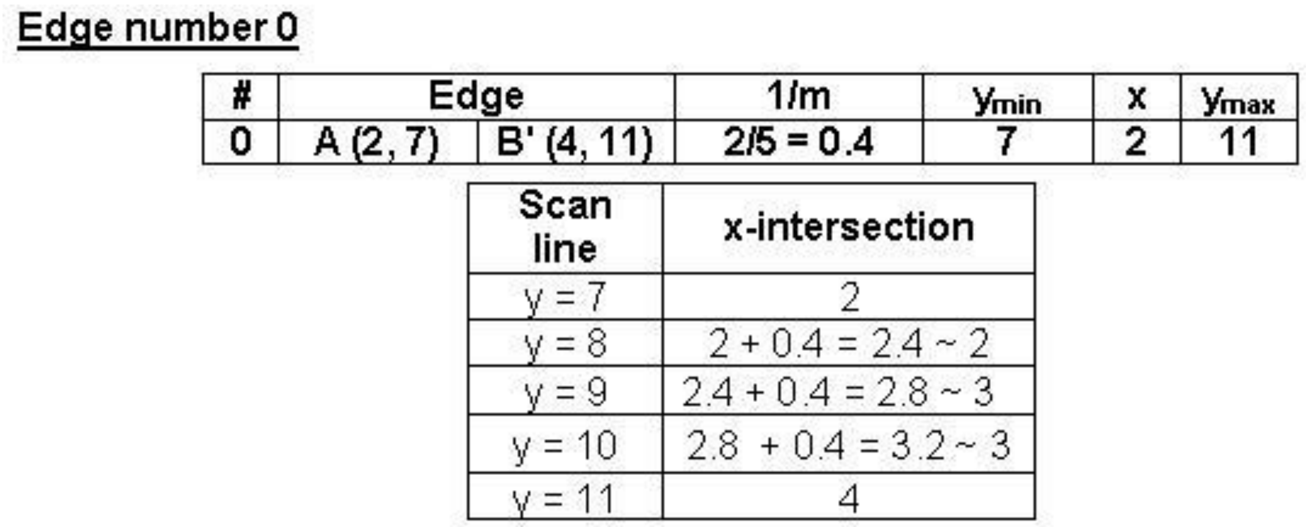
Modified Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B' (4, 11)	2/5	7	2	**11**
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D' (15, 8)	E (11, 5)	5/4	5	11	**8**
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A' (4, 6)	– 3/2	5	5	**6**



Activation Table											
y	5	6	7	8	9	10	11	12	13	14	15
Activated Edge #s	3, 4, 5, 6		0		2			1			

The Scan-Line Polygon Fill Algorithm

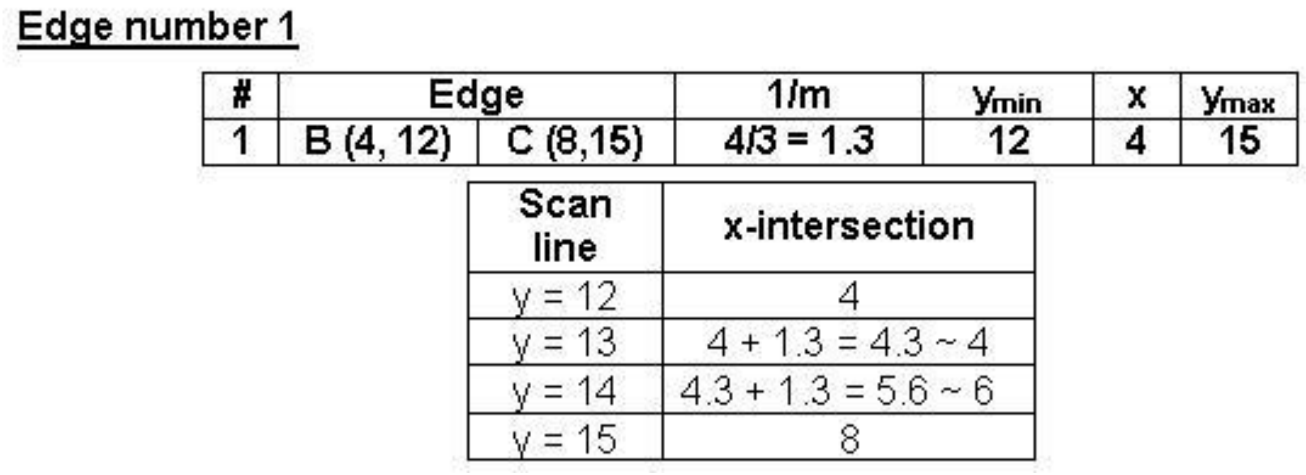
(Example)



Edge number 0

#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B' (4, 11)	2/5 = 0.4	7	2	11

Scan line	x-intersection
y = 7	2
y = 8	2 + 0.4 = 2.4 ~ 2
y = 9	2.4 + 0.4 = 2.8 ~ 3
y = 10	2.8 + 0.4 = 3.2 ~ 3
y = 11	4



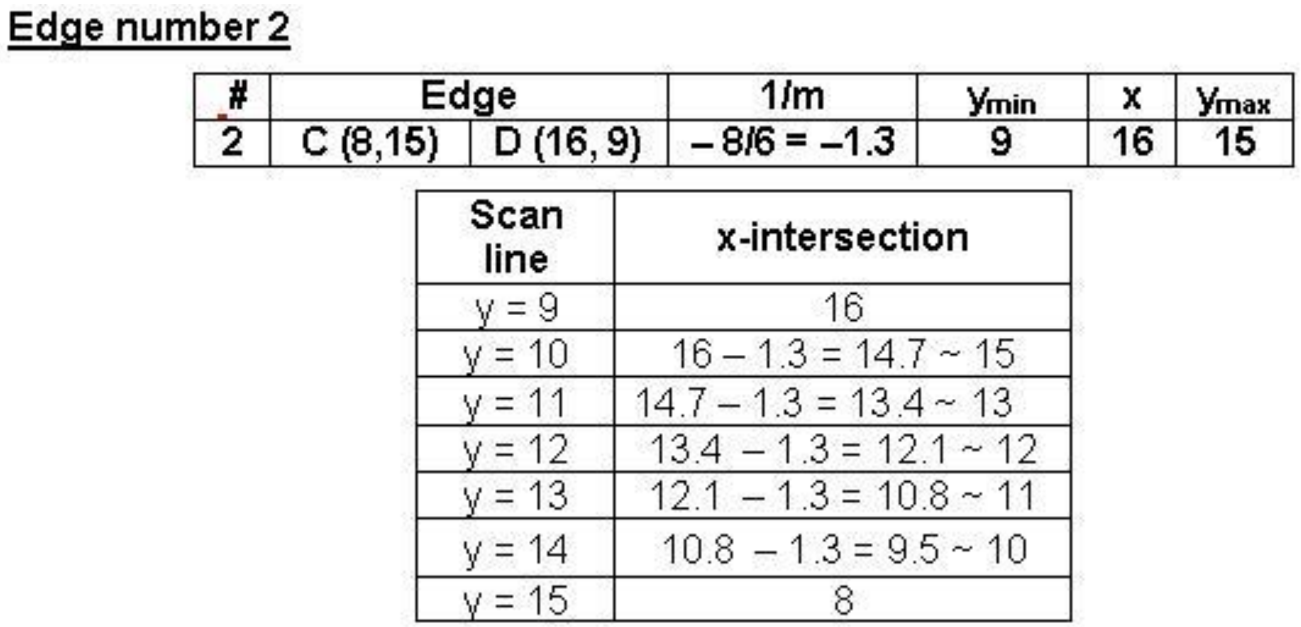
Edge number 1

#	Edge		1/m	y_{min}	x	y_{max}
1	B (4, 12)	C (8,15)	4/3 = 1.3	12	4	15

Scan line	x-intersection
y = 12	4
y = 13	4 + 1.3 = 4.3 ~ 4
y = 14	4.3 + 1.3 = 5.6 ~ 6
y = 15	8

The Scan-Line Polygon Fill Algorithm

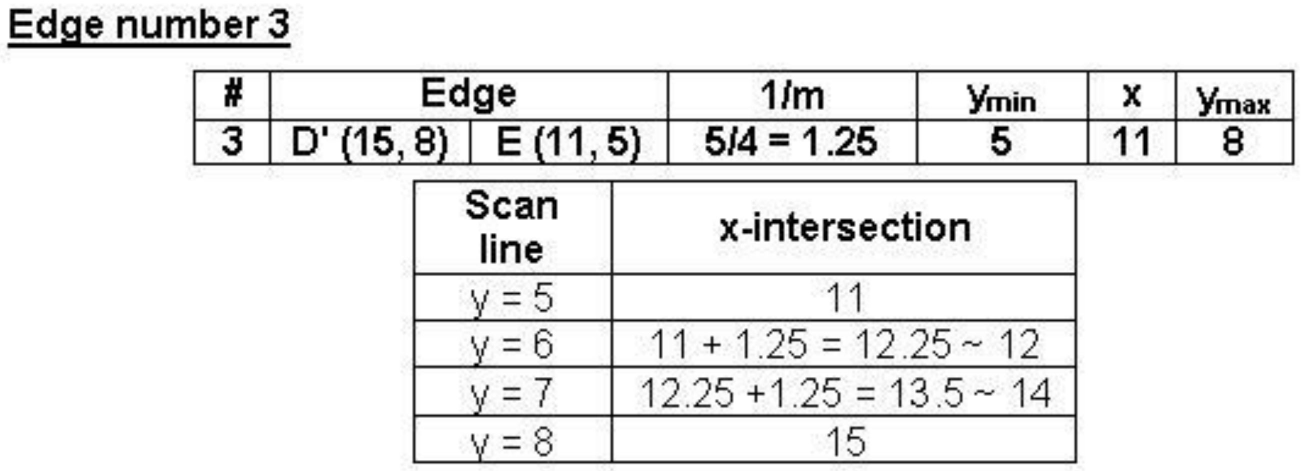
(Example)



Edge number 2

#	Edge		1/m	y_{min}	x	y_{max}
2	C (8,15)	D (16, 9)	– 8/6 = –1.3	9	16	15

Scan line	x-intersection
y = 9	16
y = 10	16 – 1.3 = 14.7 ~ 15
y = 11	14.7 – 1.3 = 13.4 ~ 13
y = 12	13.4 – 1.3 = 12.1 ~ 12
y = 13	12.1 – 1.3 = 10.8 ~ 11
y = 14	10.8 – 1.3 = 9.5 ~ 10
y = 15	8



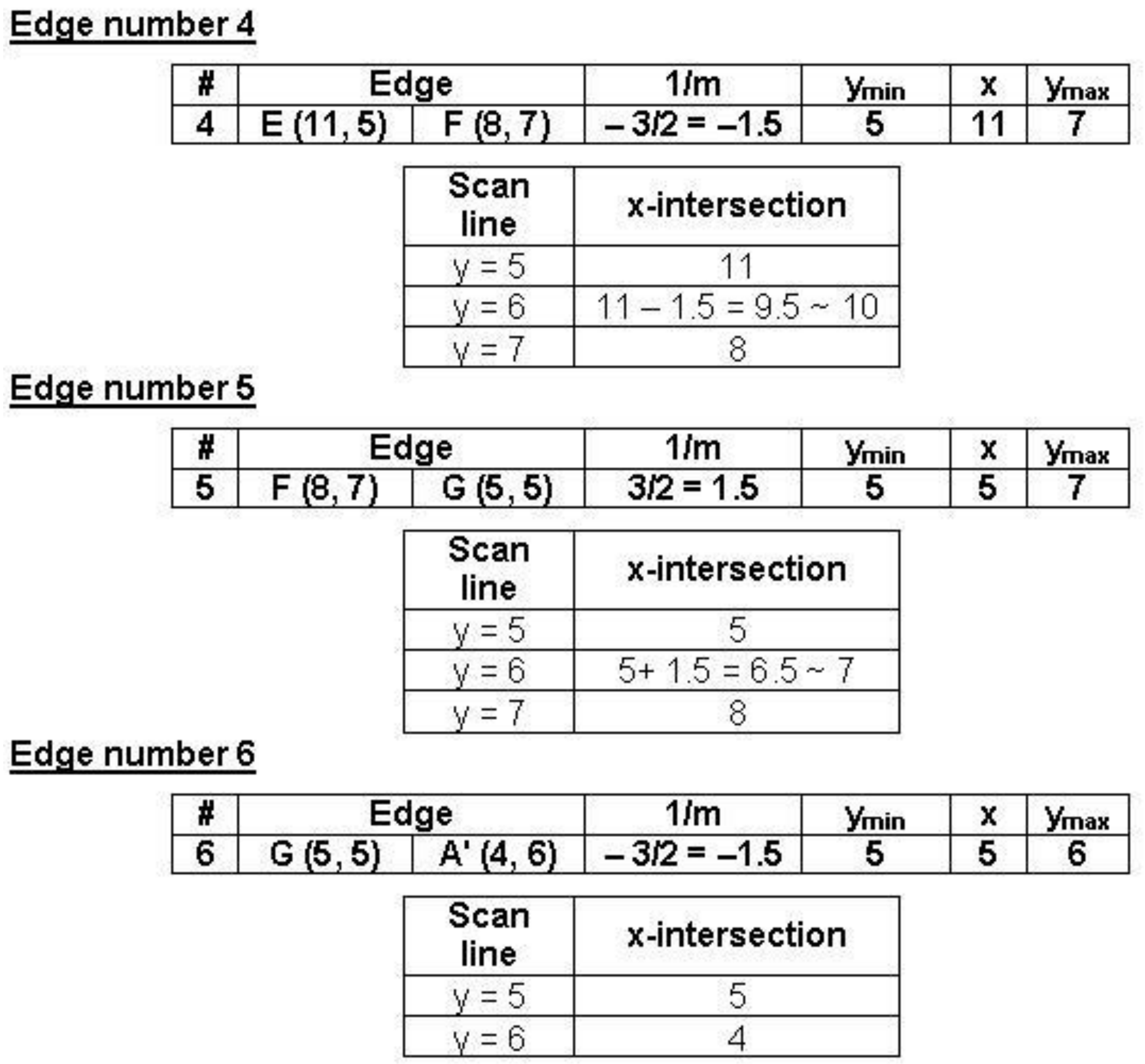
Edge number 3

#	Edge		1/m	y_{min}	x	y_{max}
3	D' (15, 8)	E (11, 5)	5/4 = 1.25	5	11	8

Scan line	x-intersection
y = 5	11
y = 6	11 + 1.25 = 12.25 ~ 12
y = 7	12.25 +1.25 = 13.5 ~ 14
y = 8	15

The Scan-Line Polygon Fill Algorithm

(Example)



Edge number 4

#	Edge		1/m	y_{min}	x	y_{max}
4	E (11, 5)	F (8, 7)	– 3/2 = –1.5	5	11	7

Scan line	x-intersection
y = 5	11
y = 6	11 – 1.5 = 9.5 ~ 10
y = 7	8

Edge number 5

#	Edge		1/m	y_{min}	x	y_{max}
5	F (8, 7)	G (5, 5)	3/2 = 1.5	5	5	7

Scan line	x-intersection
y = 5	5
y = 6	5+ 1.5 = 6.5 ~ 7
y = 7	8

Edge number 6

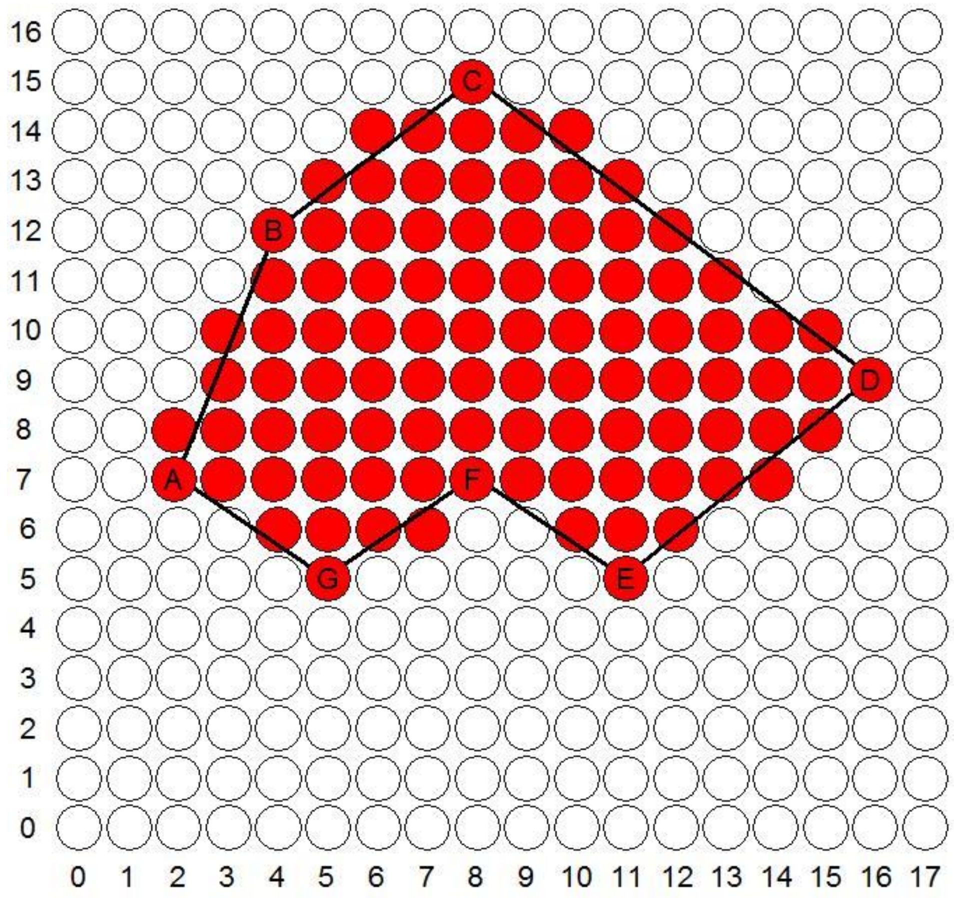
#	Edge		1/m	y_{min}	x	y_{max}
6	G (5, 5)	A' (4, 6)	– 3/2 = –1.5	5	5	6

Scan line	x-intersection
y = 5	5
y = 6	4

The Scan-Line Polygon Fill Algorithm

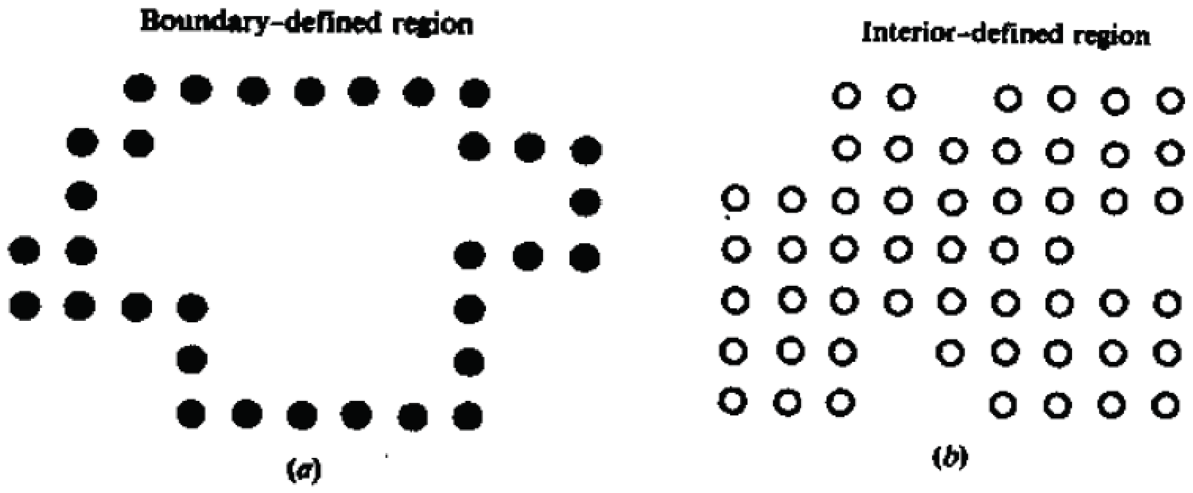
(Example)

Scan line	x-intersections							x-intersections pair Ascending order
	Edge#							
	0	1	2	3	4	5	6	
5				11	11	5	5	(5, 5), (11, 11)
6				12	10	7	4	(4, 7), (10, 12)
7	2			14	8	8		(2, 8), (8,14)
8	2			15				(2,15)
9	3		16					(3, 16)
10	3		15					(3, 15)
11	4		13					(4, 13)
12		4	12					(4,12)
13		4	11					(4, 11)
14		6	10					(6, 10)
15		8	8					(8, 8)

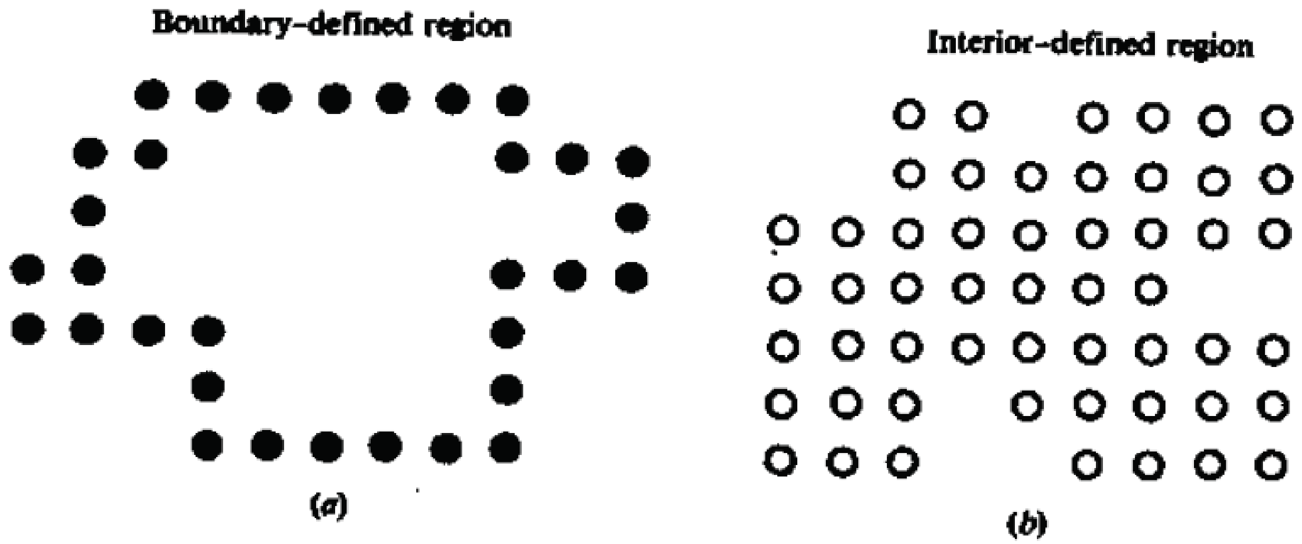


Region Filling

- Region filling is the process of "coloring in" a definite image area or region.
- defined at the pixel or geometric level.
- At the pixel level, we describe a region either in terms
  - of the bounding pixels that outline it
  - or as the totality of pixels that comprise it



Region Filling



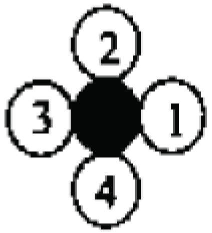
- In the first case the region is called boundary-defined region and the collection of algorithms used for filling such a region are collectively called boundary-fill algorithms.
- The other type of region is called an interior-defined region and the accompanying algorithms are called flood-fill algorithms.

Region Filling

- At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves.
- For example, a polygonal* region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of file last line connected to file beginning of file first line.

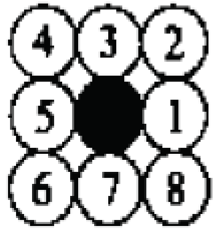
4-Connected vs. 8-Connected

- There are two ways in which pixels are considered connected to each other to form a "continuous" boundary.
- One method is called 4-connected, where a pixel may
  - have up to four neighbors [see Fig. **3-10(a)];**



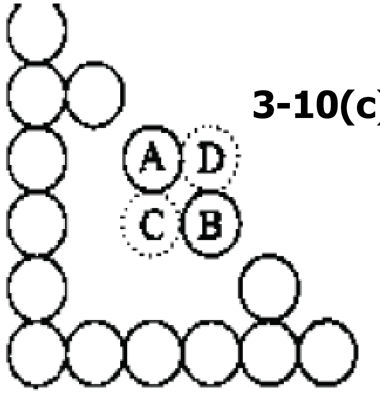
3-10(a)

- **the other is called 8-connectecd where a pixel may**
  - have up to eight neighbors [see Fig. **3-10(b)].**



3-10(b)

- **Using the 4-connected approach, the pixels in Fig.3-10(c) do not define a region since several pixels such as A and B are not connected.**
- **using the 8-connected definition we identify a triangular region.**



3-10(c)

Boundary Fill Algorithm

- In addition to scan line polygon filling, another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary.

- If the boundary is specified in a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered.

- A boundary-fill procedure accepts as input the coordinate of the interior **point (x, y)**, a **fill color**, and a **boundary color**.

Boundary Fill Algorithm

The following steps illustrate the idea of the **recursive** boundary-fill algorithm:

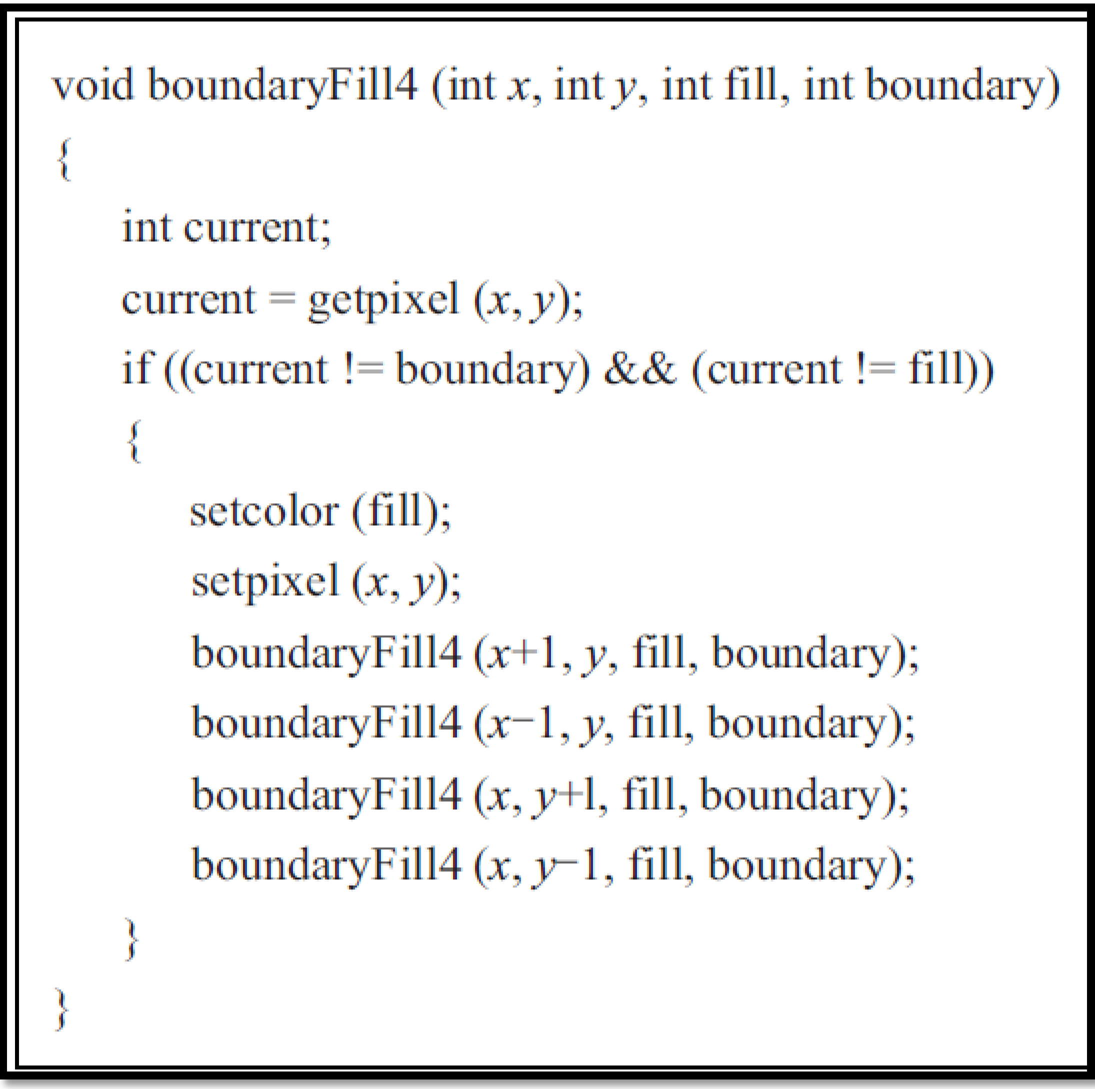
1. Start from an interior point.

2. If the current pixel is **not already** filled and if it is not an edge point, then set the pixel with the fill color, and store its neighboring pixels (**4** or **8-connected**) in the stack for processing. Store only neighboring pixel that is **not already** filled and is not an edge point.

3. Select the next pixel from the stack, and continue with step **2**.

Boundary-Fill Algorithm

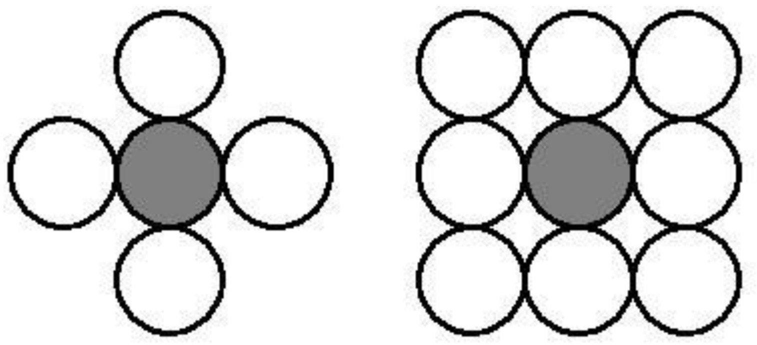
- The procedure illustrates a recursive method for filling a **4-connected area with** an intensity specified in parameter fill up to a boundary color specified with parameter boundary.
- We can extend this procedure to fill an 8-connected region by including four additional statements to test diagonal positions, such as *(x+1, y+1).*



```
void boundaryFill4 (int x, int y, int fill, int boundary)
{
    int current;
    current = getpixel (x, y);
    if ((current != boundary) && (current != fill))
    {
        setcolor (fill);
        setpixel (x, y);
        boundaryFill4 (x+1, y, fill, boundary);
        boundaryFill4 (x−1, y, fill, boundary);
        boundaryFill4 (x, y+l, fill, boundary);
        boundaryFill4 (x, y−1, fill, boundary);
    }
}
```

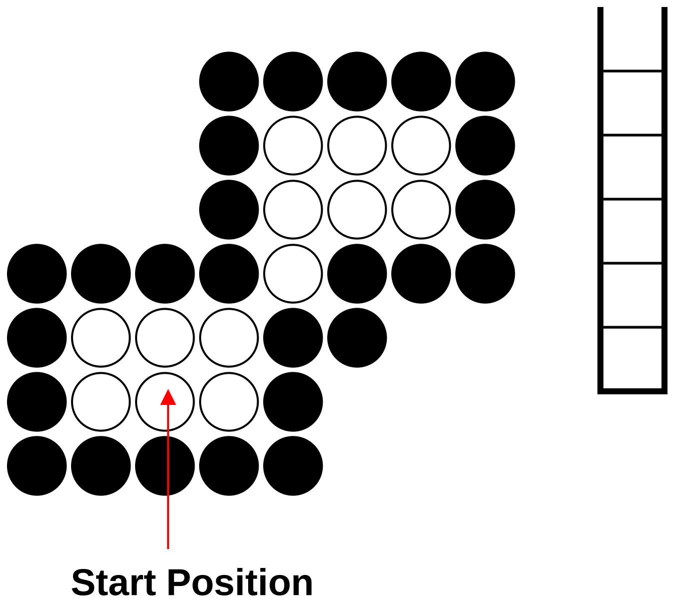
Boundary Fill Algorithm

The order of pixels that should be added to stack using **4-connected** is above, below, left, and right. For **8-connected** is above, below, left, right, above-left, above-right, below-left, and below-right.



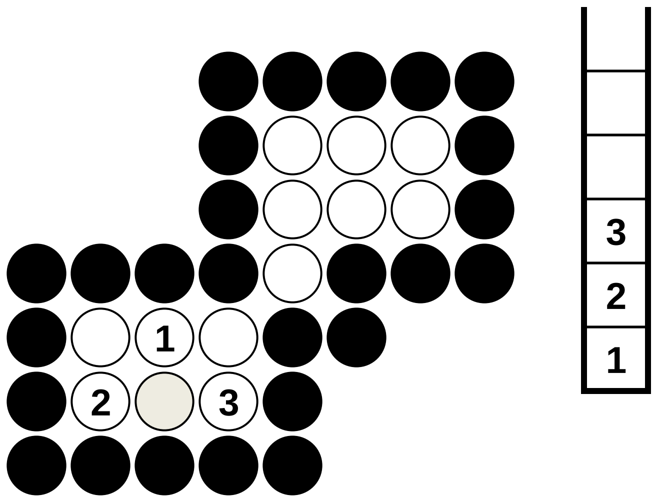
Boundary Fill Algorithm

4-connected (Example)



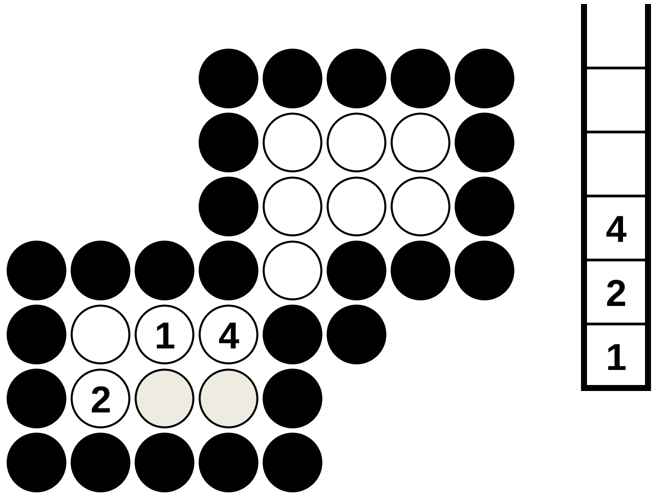
Boundary Fill Algorithm

4-connected (Example)



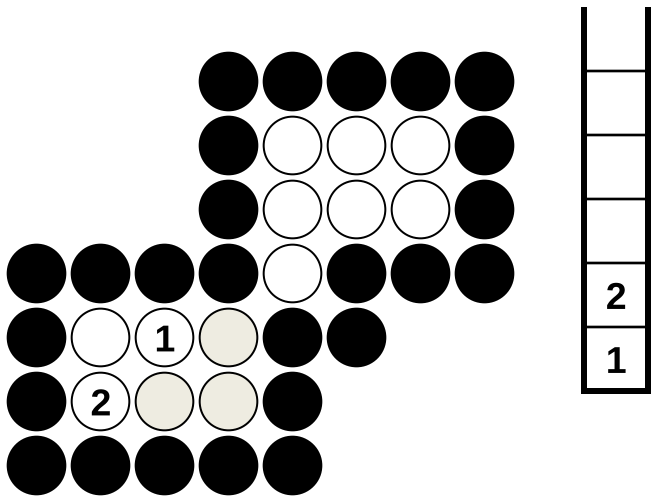
Boundary Fill Algorithm

4-connected (Example)



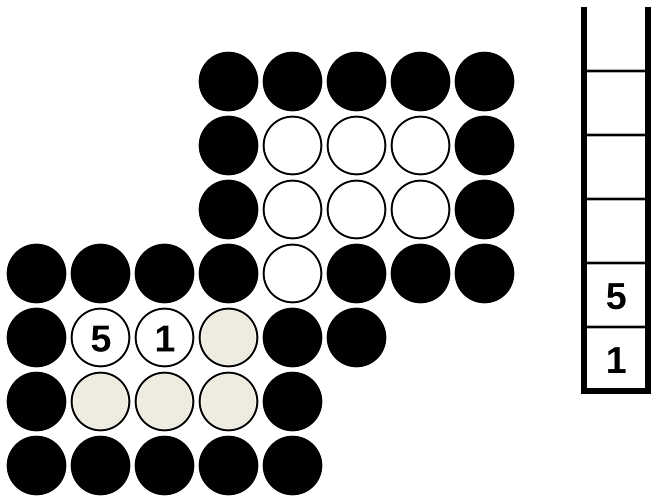
Boundary Fill Algorithm

4-connected (Example)



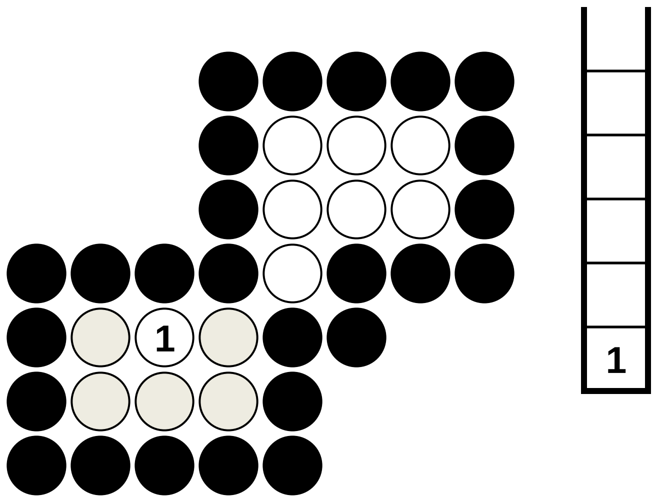
Boundary Fill Algorithm

4-connected (Example)



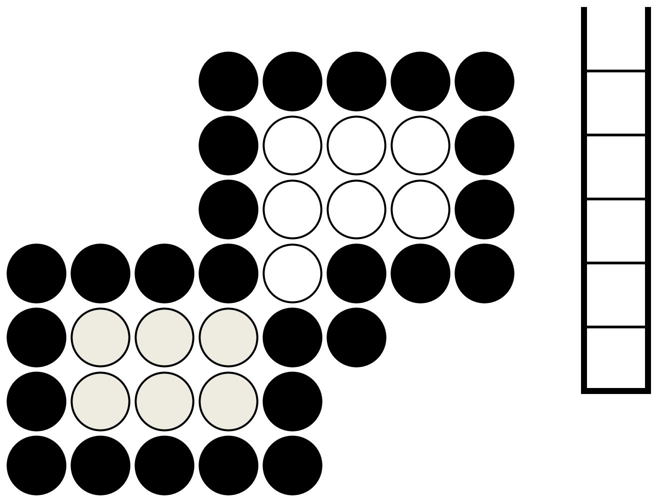
Boundary Fill Algorithm

4-connected (Example)



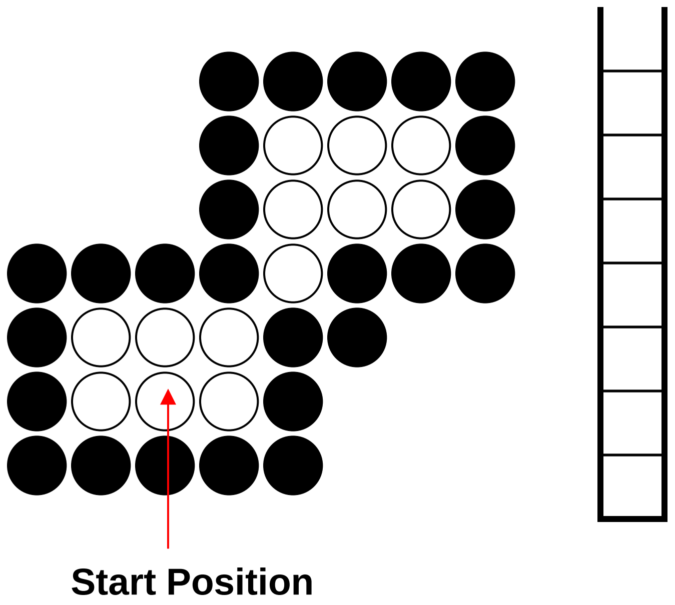
Boundary Fill Algorithm

4-connected (Example)



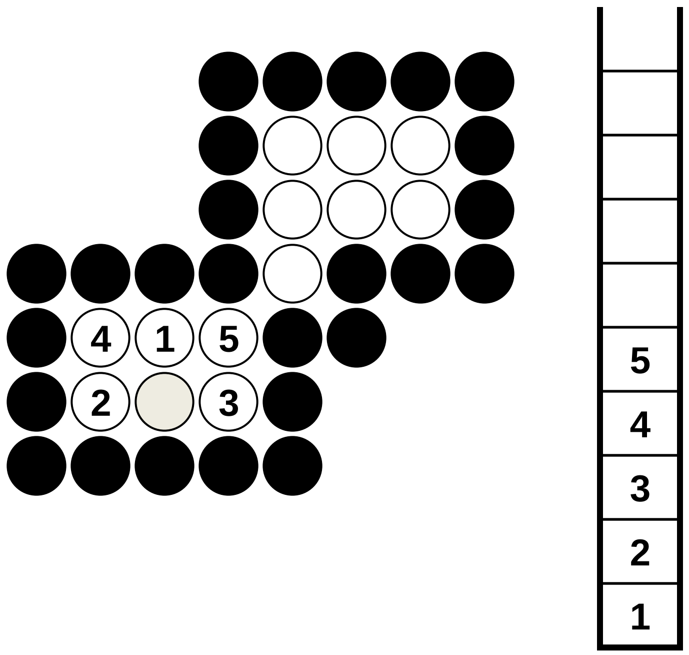
Boundary Fill Algorithm

8-connected (Example)



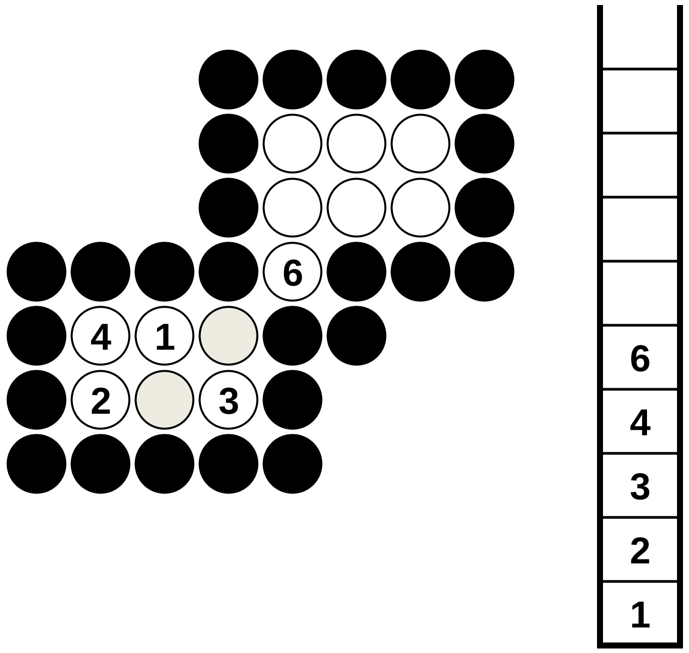
Boundary Fill Algorithm

8-connected (Example)



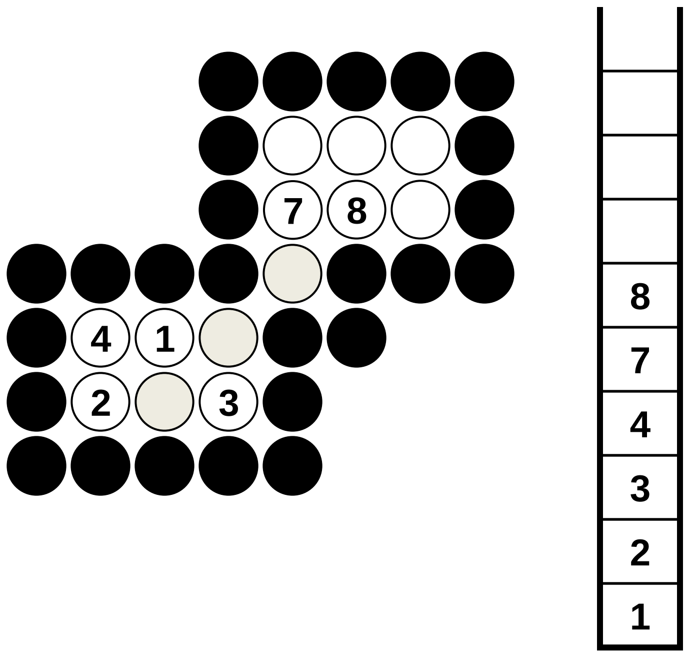
Boundary Fill Algorithm

8-connected (Example)



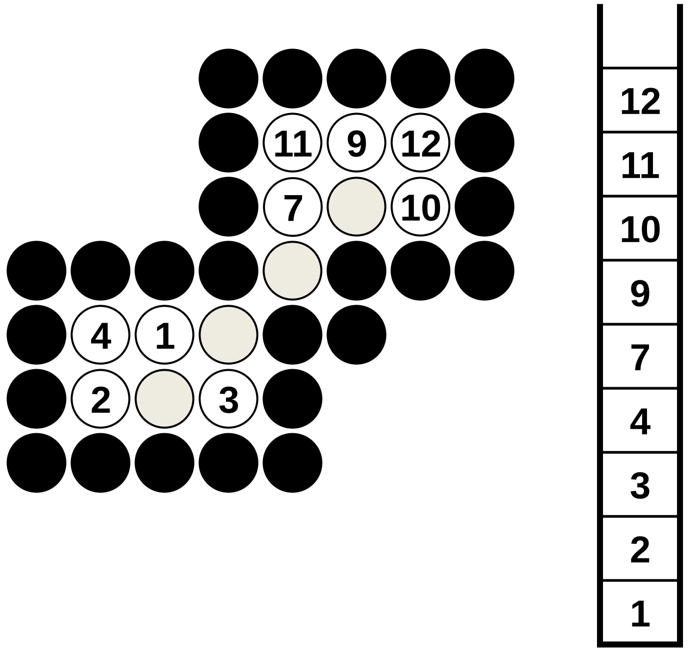
Boundary Fill Algorithm

8-connected (Example)



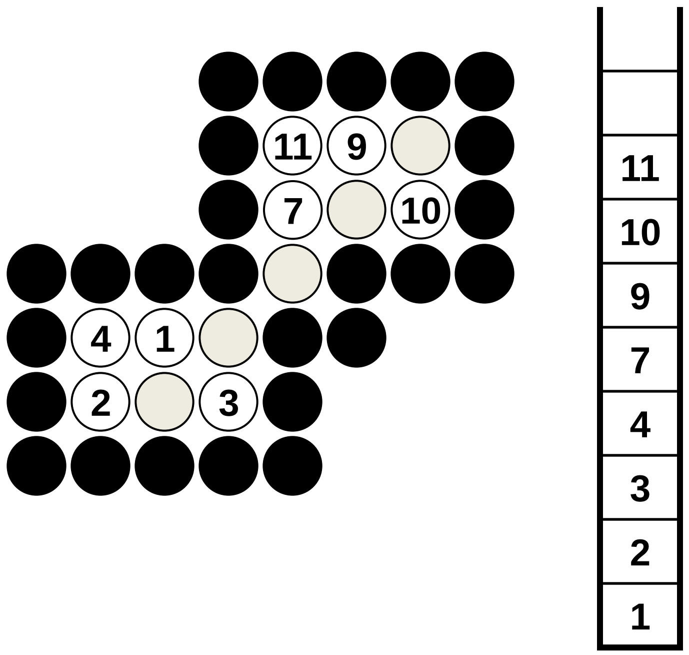
Boundary Fill Algorithm

8-connected (Example)



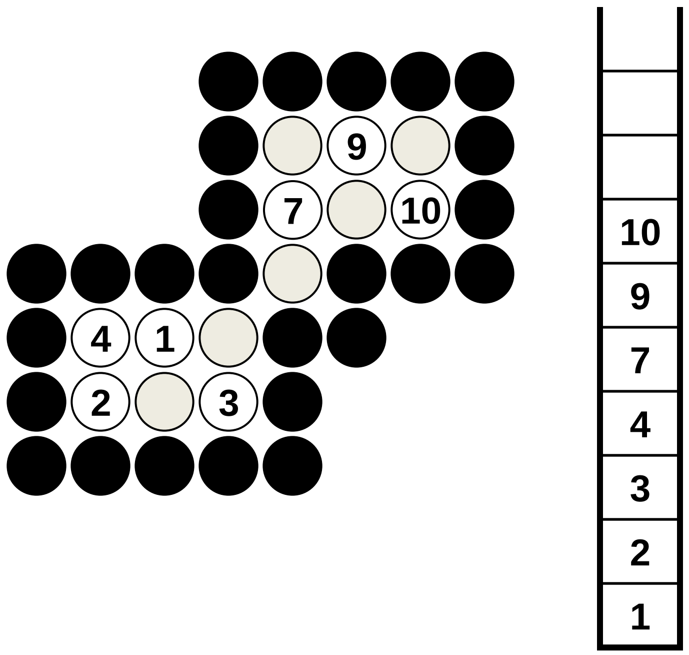
Boundary Fill Algorithm

8-connected (Example)



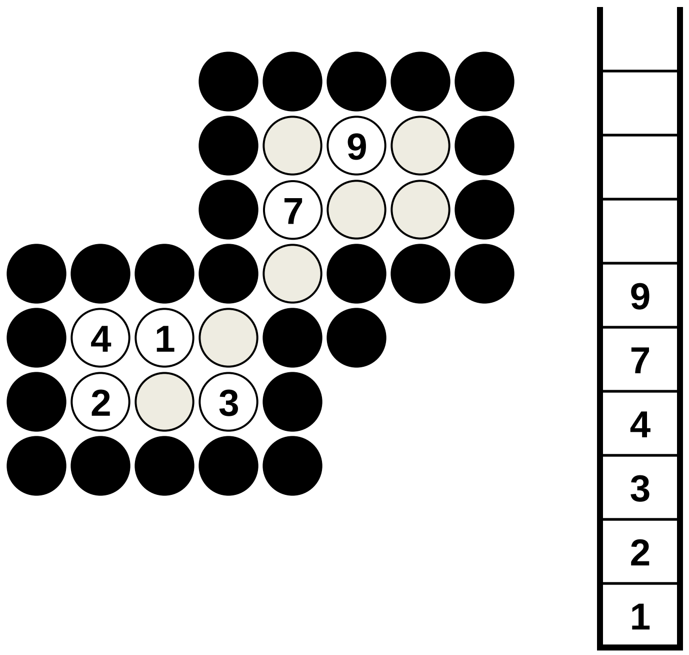
Boundary Fill Algorithm

8-connected (Example)



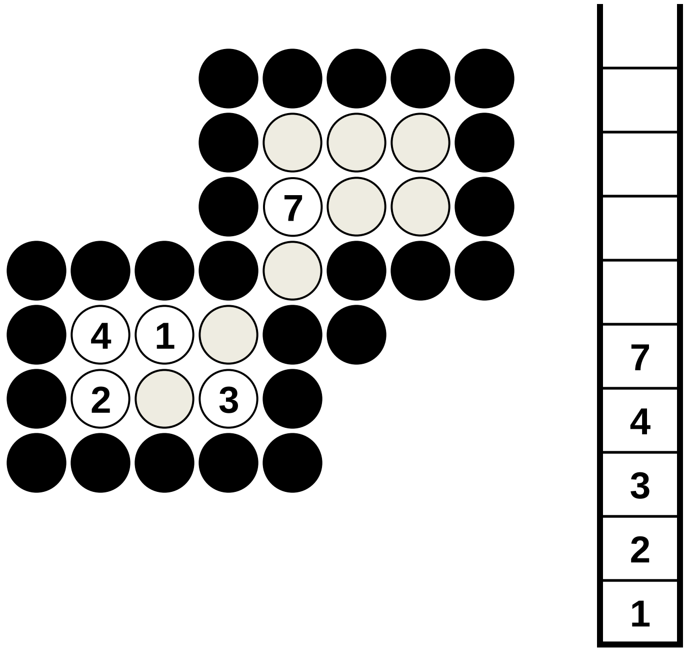
Boundary Fill Algorithm

8-connected (Example)



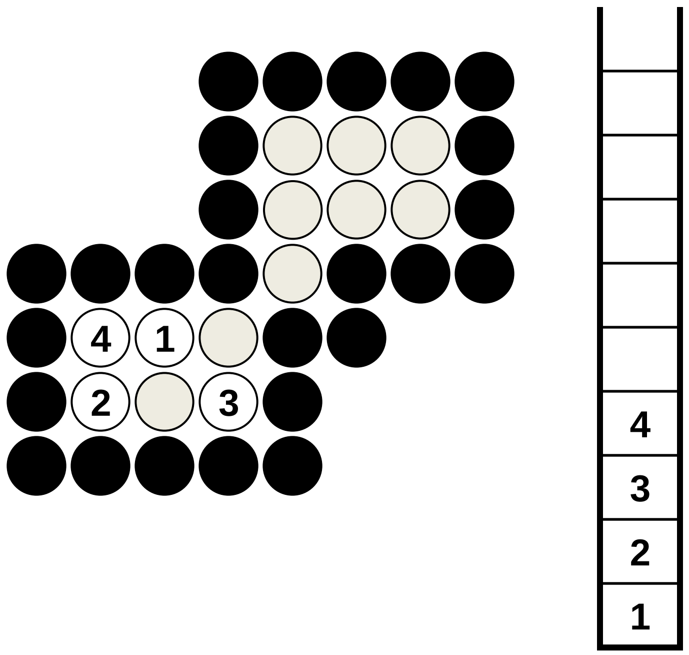
Boundary Fill Algorithm

8-connected (Example)



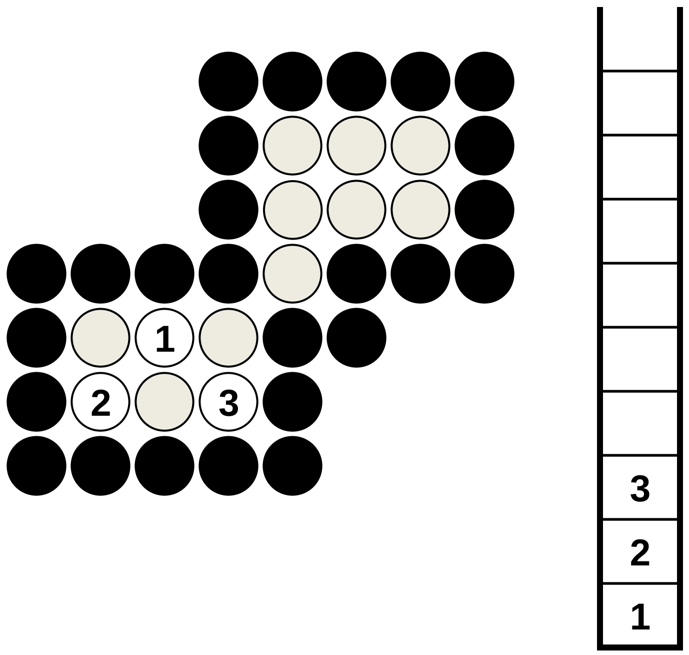
Boundary Fill Algorithm

8-connected (Example)



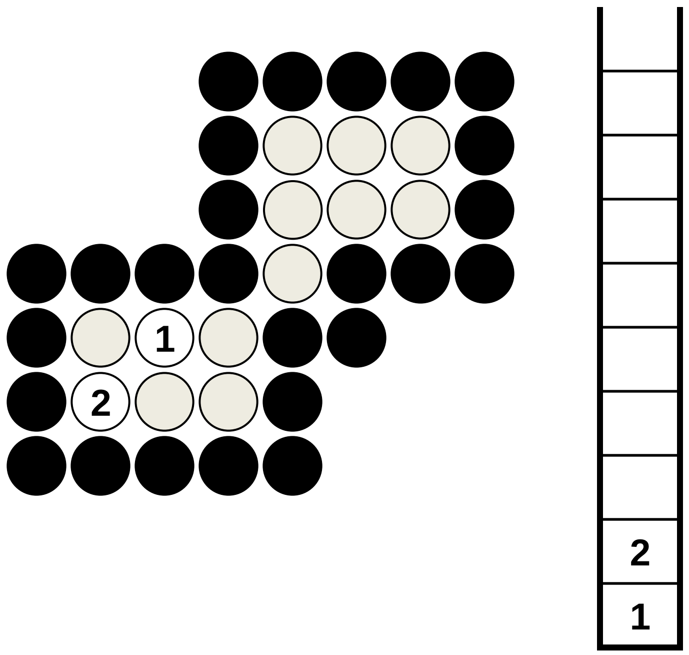
Boundary Fill Algorithm

8-connected (Example)



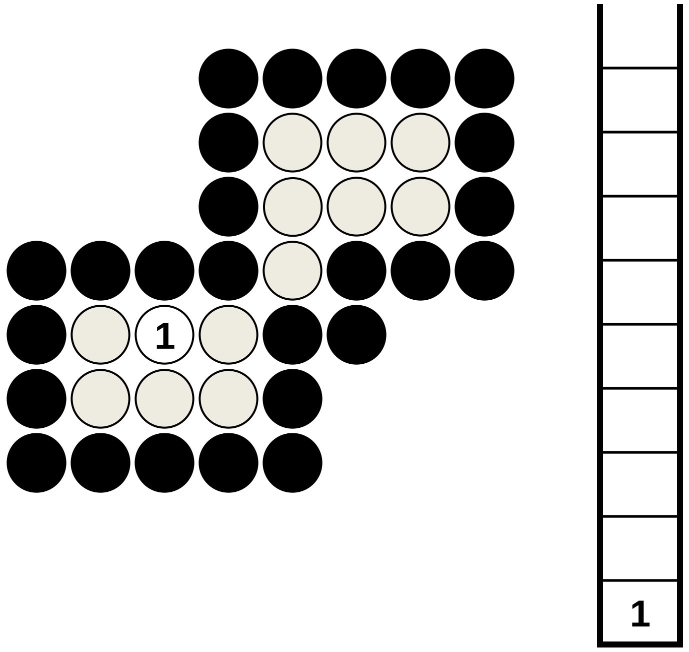
Boundary Fill Algorithm

8-connected (Example)



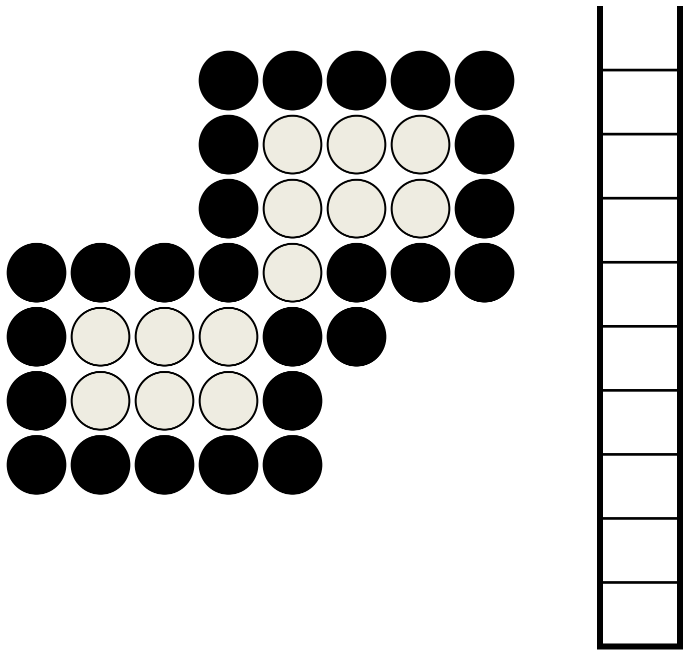
Boundary Fill Algorithm

8-connected (Example)



Boundary Fill Algorithm

8-connected (Example)



Boundary Fill Algorithm

Since the previous procedure requires considerable stacking of neighboring pixels, more **efficient methods** are generally employed.

These methods (**Span Flood-Fill**) **fill horizontal pixel spans across scan lines**, instead of proceeding to 4-connected or 8-connected neighboring pixels.

Then we need only stack a beginning position for each horizontal pixel spans, instead of stacking all unprocessed neighboring positions around the current position.

Span Flood-Fill Algorithm

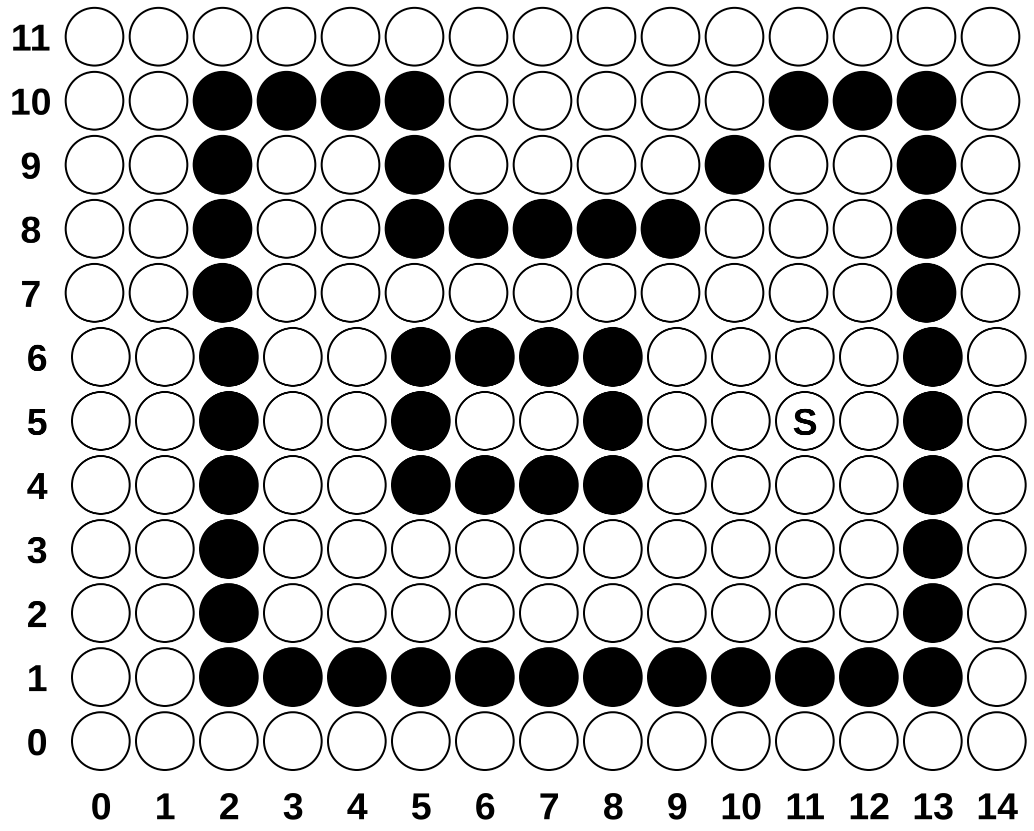
The algorithm is summarized as follows:

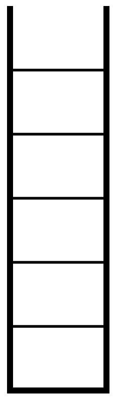
- **Starting** from the initial interior pixel, then fill in the contiguous span of pixels on this starting scan line.

- **Then locate** and **stack** starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the area border color.

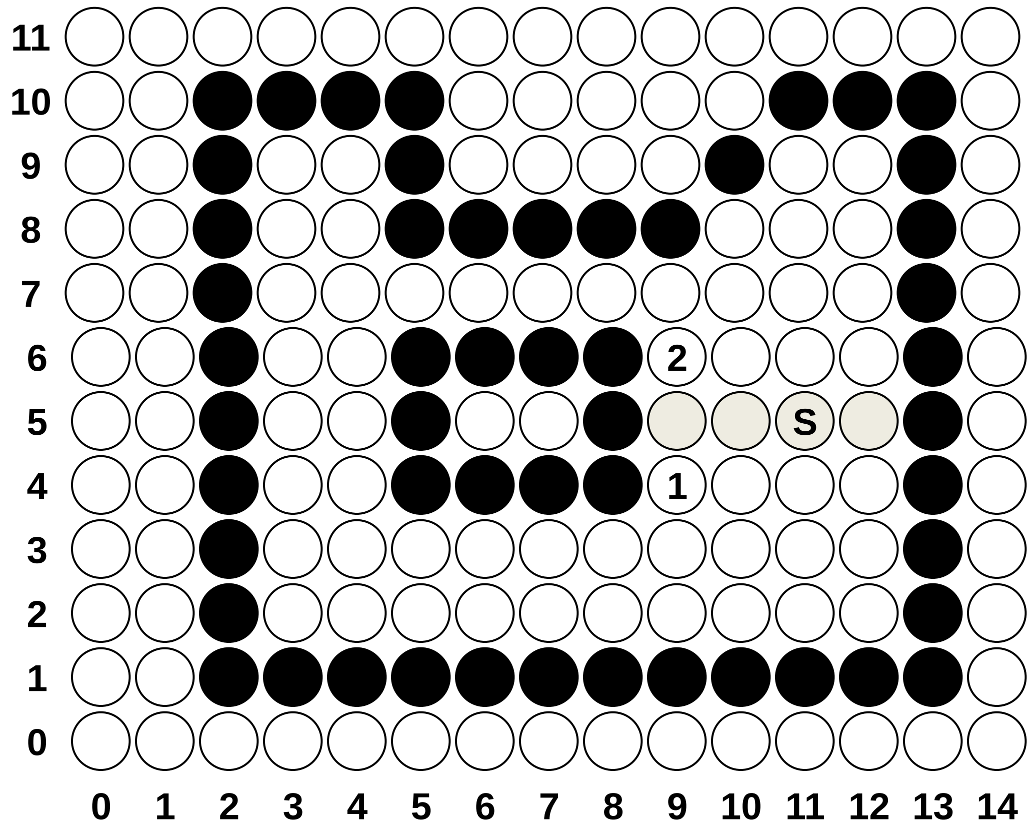
- **At each subsequent step**, **unstack** the next start position and repeat the process.

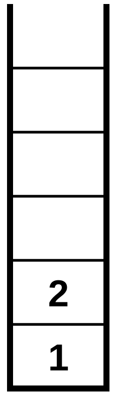
Span Flood-Fill Algorithm (example)



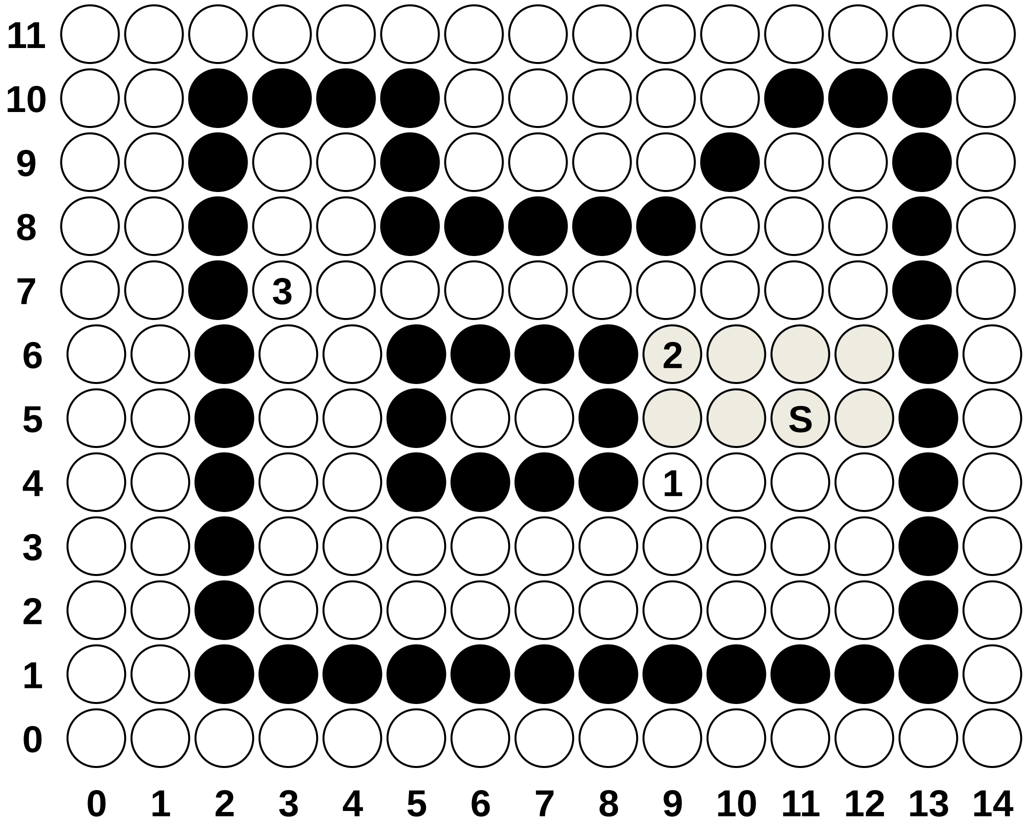


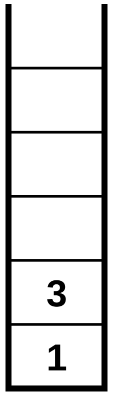
Span Flood-Fill Algorithm (example)



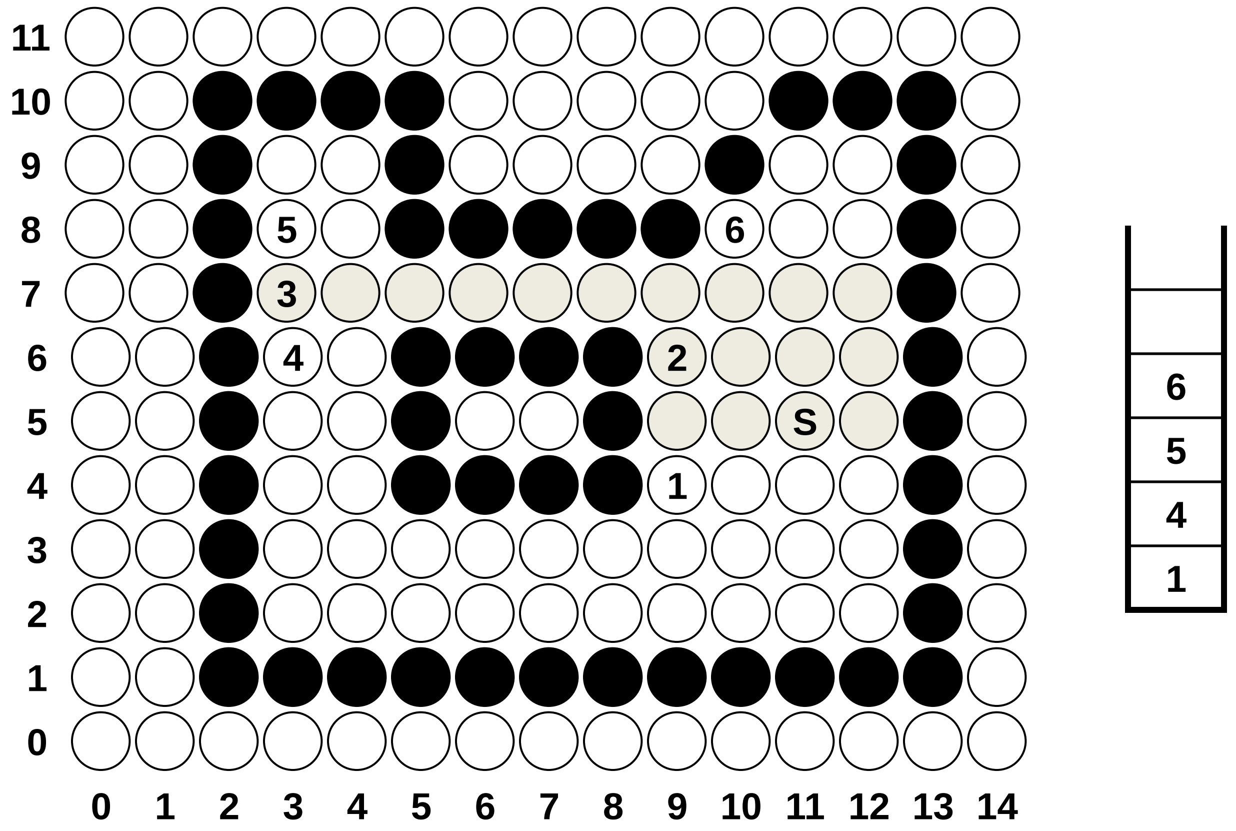


Span Flood-Fill Algorithm (example)

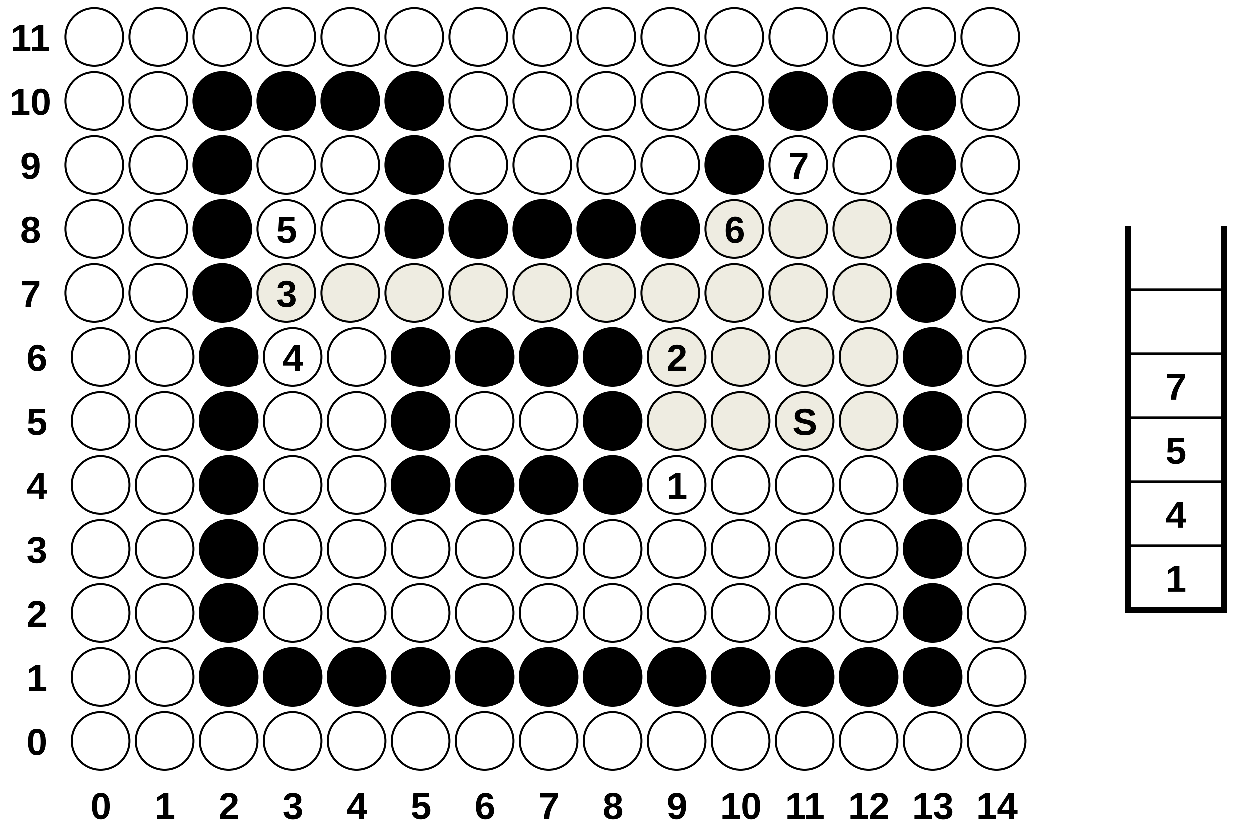




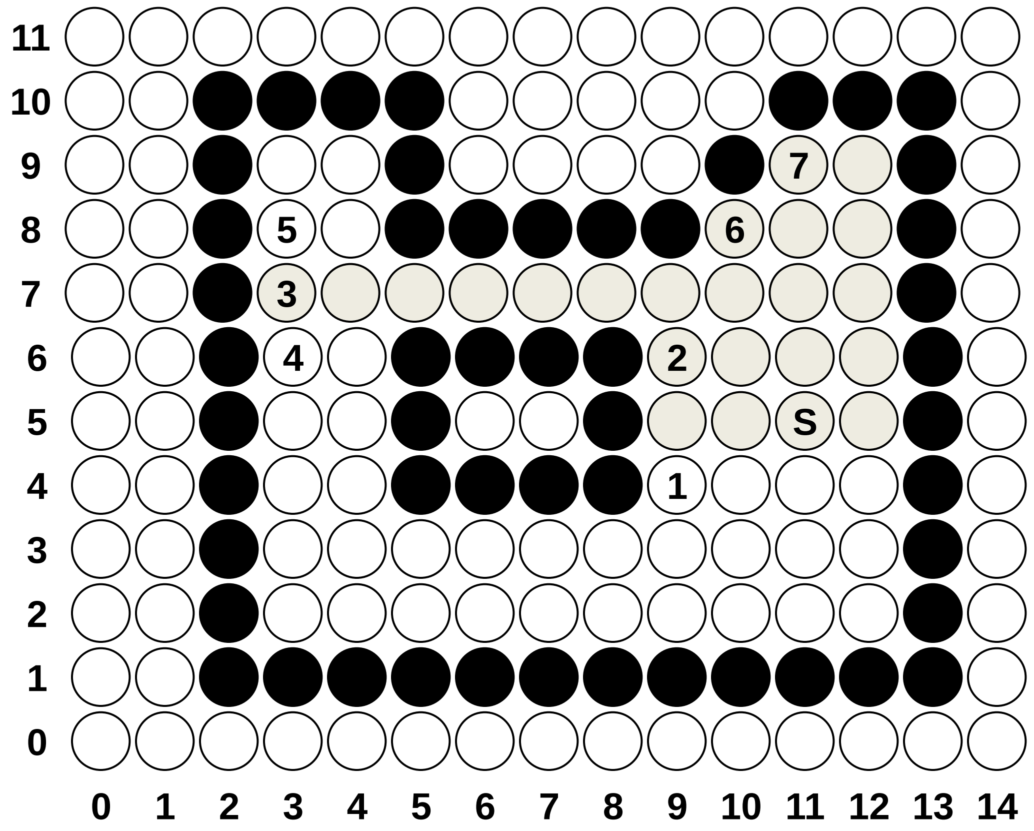
Span Flood-Fill Algorithm (example)

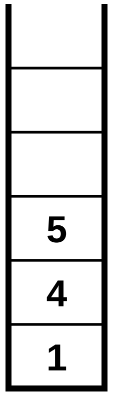


Span Flood-Fill Algorithm (example)

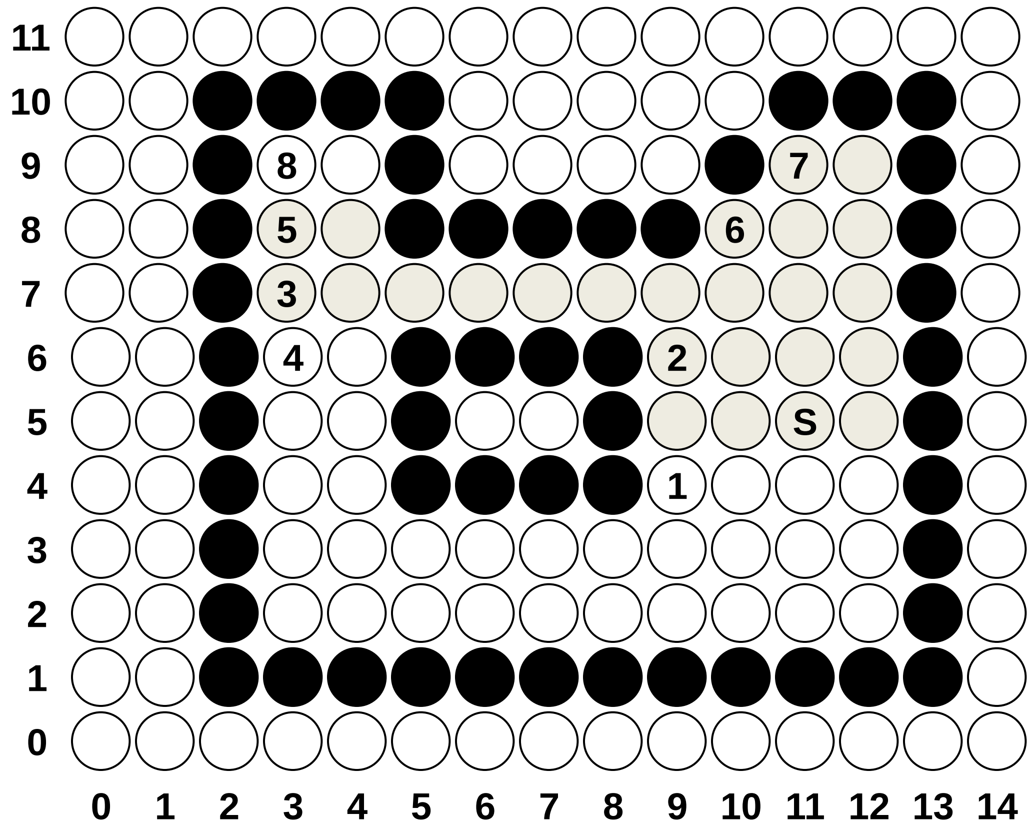


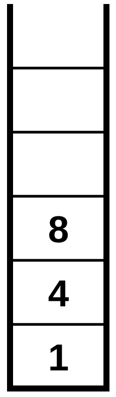
Span Flood-Fill Algorithm (example)



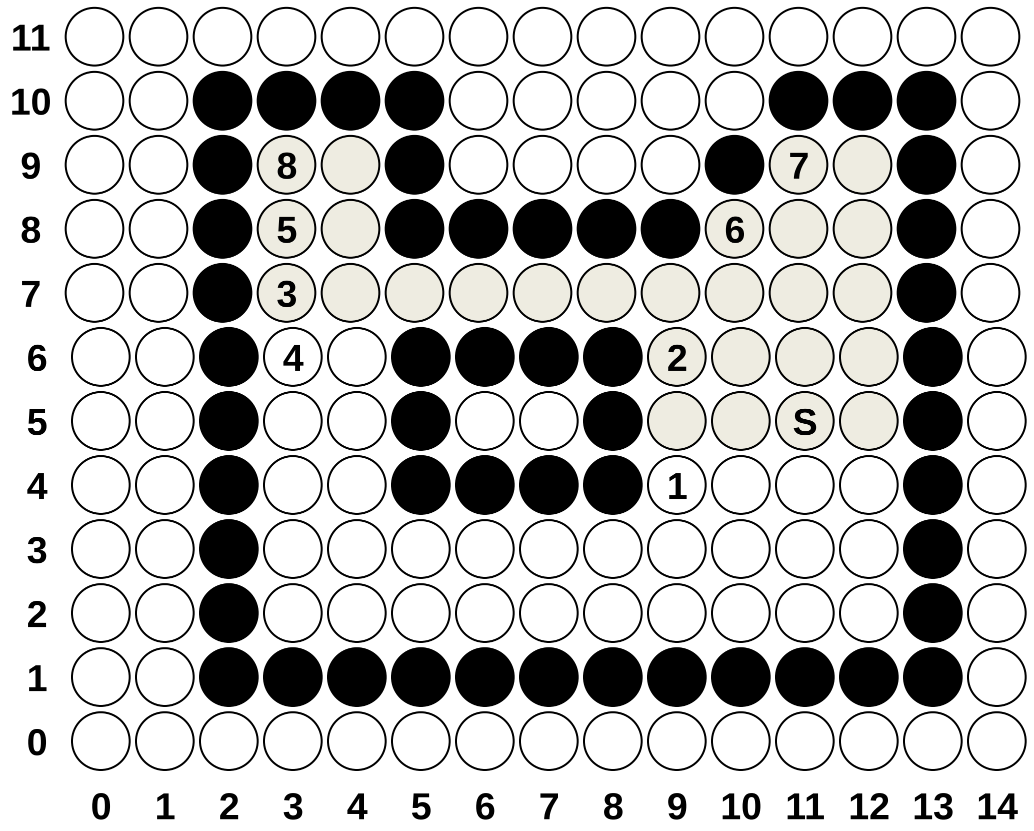


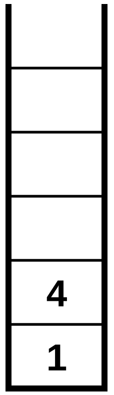
Span Flood-Fill Algorithm (example)



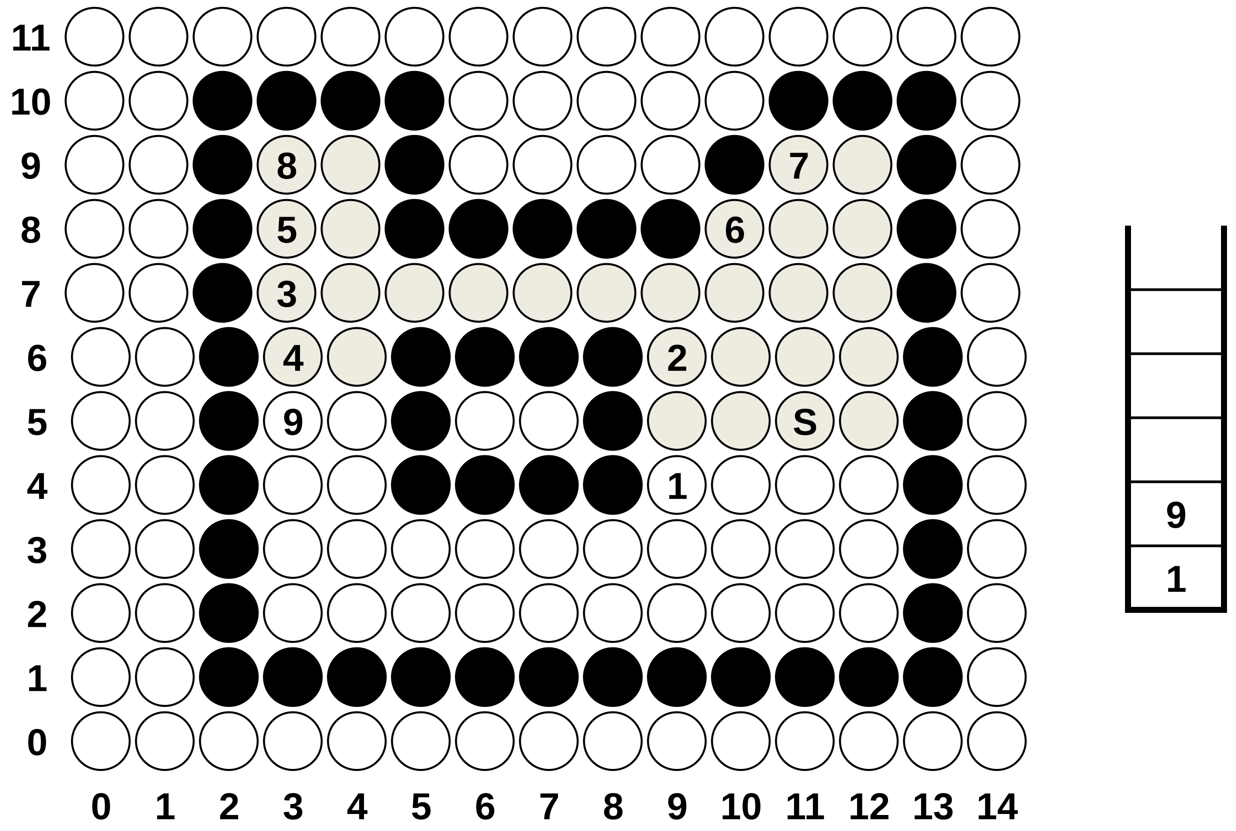


Span Flood-Fill Algorithm (example)

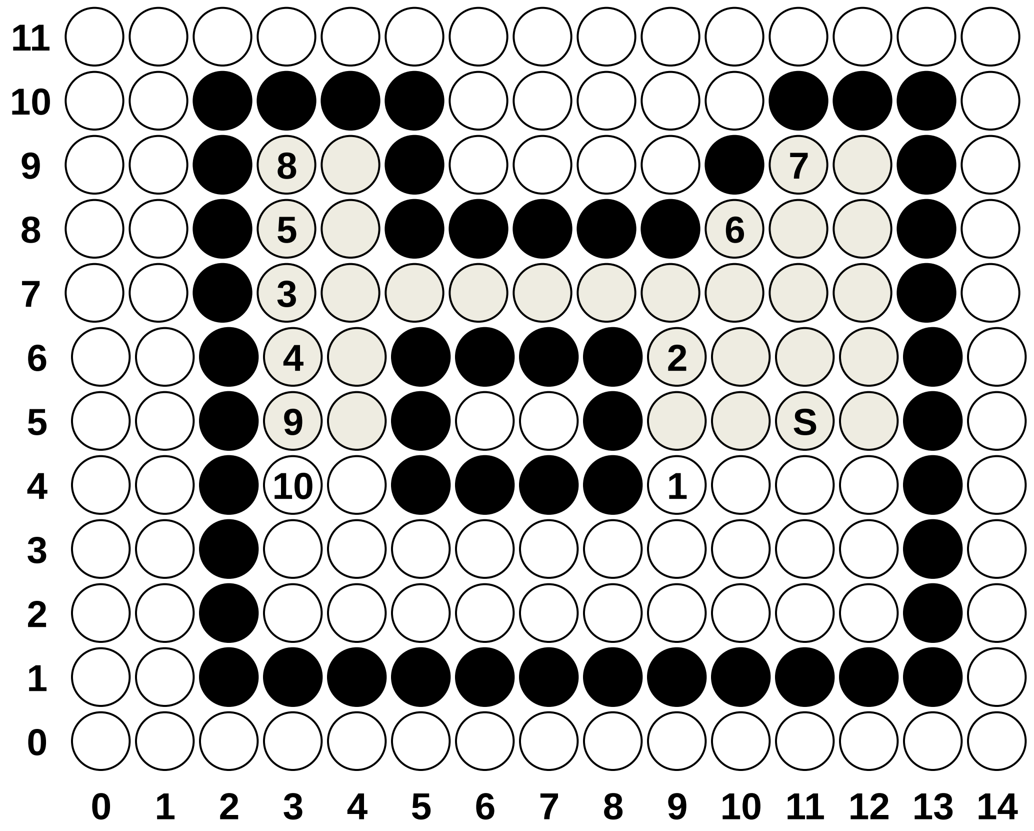


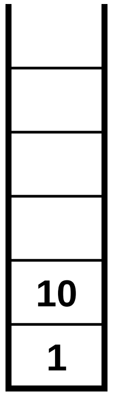


Span Flood-Fill Algorithm (example)

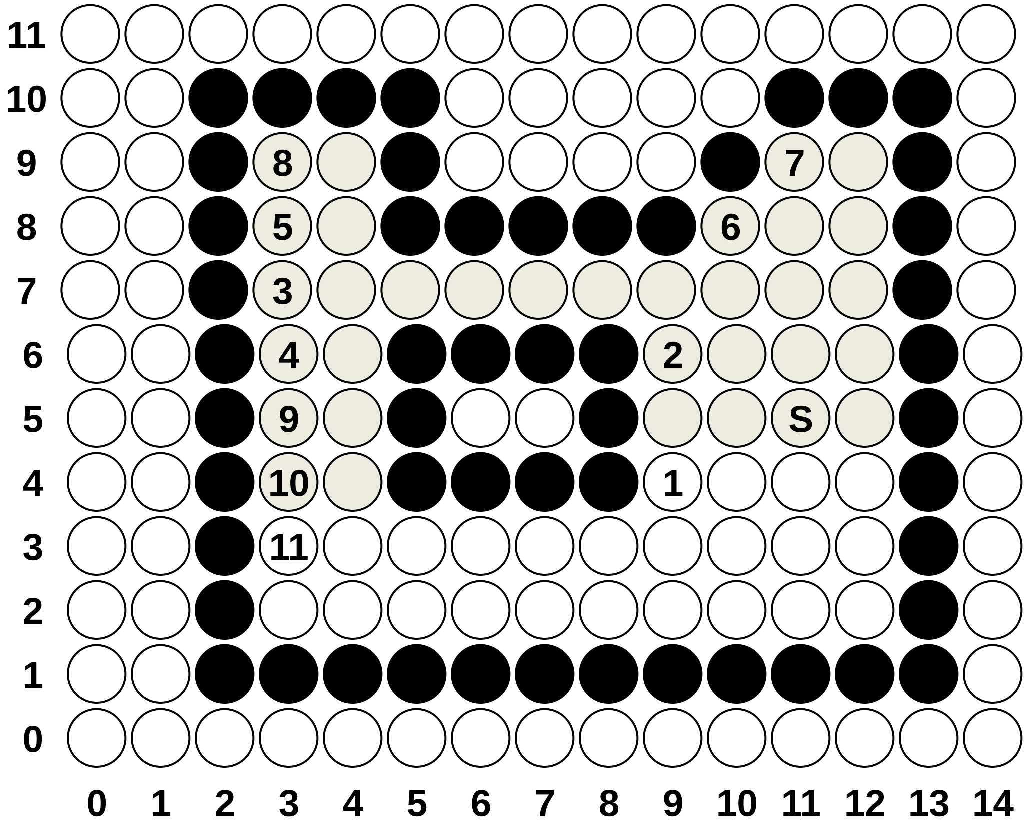


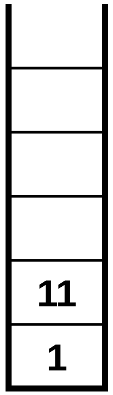
Span Flood-Fill Algorithm (example)



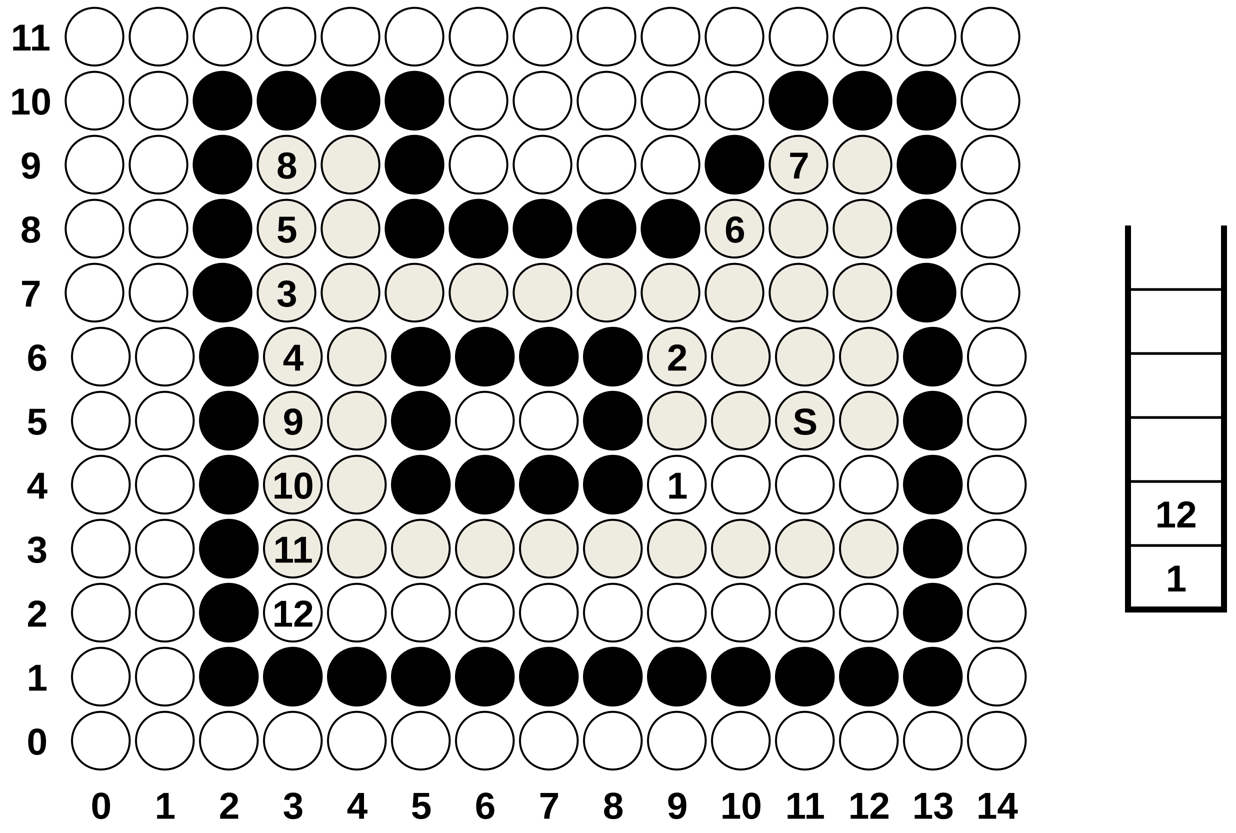


Span Flood-Fill Algorithm (example)

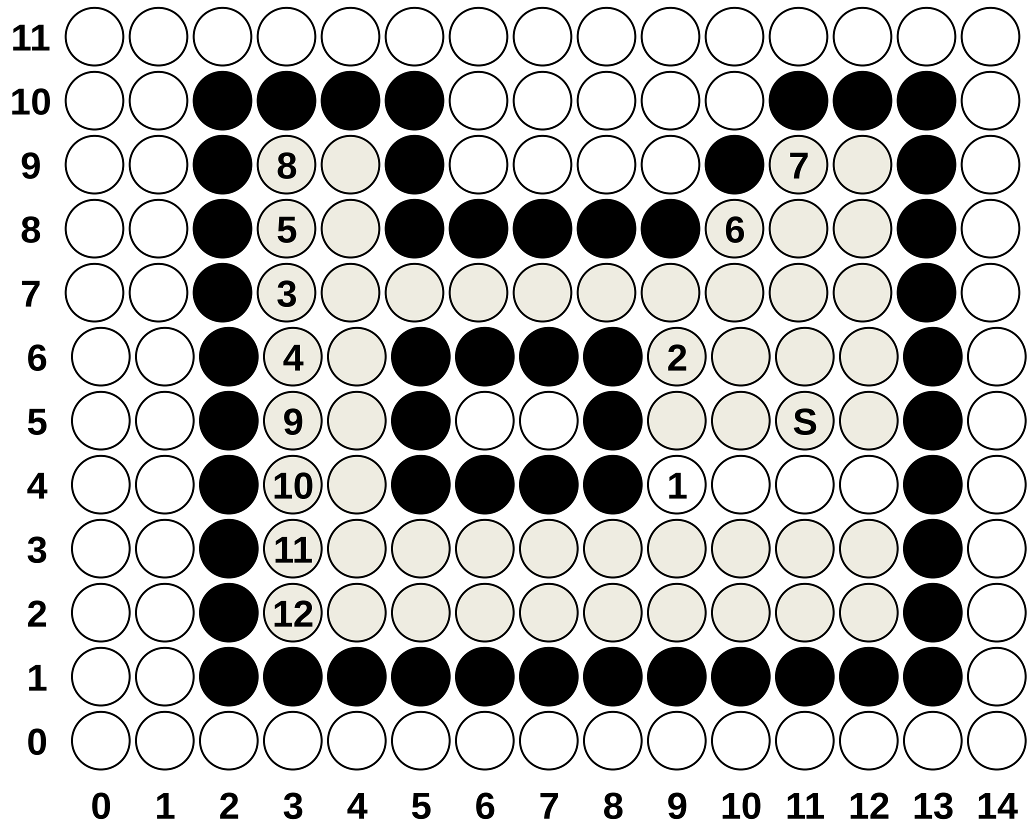


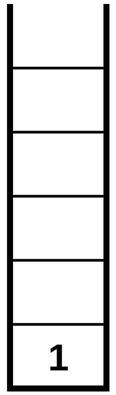


Span Flood-Fill Algorithm (example)

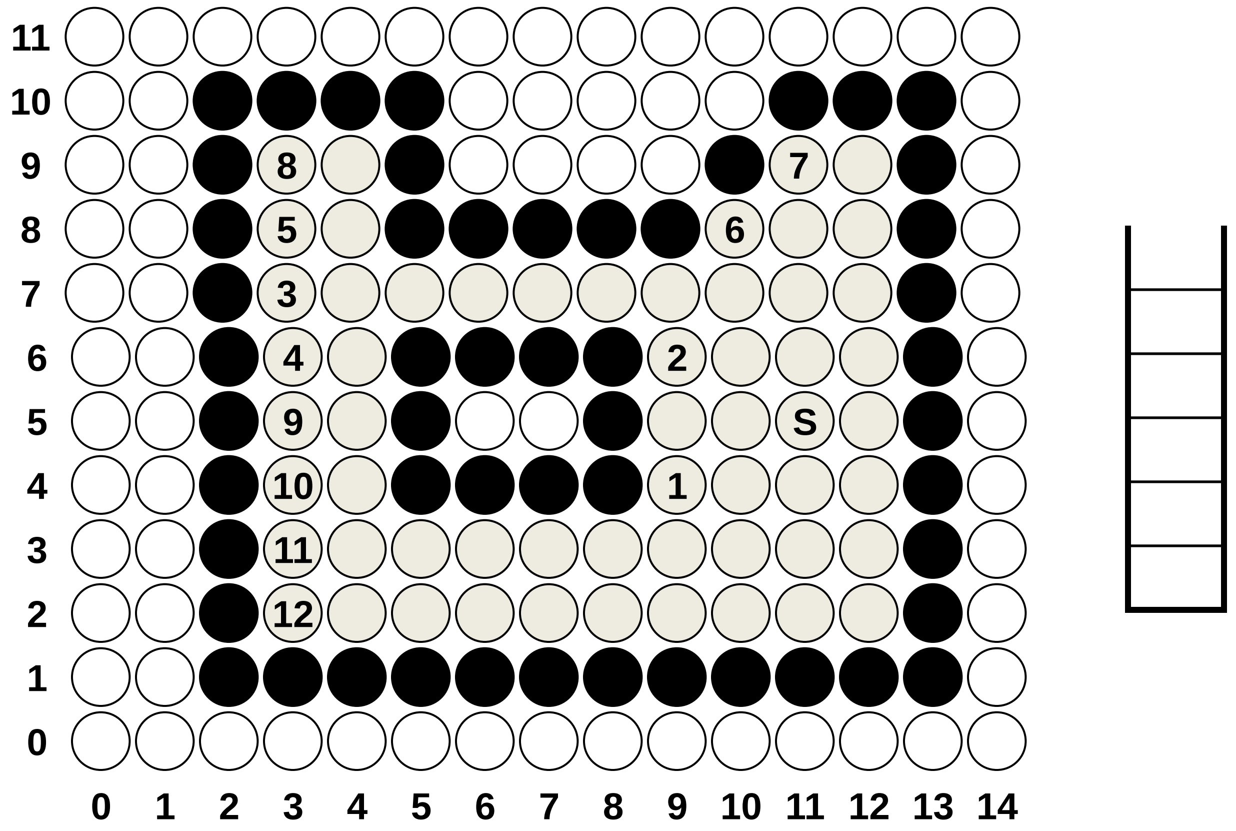


Span Flood-Fill Algorithm (example)





Span Flood-Fill Algorithm (example)

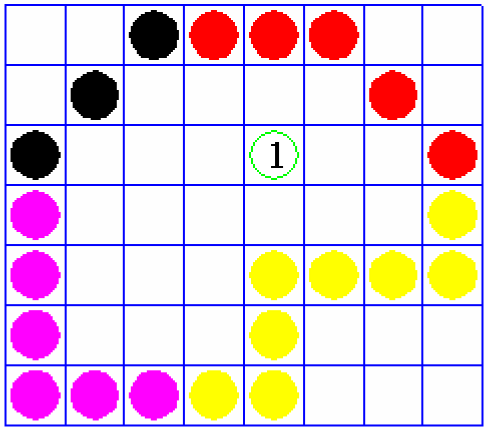


Flood Fill Algorithm

Sometimes we want to fill in (recolor) an area that is not defined within a single color boundary.

We paint such areas by replacing a specified interior color instead of searching for a boundary color value.

This approach is called a **flood-fill algorithm**.



Flood Fill Algorithm

We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color.

If the area has **more than one** interior color, we can first **reassign pixel values** so that all interior pixels have the same color.

Using either **4-connected** or **8-connected** approach, we then step through pixel positions until all interior pixels have been repainted.

Flood-Fill Algorithm

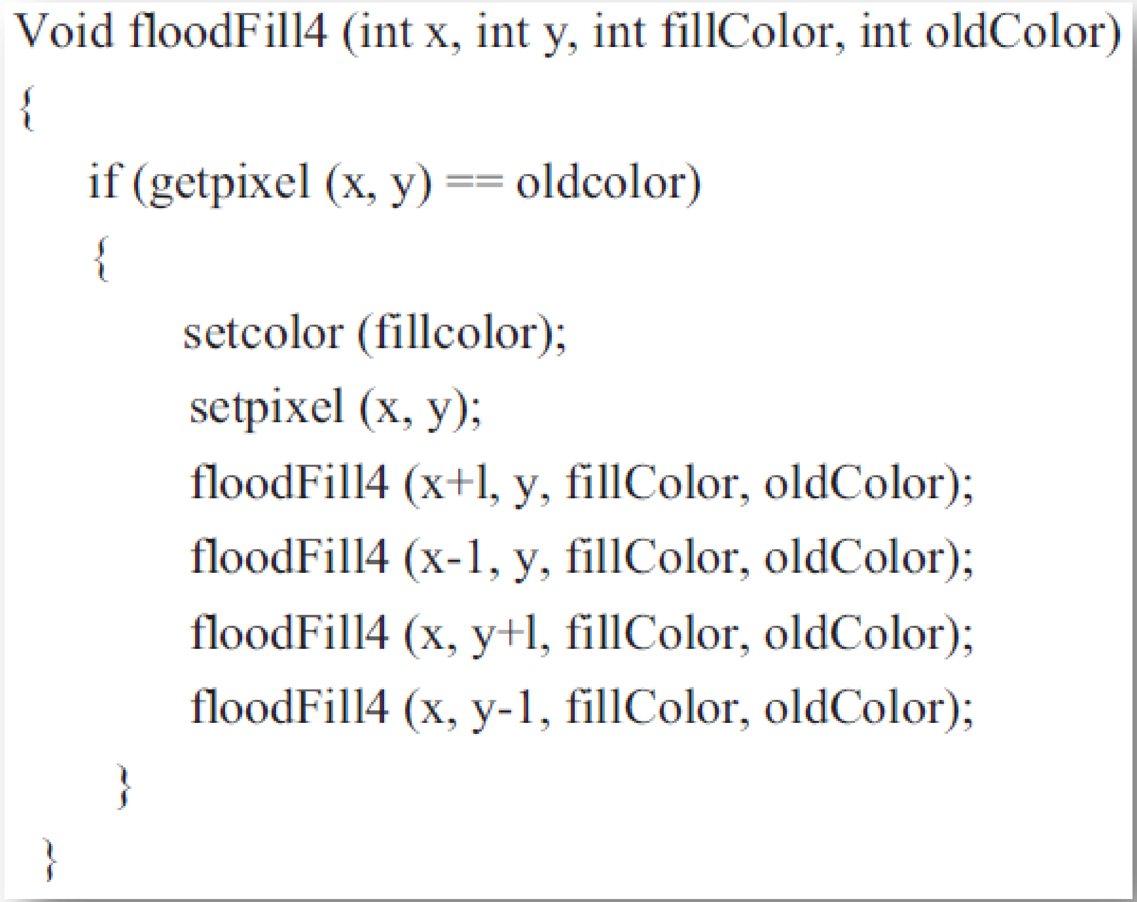
- Begin with a seed(starting pixel) inside the region.
- It checks to see if the pixel has the region’s original color.
- If the answer is yes, it fills the pixel with a new color and uses each of the pixel’s neighbors as anew seed in a recursive call.
- If answer is no , it returns to the caller.

Flood-Fill Algorithm

- We start from a specified interior point (*x, y) and reassign all pixel values that are currently* set to a given interior color with the desired fill color.
- If the area we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color.
- Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted.

Flood-Fill Algorithm

- The following procedure flood fills a 4-connected region recursively, starting from the input position.



```
Void floodFill4 (int x, int y, int fillColor, int oldColor)
{
    if (getpixel (x, y) == oldcolor)
    {
         setcolor (fillcolor);
         setpixel (x, y);
         floodFill4 (x+l, y, fillColor, oldColor);
         floodFill4 (x-1, y, fillColor, oldColor);
         floodFill4 (x, y+l, fillColor, oldColor);
         floodFill4 (x, y-1, fillColor, oldColor);
     }
 }
```