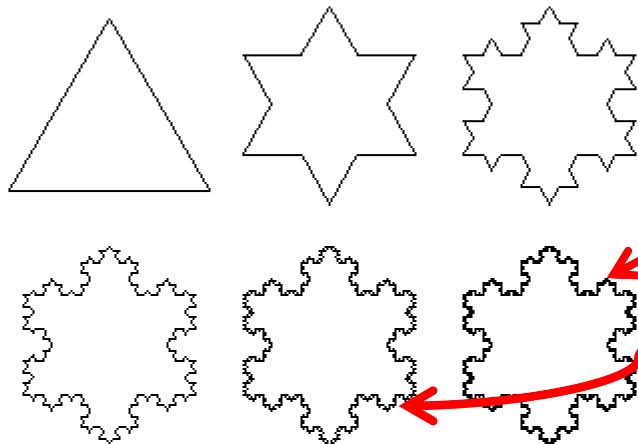
18. Recursion

Recursive Partitioning
Random Mondrian
The Mechanics of Recursion Using n!
Back to MergeSort

What is Recursion?

A function is recursive if it calls itself.

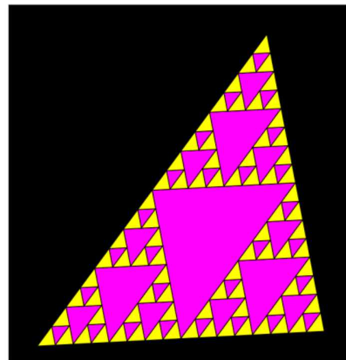
A pattern is recursive if it is defined in terms of itself.



I can tell you what this
is in terms of what that
is.

Recursive Graphics

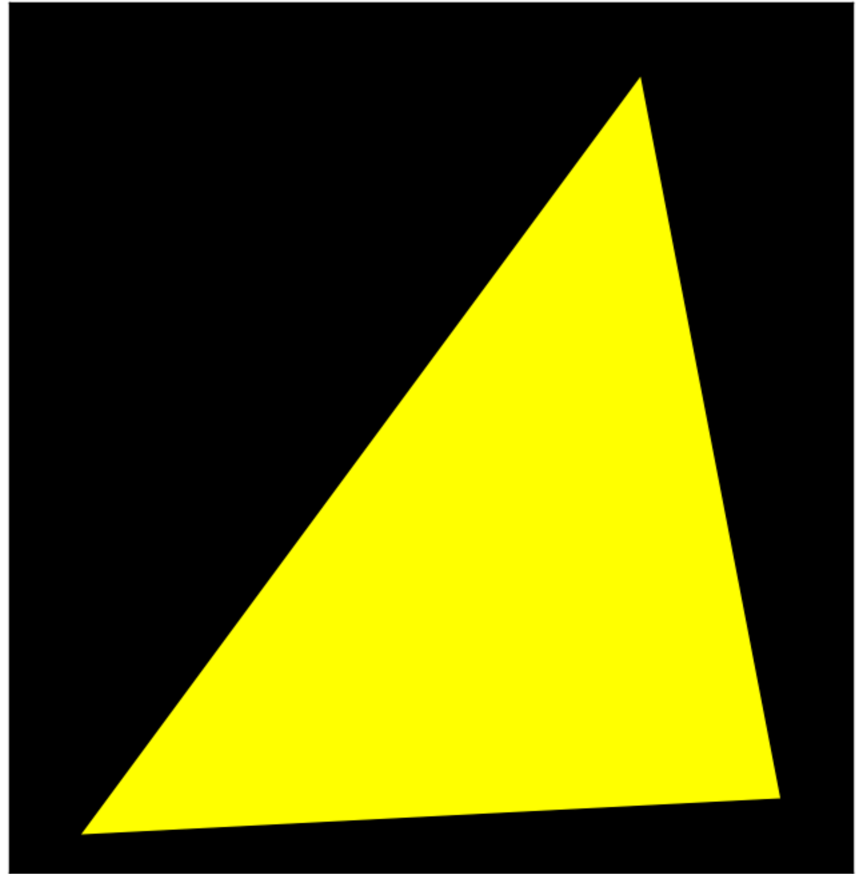
We will develop a graphics procedure that draws this:



The procedure will call itself.

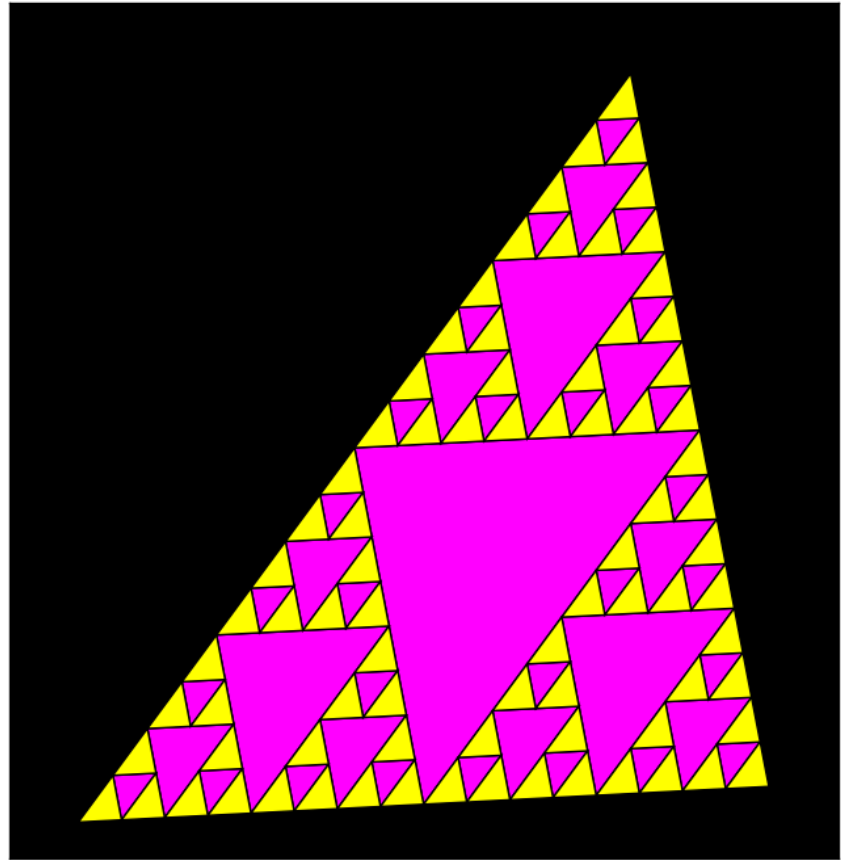
Partitioning a Triangle

Given This:

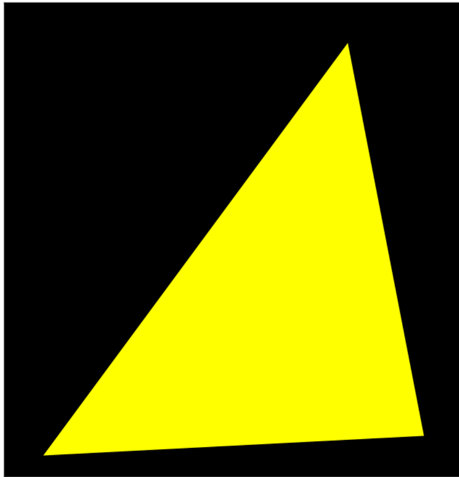


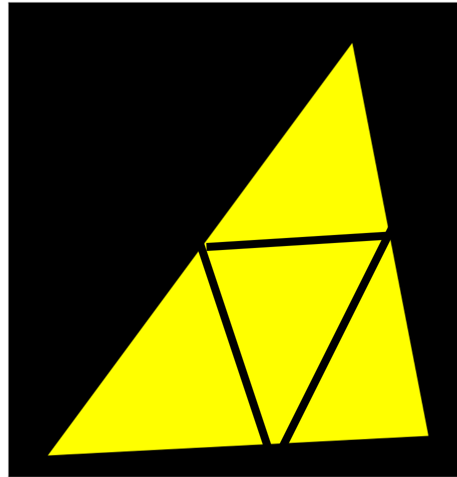
Partitioning a Triangle

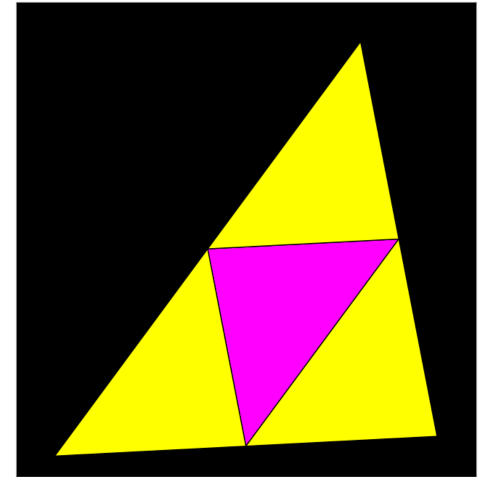
Draw This:



Requires Repetition of This…





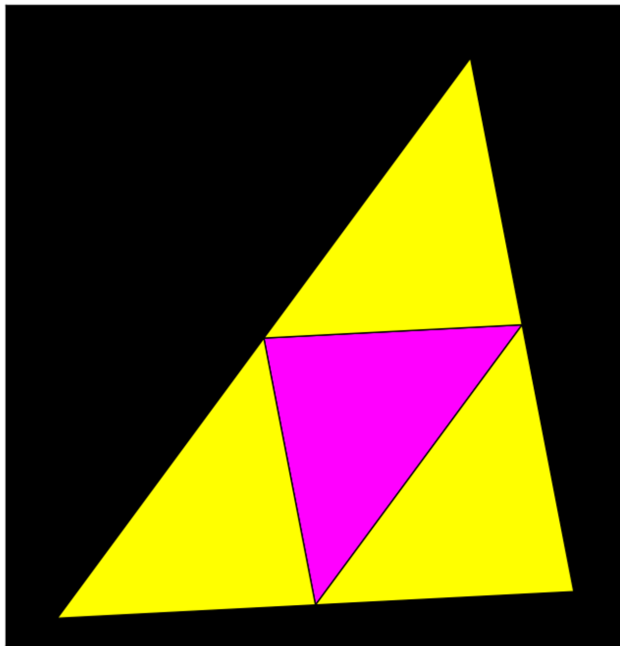


Given a yellow triangle

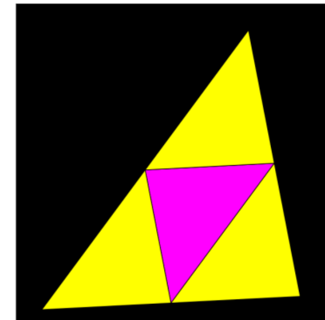
Define the inner triangle and the 3 corner triangles

Color the inner triangle and repeat the process on the 3 corner triangles

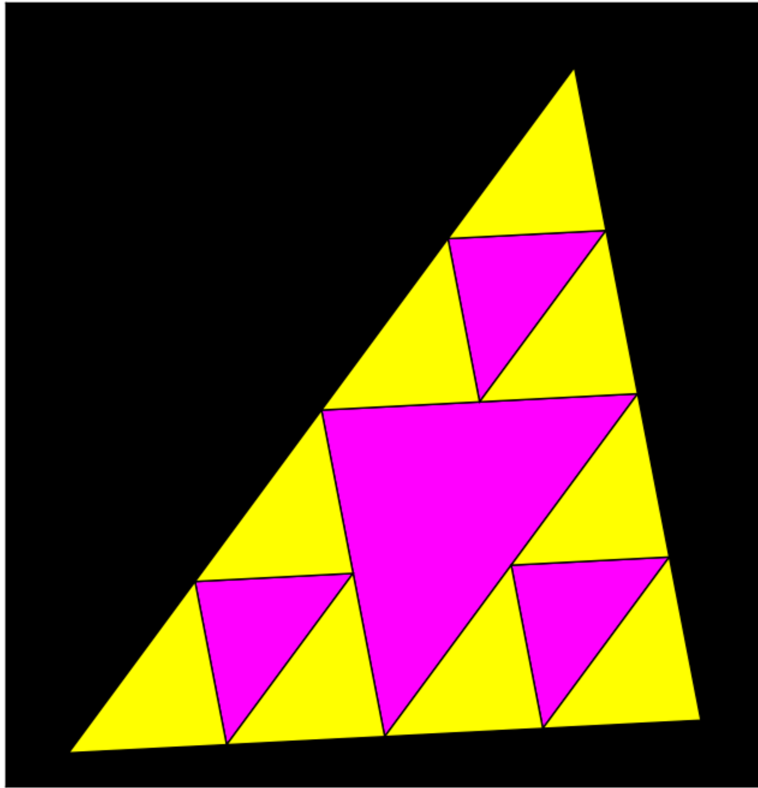
"Repeat the Process"



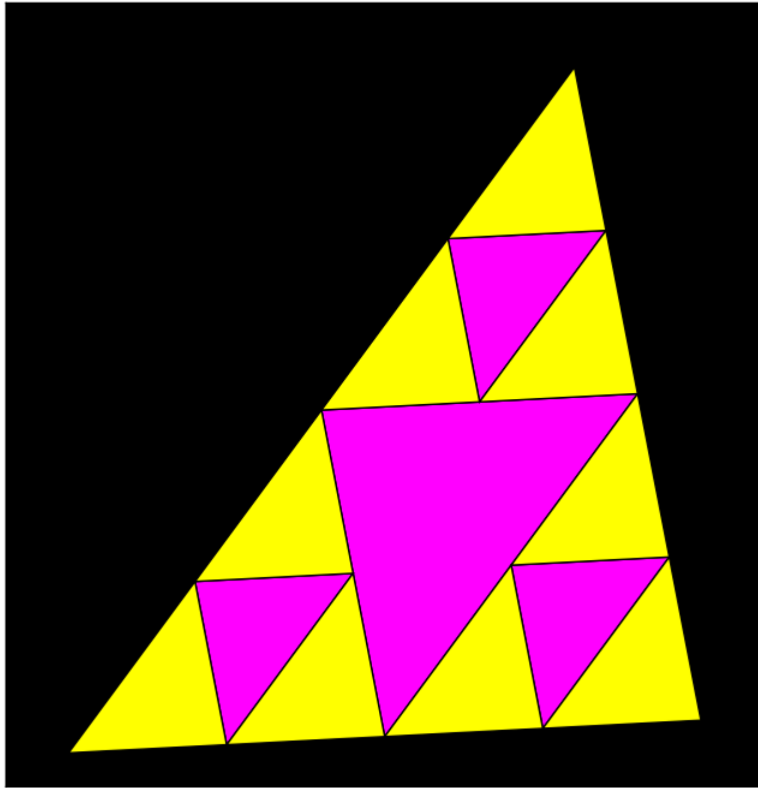
Visit every yellow triangle and replace it with



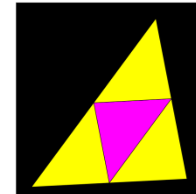
Gives Us This...



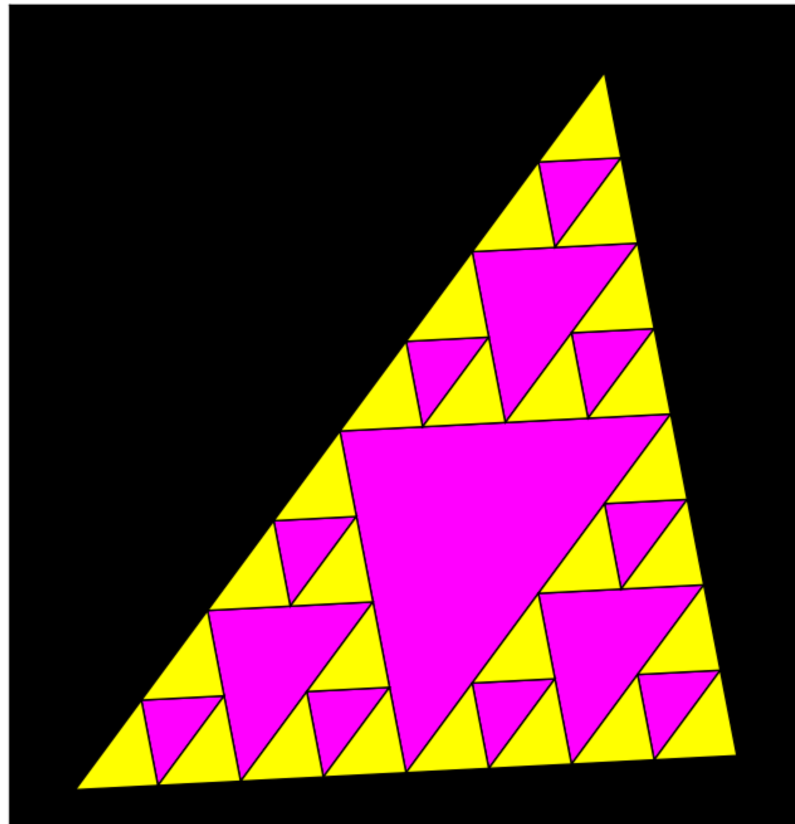
“Repeat the Process”



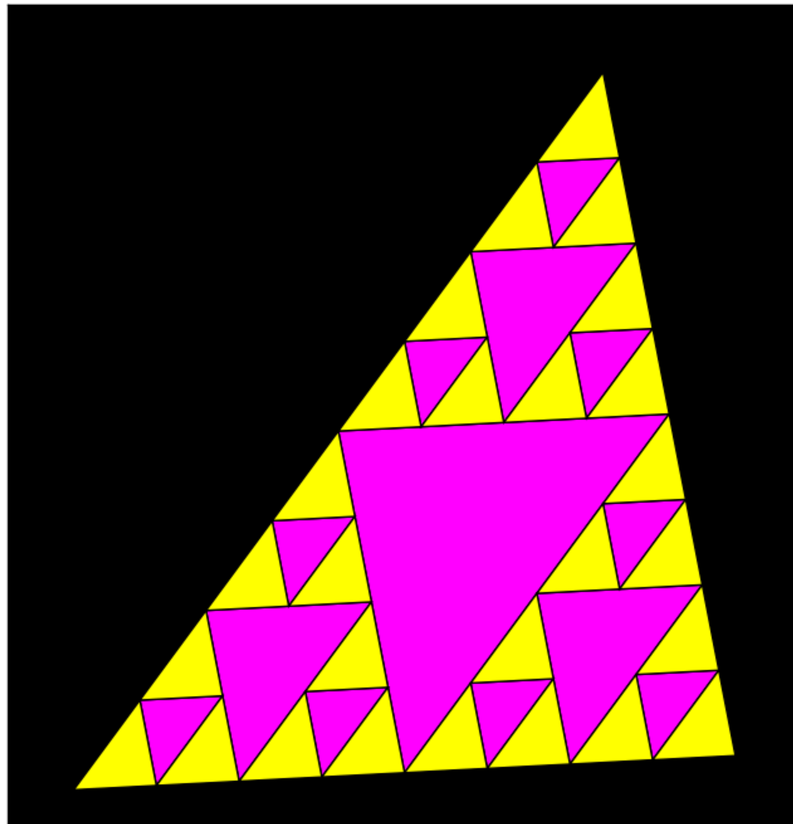
Visit every yellow triangle and replace it with



Gives Us This…

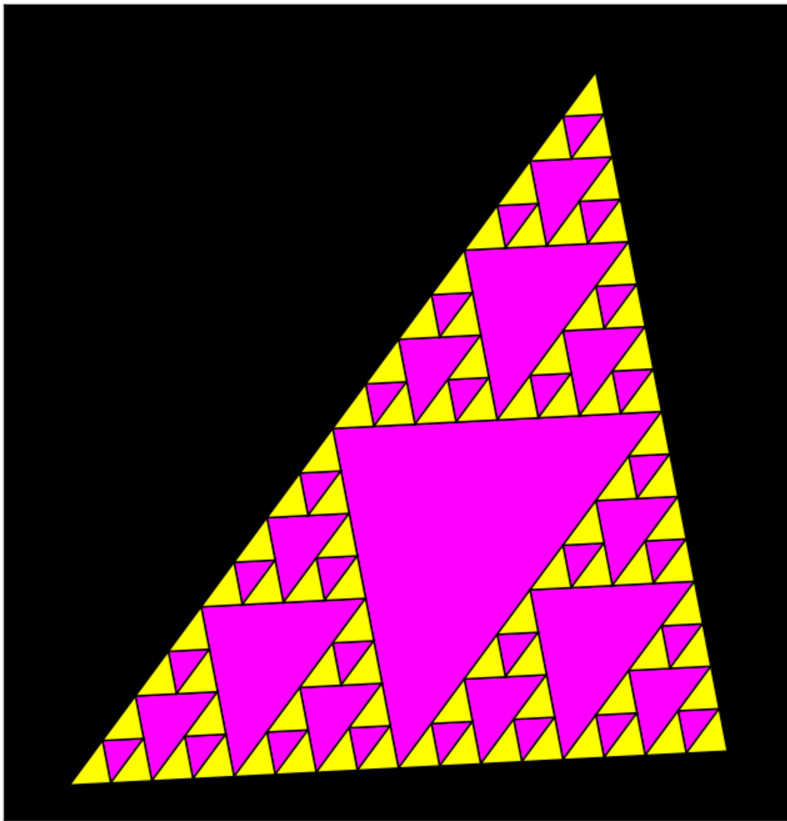


"Repeat the Process"



Visit every yellow triangle and replace it with

Gives Us This



Etc.

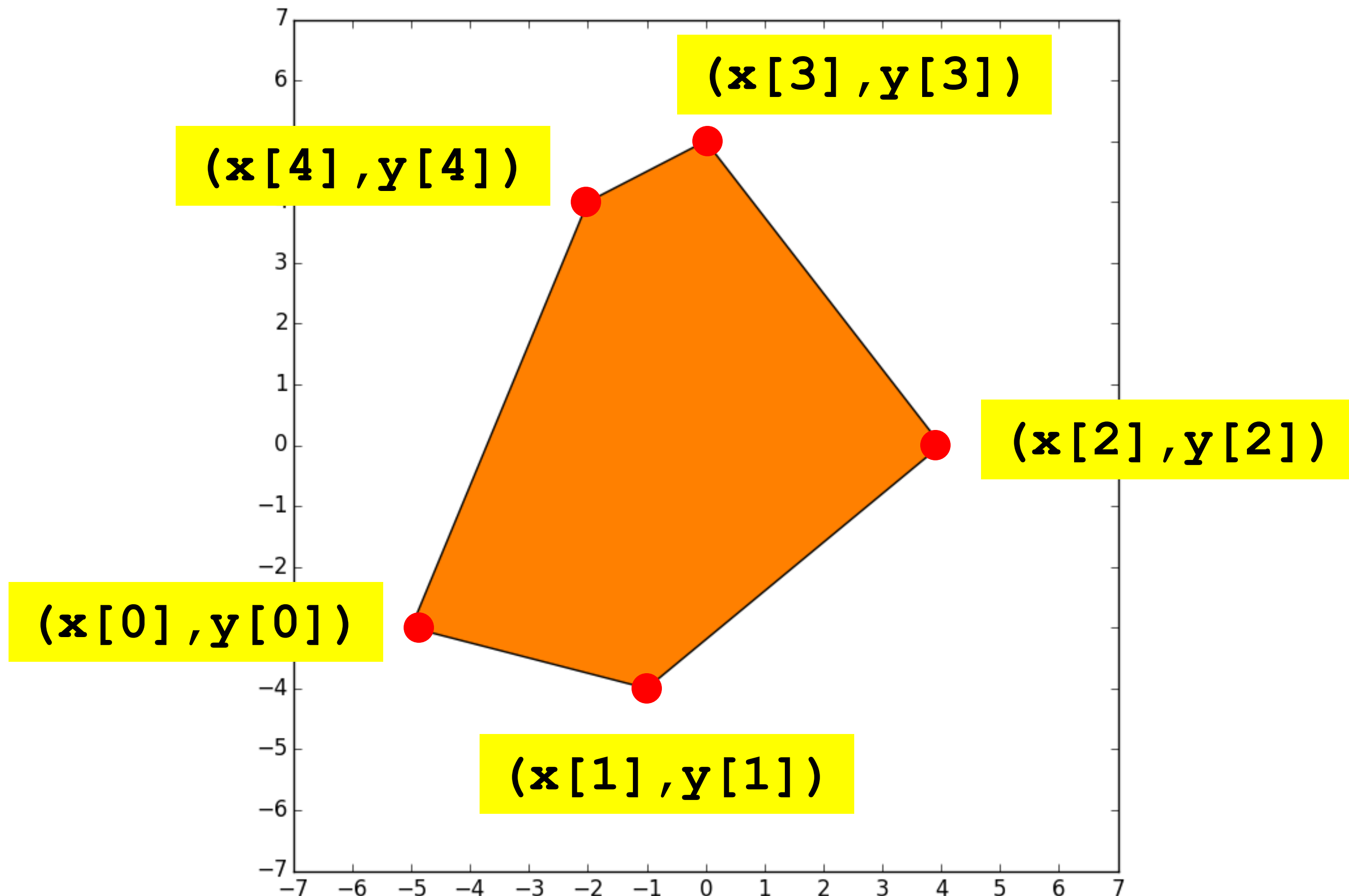
To Pull This Off
We Will Need Some
Triangle Drawing Tools

We Augment simpleGraphicsE -- Again

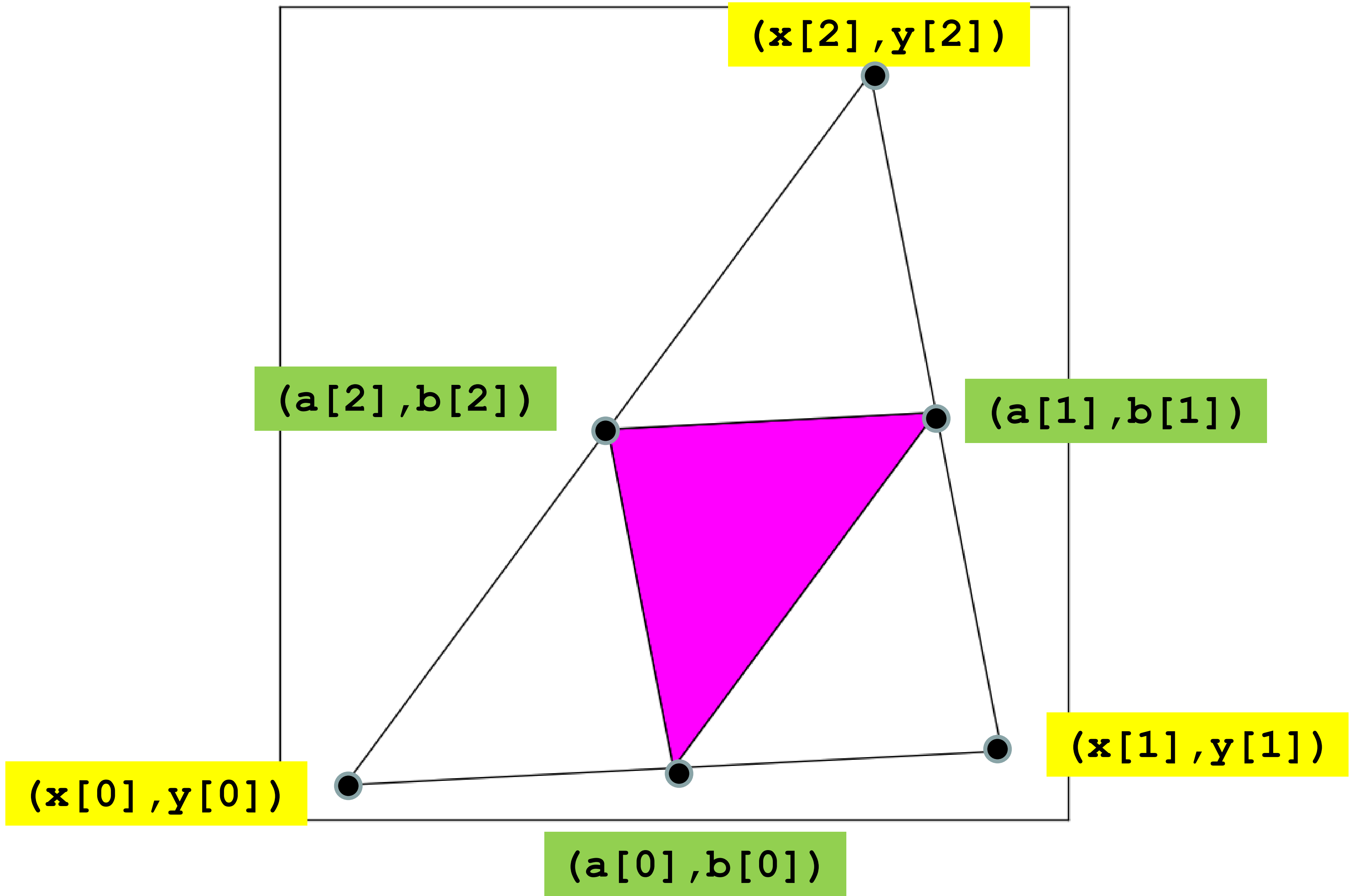
```
def DrawPoly(x,y,color=None,stroke=1):
    """ Draws a polygon whose vertices
    are specified by float lists x and y.
    """
```

```
x = [-5,-1,4,0,-2]
y = [-3,-4,0,5, 4]
DrawPoly(x,y,color=ORANGE)
```

Drawing a Polygon



Drawing the Inner Triangle

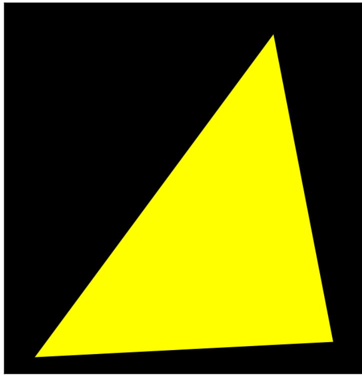


Drawing the Inner Triangle

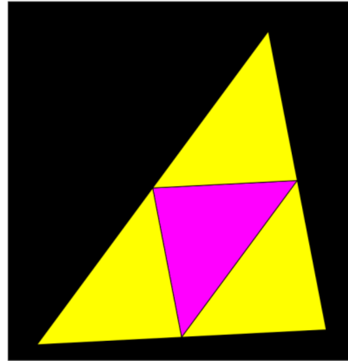
```
a = [(x[0]+x[1])/2,(x[1]+x[2])/2,(x[2]+x[0])/2]

b = [(y[0]+y[1])/2,(y[1]+y[2])/2,(y[2]+y[0])/2]

DrawPoly(a,b,color=MAGENTA)
```

Computing Midpoints

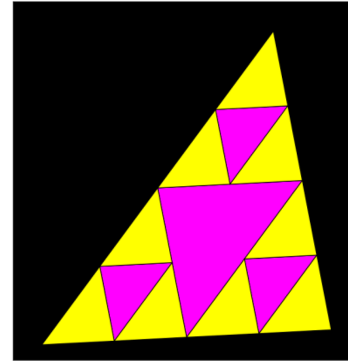
The Notion of Level



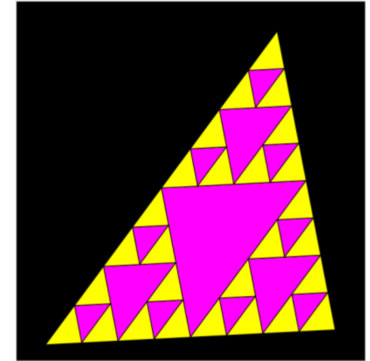
A 0-level partition



A 1-level partition



A 2-level partition



A 3-level partition

Notice that a 3-level partition involves
a display of the inner triangle and
a 2-level partitioning of each corner triangle

PseudoCode

```
def Partition(VertexInformation,Level):
    if Level ==0:
        Draw a yellow triangle
    else:
        Draw inner triangle magenta
        # Partition the 3 corner triangles
        # with the level reduced by one.
        Partition(VertexInformation,Level-1)
        Partition(VertexInformation,Level-1)
        Partition(VertexInformation,Level-1)
```

```
def Partition(x,y,Level):
    if Level==0:
        DrawPoly(x,y,color=YELLOW)
    else:
        a =     specify x-values of vertices
        b =     specify y-values of vertices
        DrawPoly(a,b,color=MAGENTA)
        u0 =    specify x-values of vertices
        v0 =    specify y-values of vertices
        Partition(u0,v0,Level-1)
        u1 =    specify x-values of vertices
        v1 =    specify y-values of vertices
        Partition(u1,v1,Level-1)
        u2 =    specify x-values of vertices
        v2 =    specify y-values of vertices
        Partition(u2,v2,Leval-1)
```

4 things to do

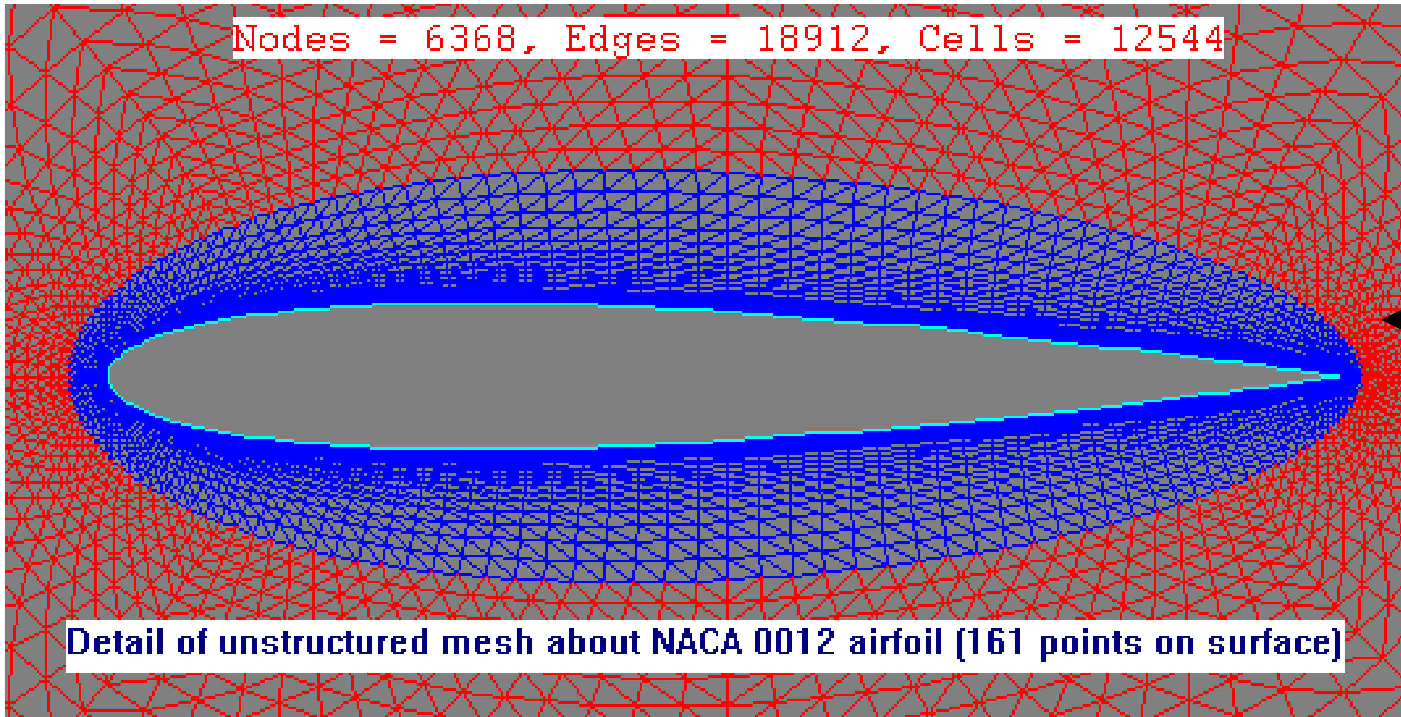
```
def Partition(x,y,Level):
    if Level==0:
        DrawPoly(x,y,color=YELLOW)
    else:
        a =     specify x-values of vertices
        b =     specify y-values of vertices
        DrawPoly(a,b,color=MAGENTA)
        u0 =    specify x-values of vertices
        v0 =    specify y-values of vertices
        Partition(u0,v0,Level-1)
        u1 =    specify x-values of vertices
        v1 =    specify y-values of vertices
        Partition(u1,v1,Level-1)
        u2 =    specify x-values of vertices
        v2 =    specify y-values of vertices
        Partition(u2,v2,Leval-1)
```

Inner Triangle

Perform a level L-1 partition of each corner triangle

A Note on Chopping up a Region into Triangles…

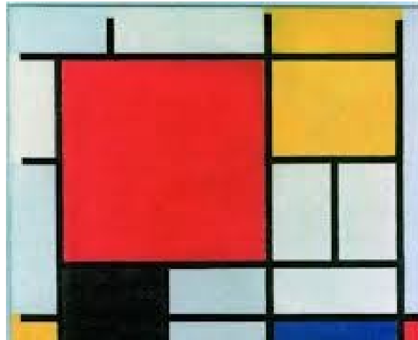
It is Important!

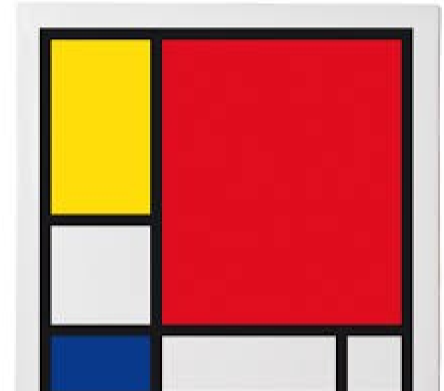


An Area Of Interest

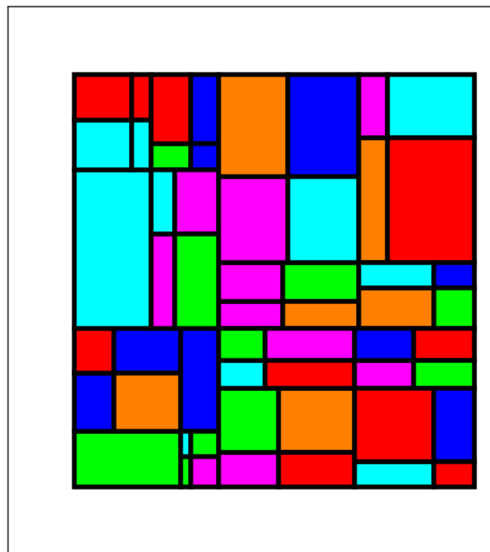
Step One in simulating flow around an airfoil is to generate a mesh and (say) estimate velocity at each mesh point.

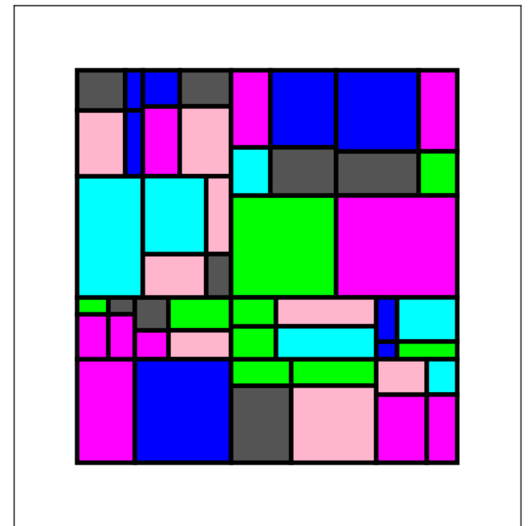
Next Up: Random Mondrians





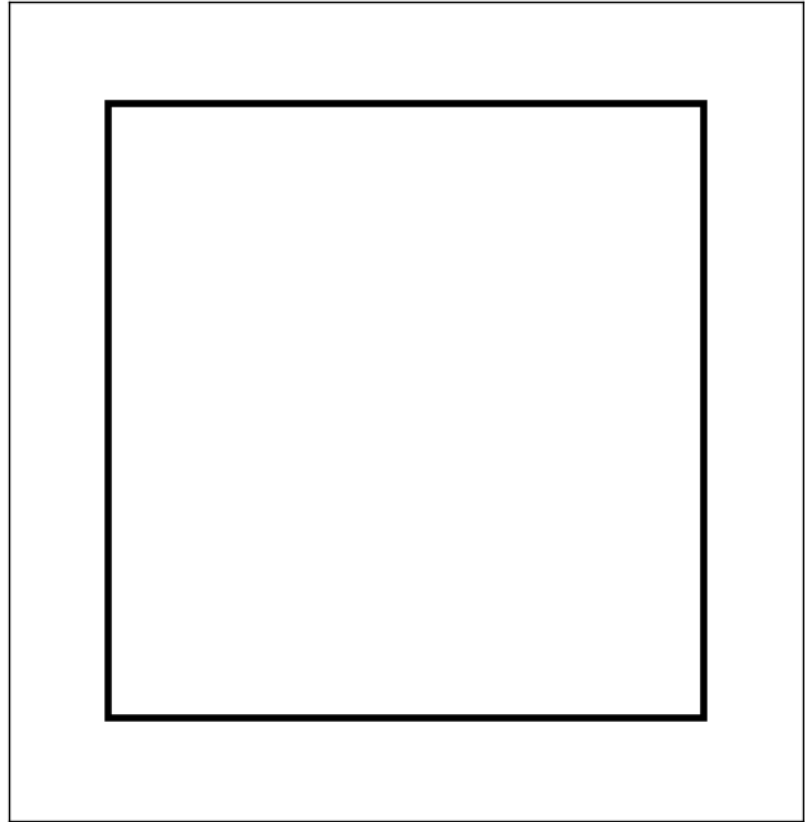
Using Python:





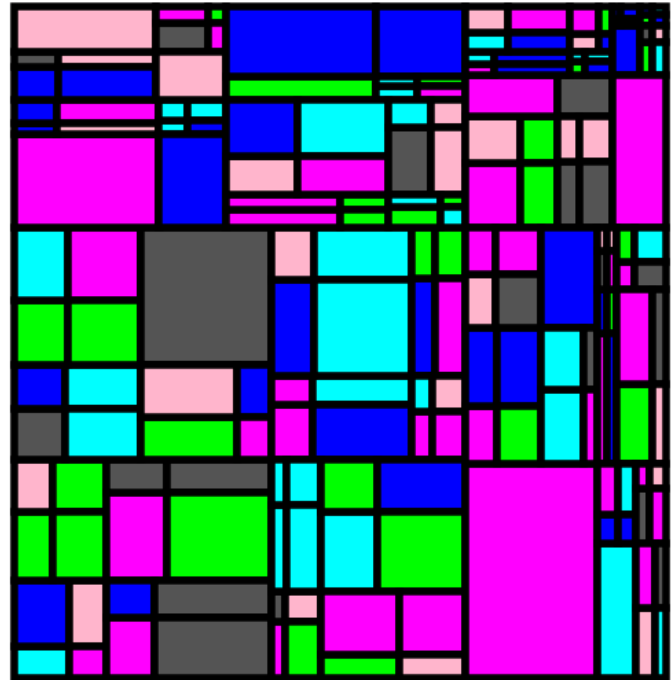
Random Mondrian

Given This:

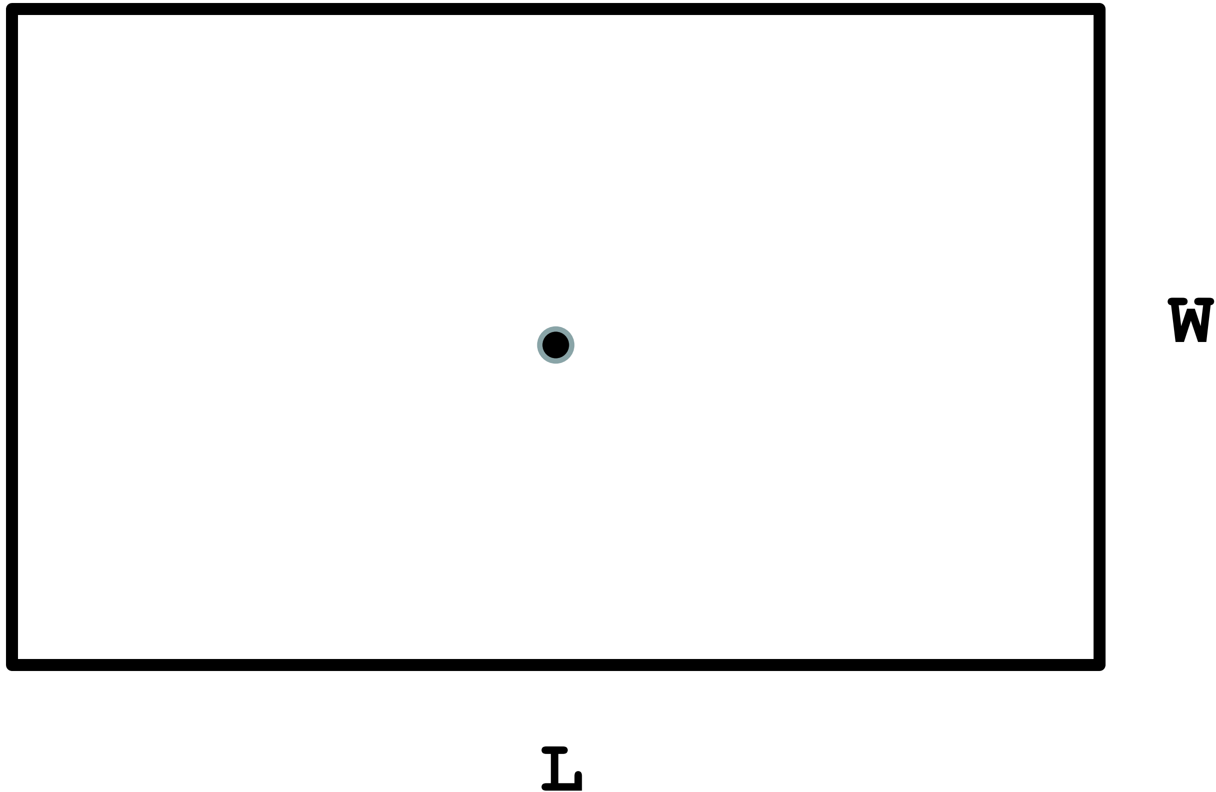


Random Mondrian

Draw This:

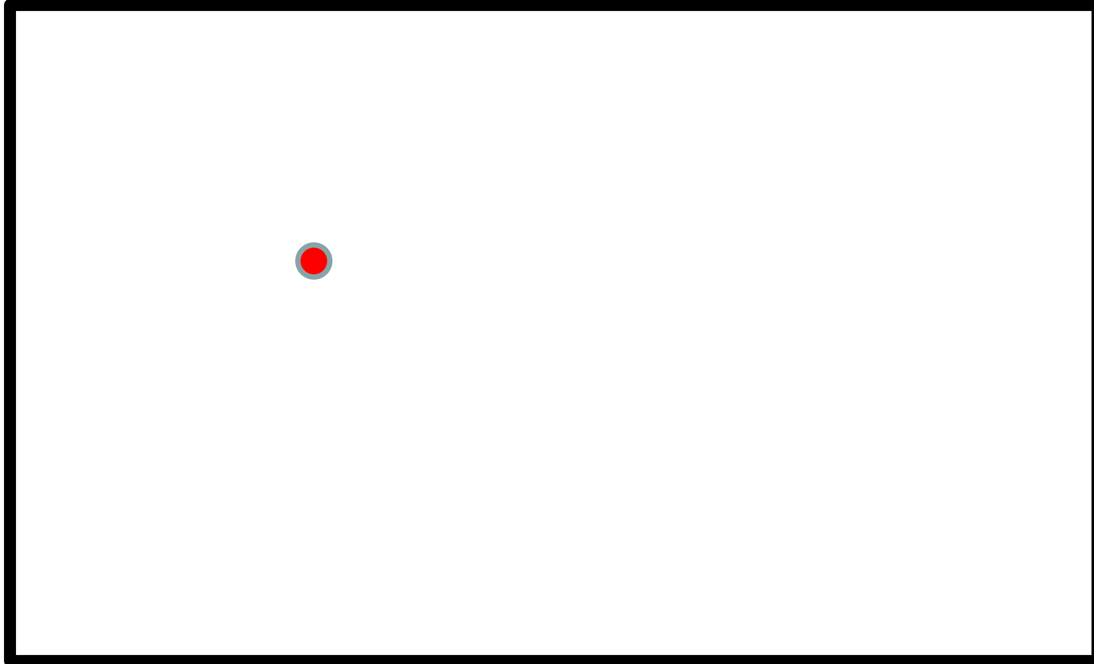


The Subdivide Process Applies to a Rectangle

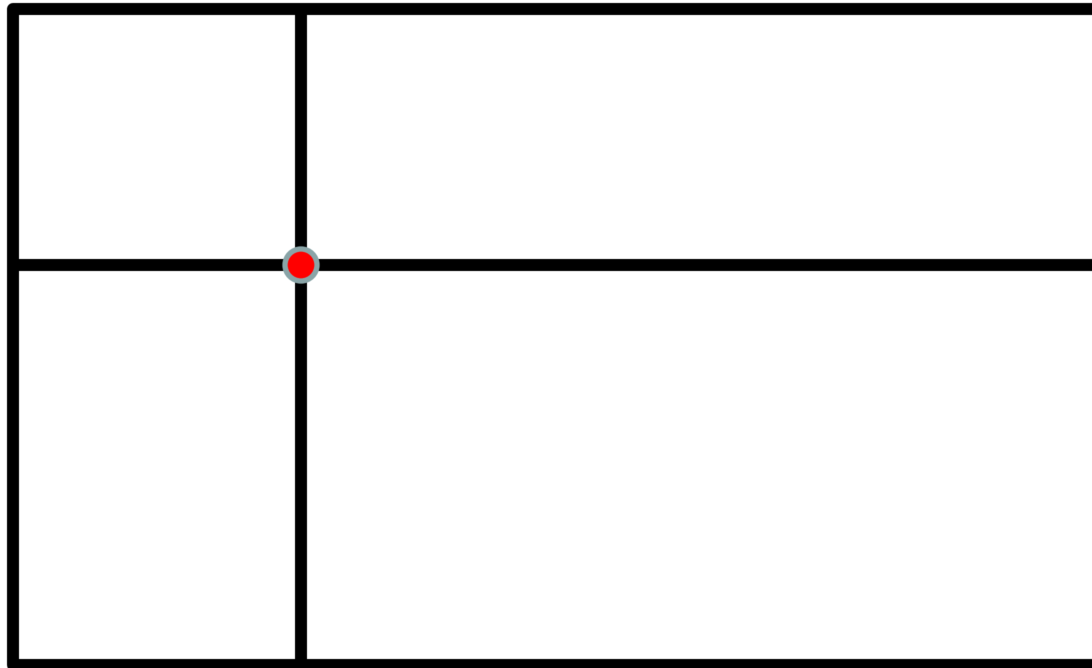


Given a rectangle, either randomly color it or subdivide it

Subdivision Starts with a Random Dart Throw

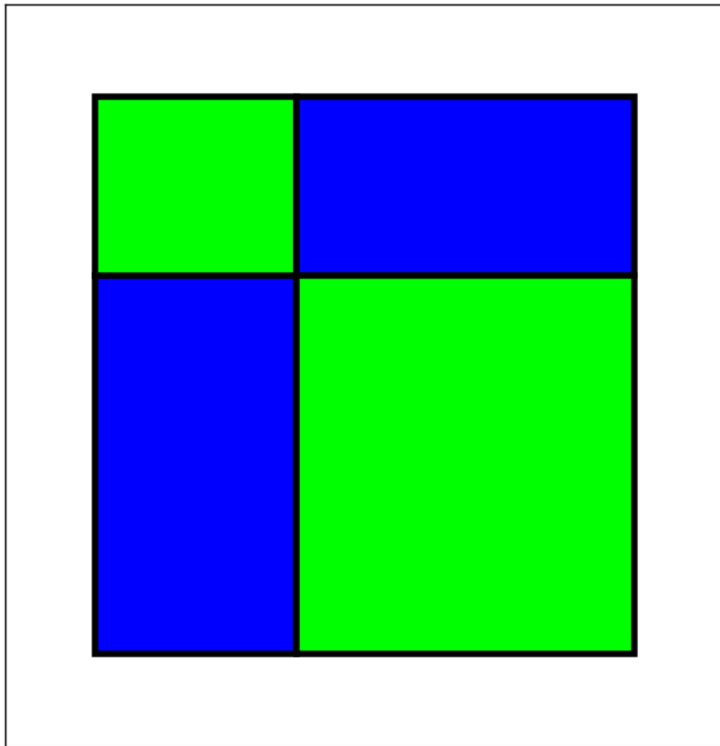


This Defines 4 Smaller Rectangles

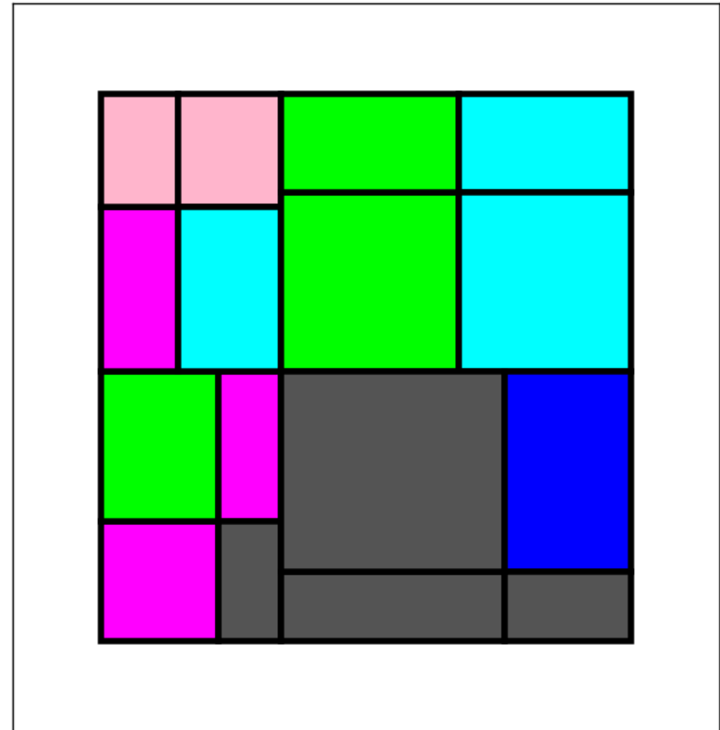


Repeat the Process on Each of the Smaller Rectangles

Notion of Level



A 1-level Partitioning



A 2-level Partitioning

Pseudocode

```
def Mondrian(x,y,L,W,level):
    if level ==0:
        c = RandomColor())
        DrawRect(x,y,L,W,color=c))
    else:
        Mondrian(upper left rectangle info,level-1)
        Mondrian(upper right rectangle info,level-1)
        Mondrian(lower left rectangle info,level-1)
        Mondrian(lower right rectangle info,level-1)
```

Side Note…

We need some new technology to organize the selection random colors.

We need lists whose entries are lists.

Lists with Entries that Are Lists

An Example:

```
cyan     = [0.0,1.0,1.0]
magenta  = [1.0,0.0,1.0]
yellow   = [1.0,1.0,0.0]
myColors = [cyan,magenta,yellow]
print myColors[1][2]
```

Lists with Entries that Are Lists

```
from simpleGraphics import *
from random import randint as randi

def RandomColor():
    """ Returns a randomly selected
    rgb list."""
    c = [RED,GREEN,BLUE,ORANGE,CYAN]
    i = randi(0,len(c)-1)
    return c[i]
```

Next Up

A Non-Graphics Example of Recursion: The Factorial Function

Recursive Evaluation of Factorial

Recall the factorial function:

```
def F(n):
    x = 1
    for k in range(1,n+1):
        x = x*k
    return x
```

5! = 1x2x3x4x5

Recursive Evaluation of Factorial

Q. How would you compute 6! given that you have computed 5! = 120 ?

A. 6! = 120 x 6

5! = 1x2x3x4x5

Recursive Evaluation of Factorial

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

How does this work?

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
m --> 3

x -->
```

We are in the calling script

Executing F(3)

```
m = 3
x = F(m)
print
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
m --> 3
x -->
```

```
n --> 3
a -->
return
```

The function F is called with argument 3. We open up a call frame.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x -->

n --> 3

a -->

return

We encounter a function call. F is called with argument equal to 2.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x -->

n --> 3

a -->

return

n --> 2

a -->

return

We open up a call frame.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3
x -->

n --> 3
a -->
return

n --> 2
a -->
return

We encounter a function call. F is called with argument 1

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3
x -->

n --> 3
a -->
return

n --> 2
a -->
return

n --> 1
a -->
return

We open up a call frame.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3
x -->

n --> 3
a -->
return

n --> 2
a -->
return

n --> 1
a -->
return 1

The value of 1 is "assigned" to return

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3
x -->

n --> 3
a -->
return

n --> 2
a --> 1
return

n --> 1
a -->
return 1

The value is sent back to the caller.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3
x -->

n --> 3
a -->
return

n --> 2
a --> 1
return

n --> 1
a -->
return 1

That function call is over

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m -->	3
x -->	

n -->	3
a -->	
return	

n -->	2
a -->	1
return	

Control now passes to this "edition" of F

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m -->	3
x -->	

n -->	3
a -->	
return	

n -->	2
a -->	1
return	2

Control passes to this "edition" of F. The value 2 is "assigned" to return

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m -->	3
x -->	

n -->	3
a -->	2
return	

n -->	2
a -->	1
return	2

The value is returned to the caller.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m -->	3
x -->	

n -->	3
a -->	2
return	

n -->	2
a -->	1
return	2

The function call is over

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x -->

n --> 3

a --> 2

return

Control now passes to this "edition" of F

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x -->

n --> 3

a --> 2

return 6

The value 6 is "assigned" to return

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x --> 6

n --> 3

a --> 2

return 6

The value is returned to the caller.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
def F(n):
   if n<=1:
      return 1
   else:
      a = F(n-1)
      return n*a
```

m --> 3

x --> 6

n --> 3

a --> 2

return 6

This function call is over.

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
m --> 3

x --> 6
```

Control passes to the script that asked for F(3)

Executing F(3)

```
m = 3
x = F(m)
print x
```

```
m --> 3

x --> 6
```

Output: 6

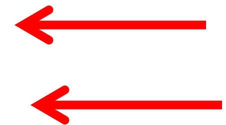
All Done!

Back To Merge Sort

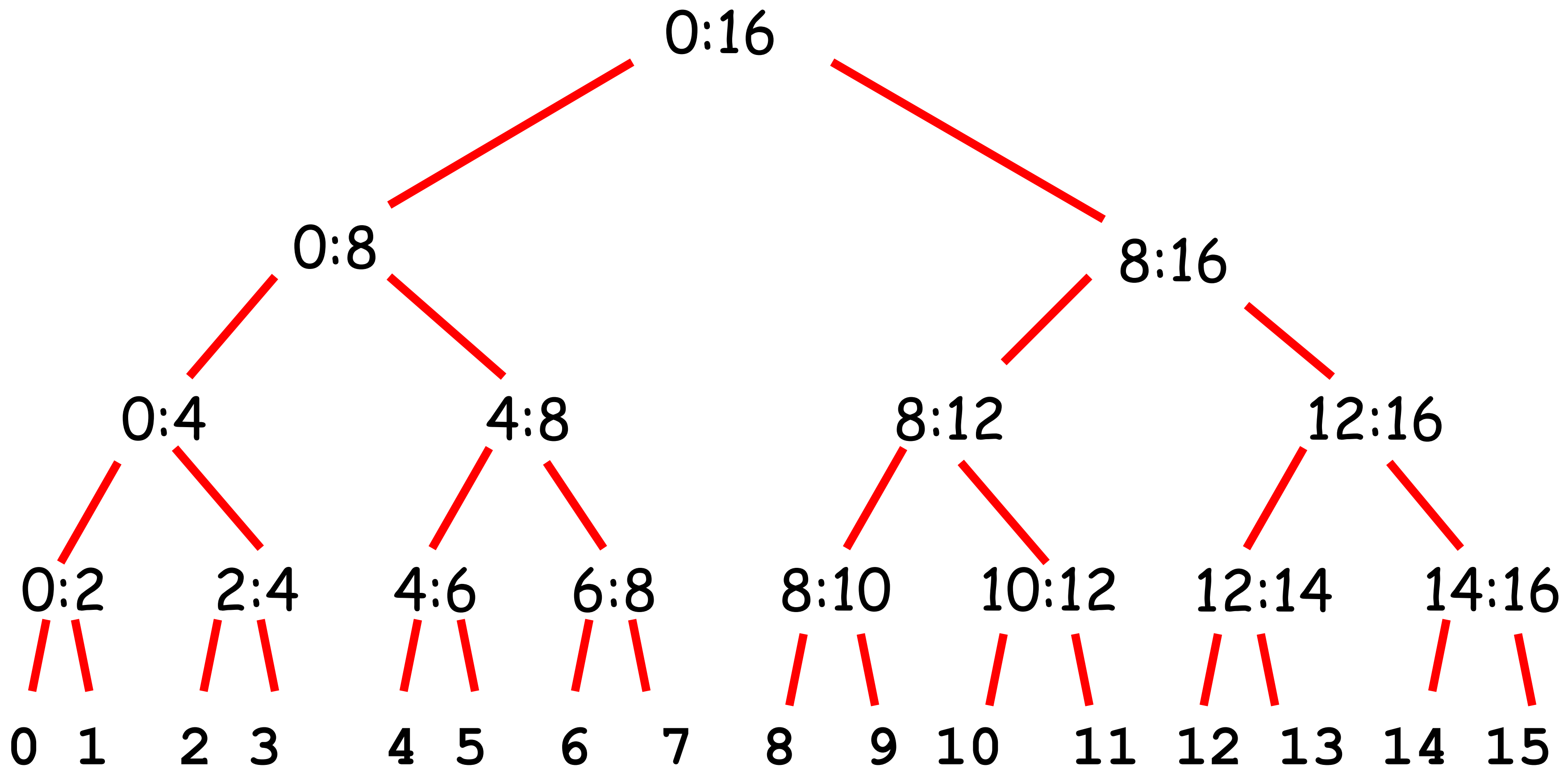
Recursive Merge Sort

```
def MergeSort(a):
     n = length(a)
     if n==1:
         return a
     else:
         m  = n/2
         u0 = list(a[:m])
         u1 = list(a[m:])
         y0 = MergeSort(u0)
         y1 = MergeSort(u1)
         return Merge(y0,y1)
```

A function can call Itself!

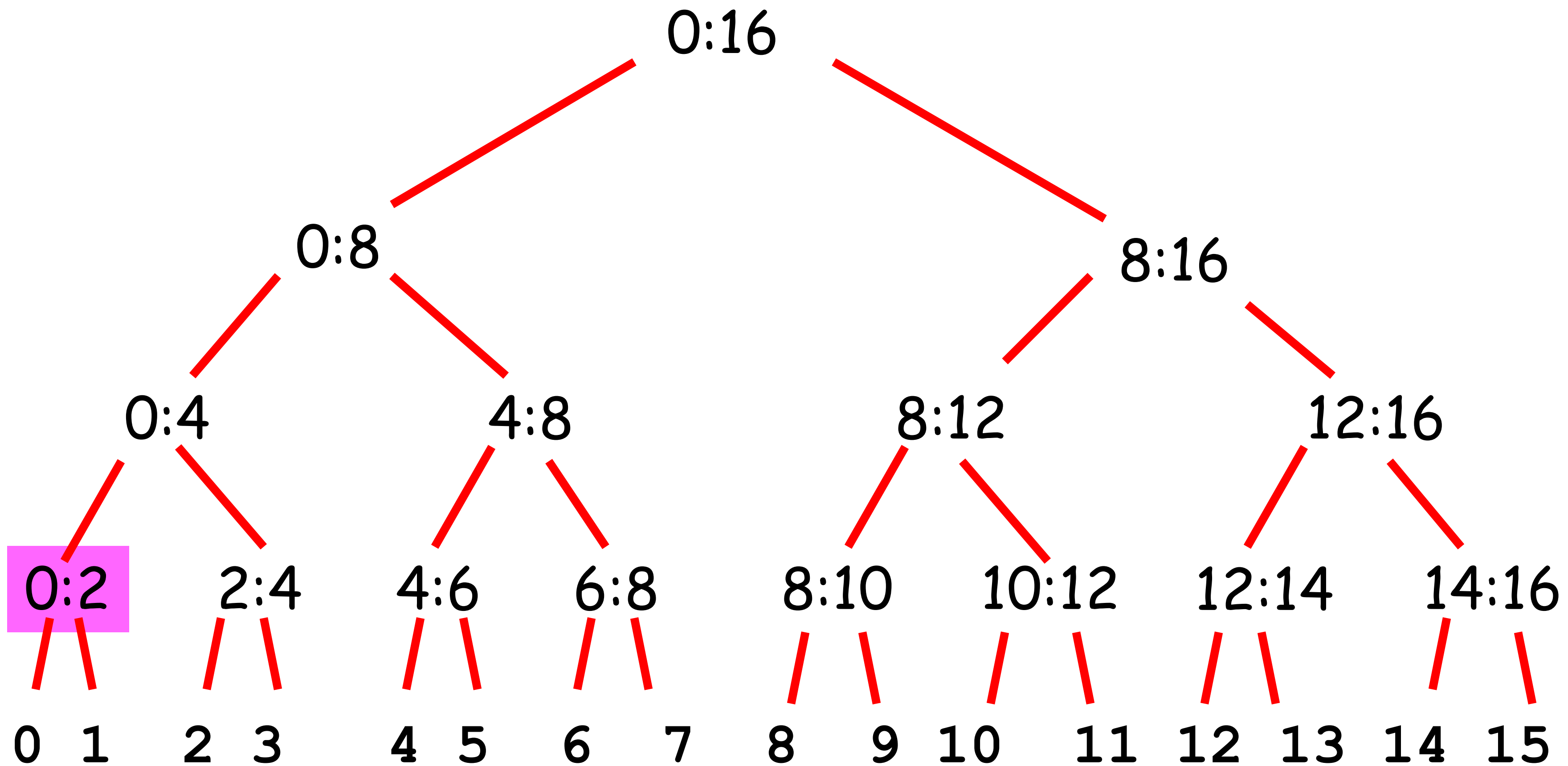


A Schematic



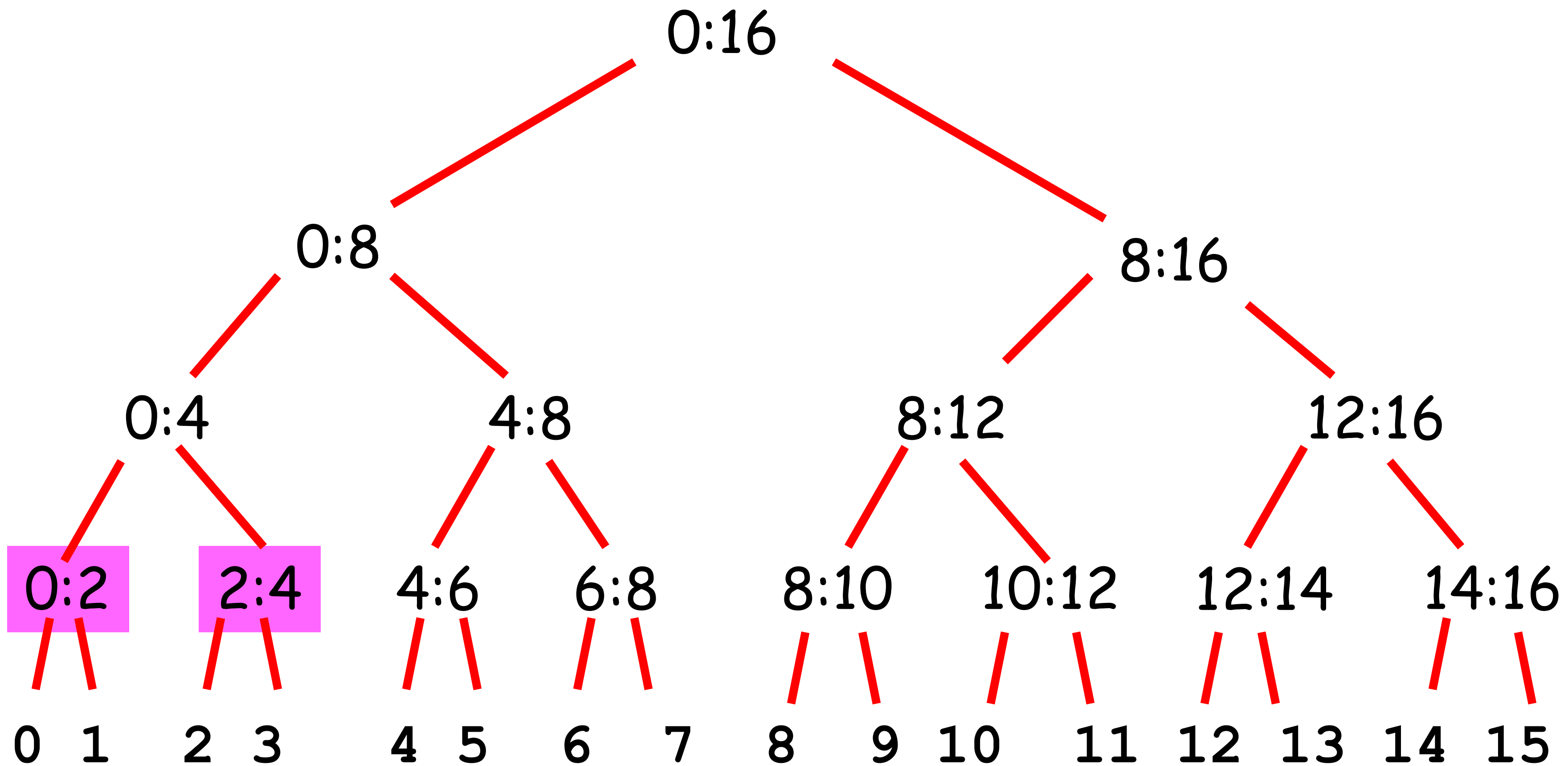
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted

A Schematic



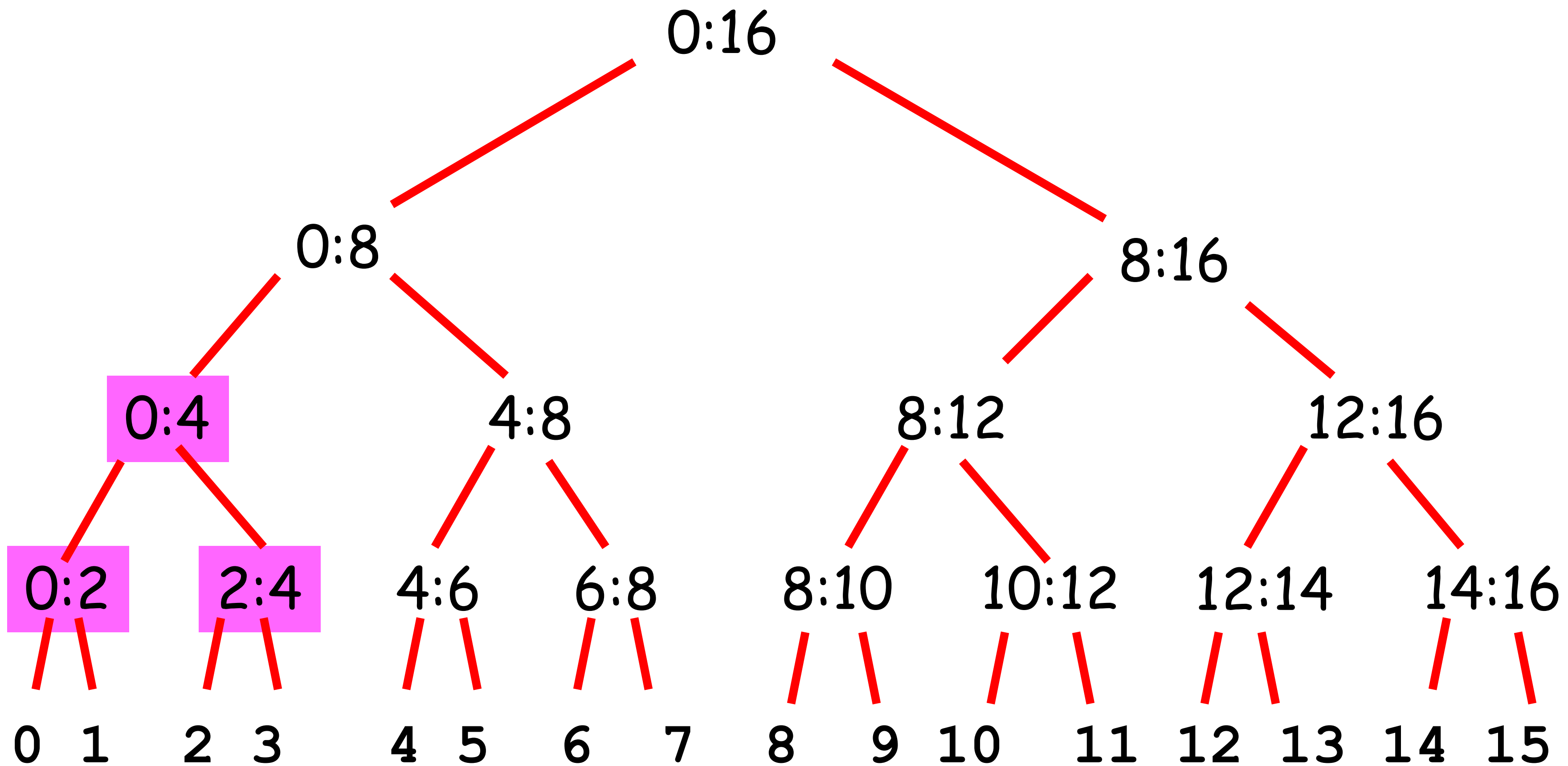
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



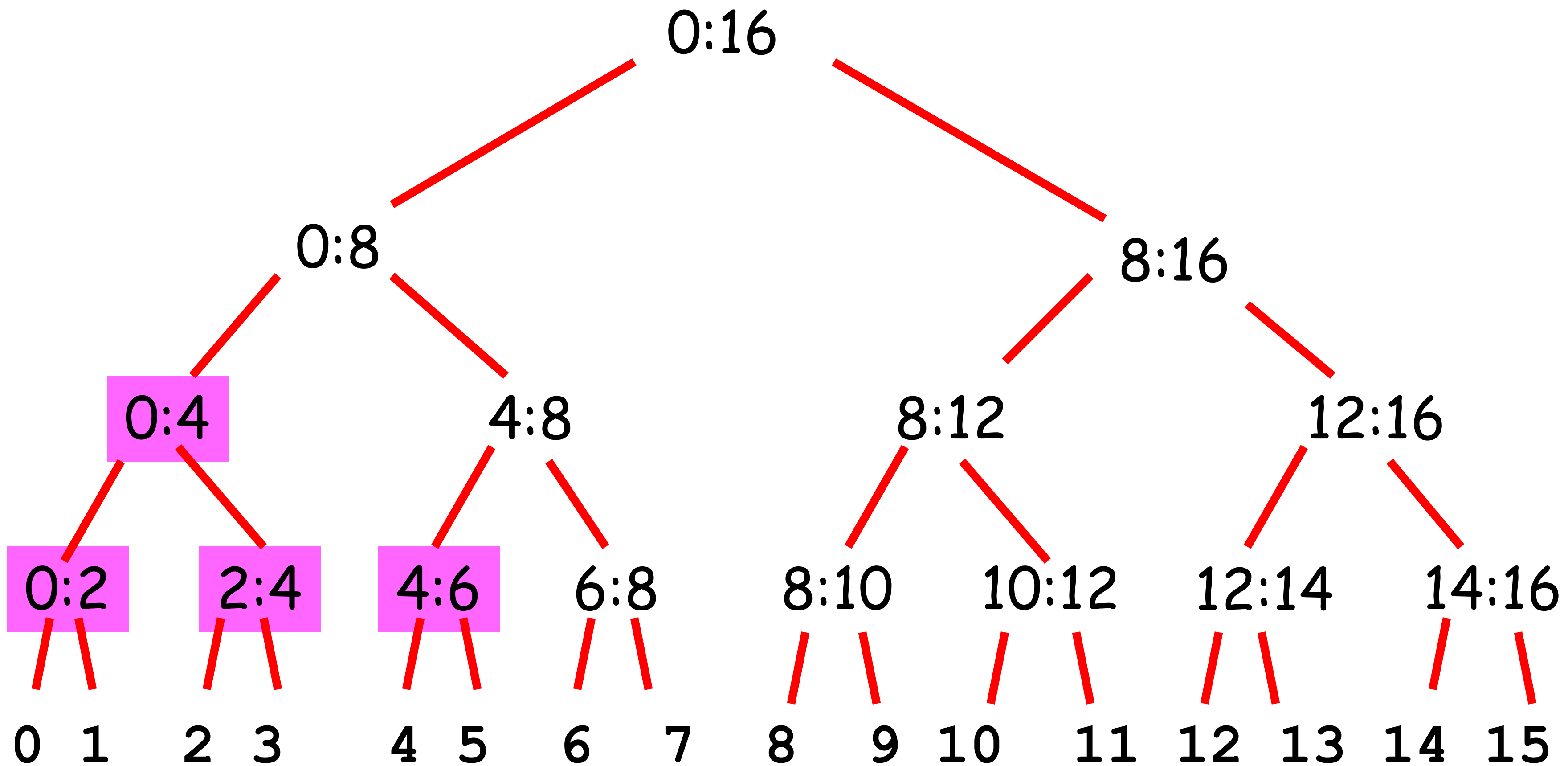
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



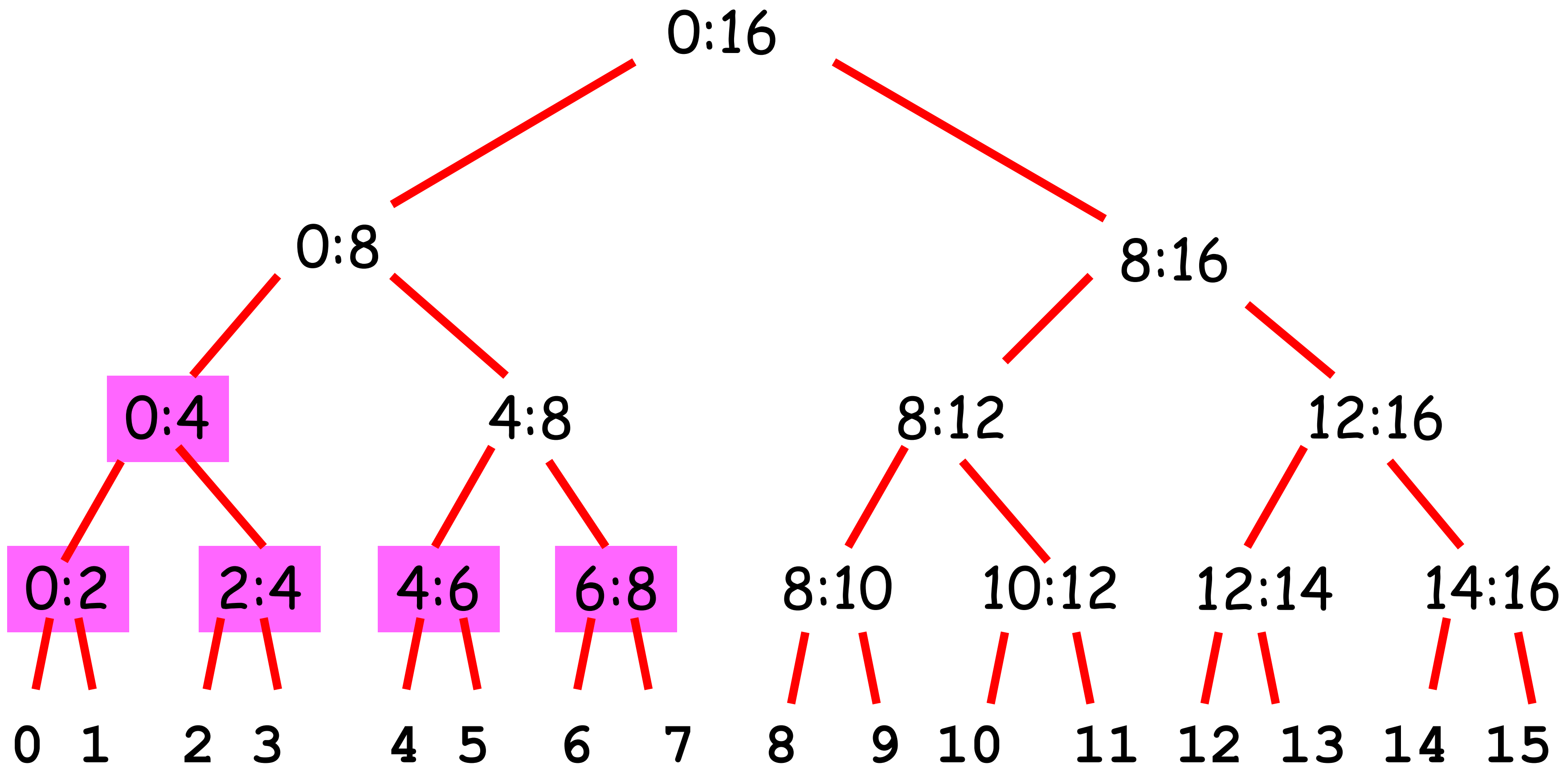
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



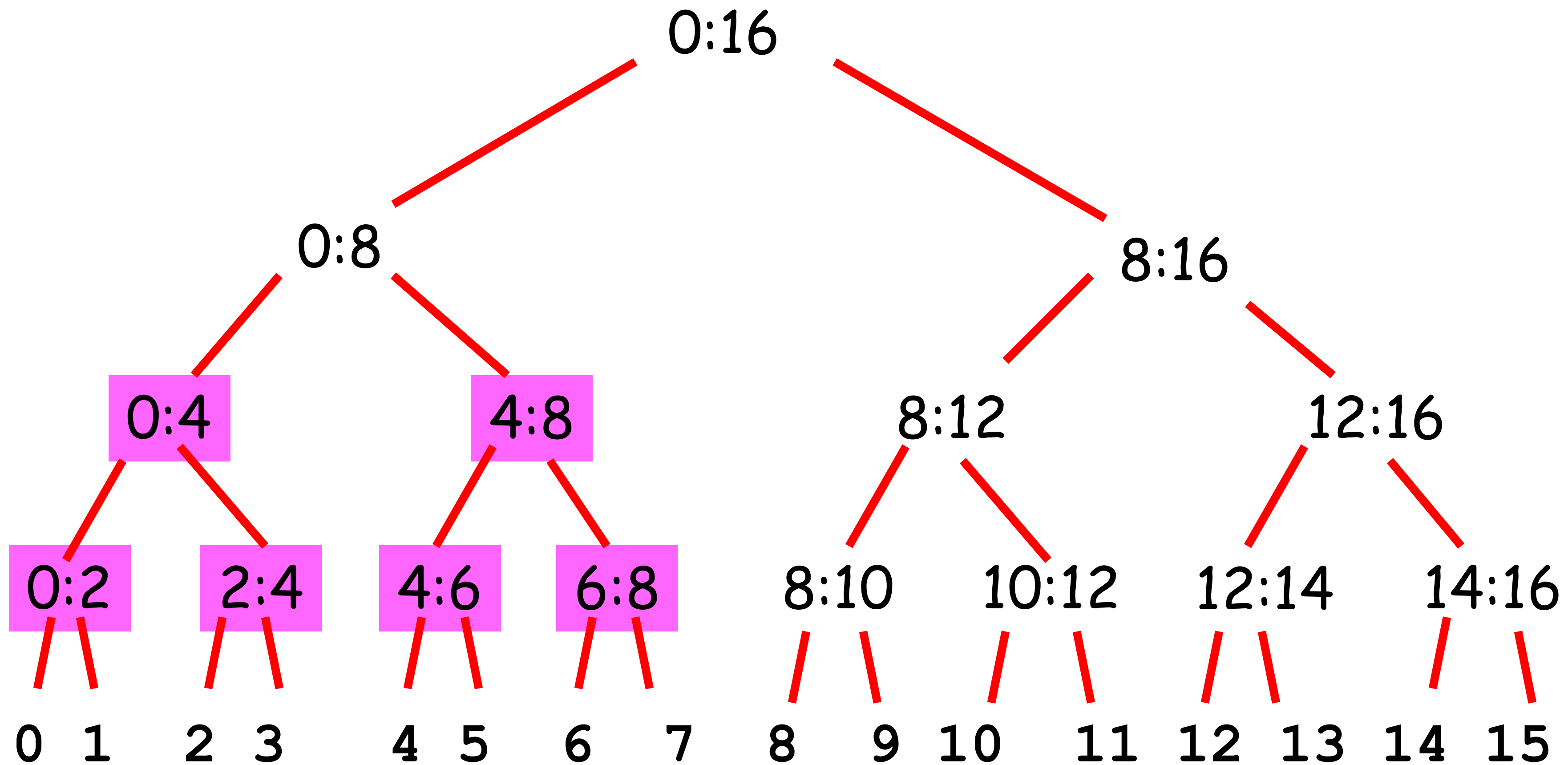
A Sorted List is produced at each ":"   Let's look at the order in which

A Schematic



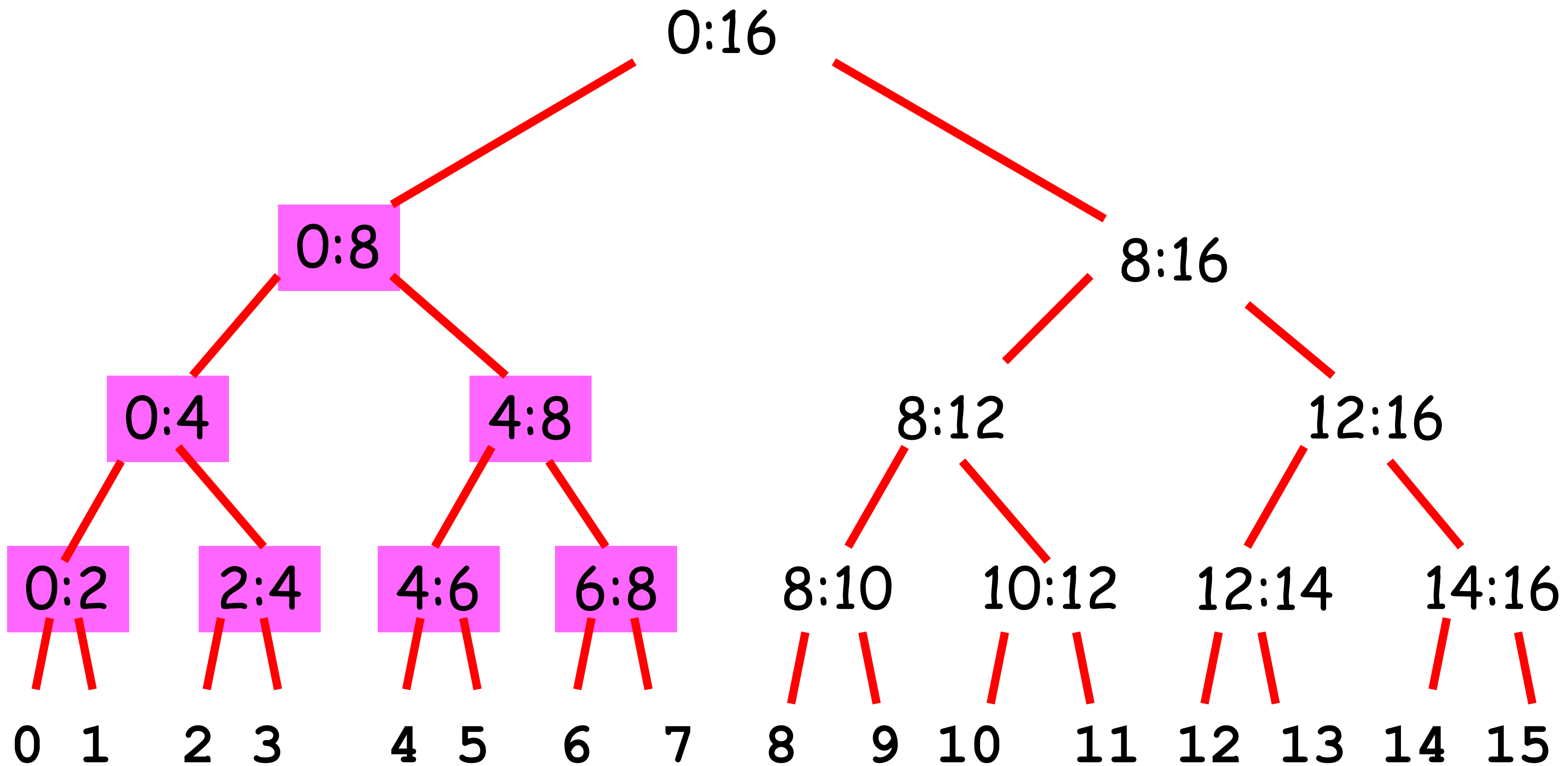
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



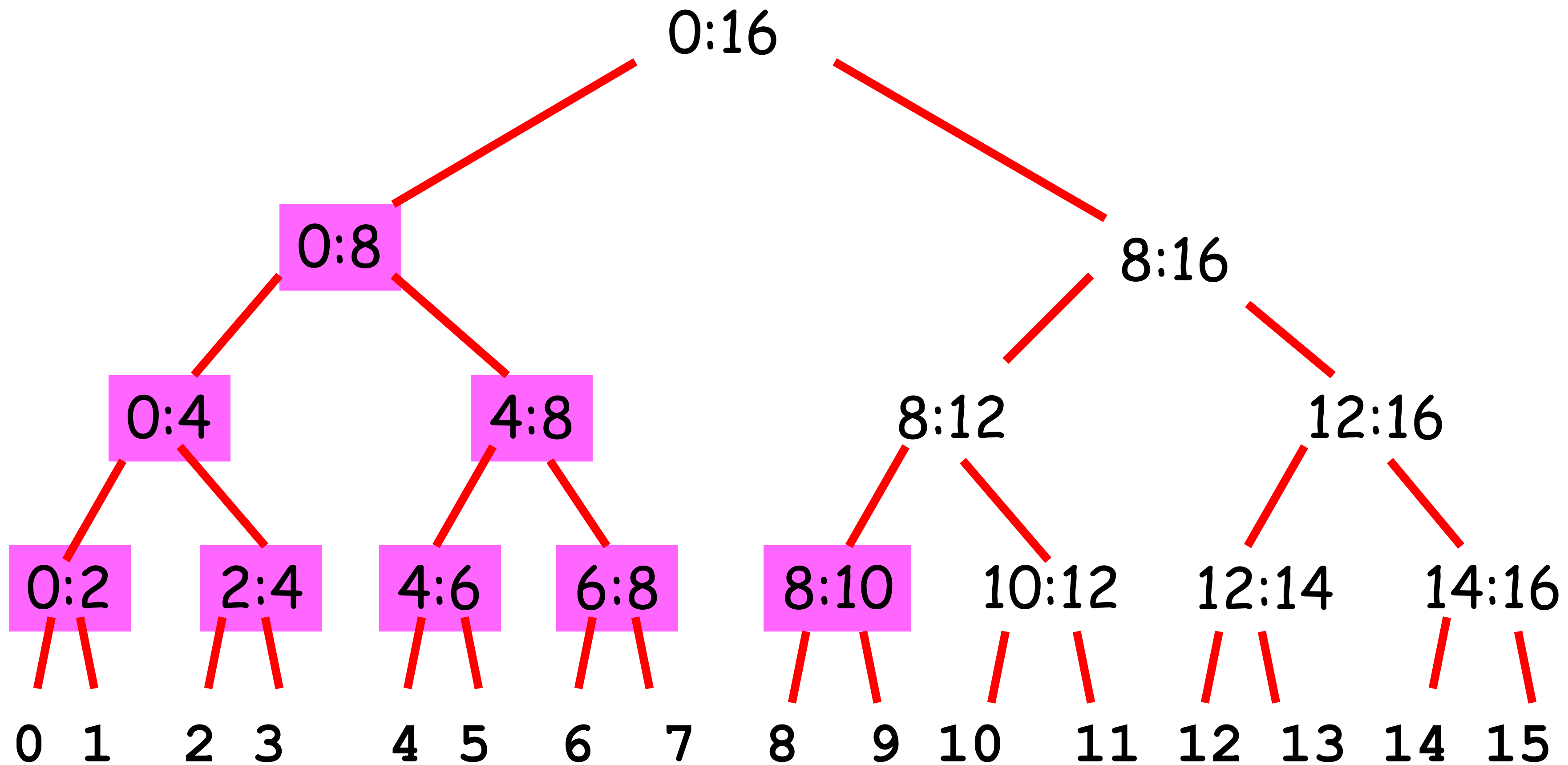
A Sorted List is produced at each ":"   Let's look at   the order in which

A Schematic



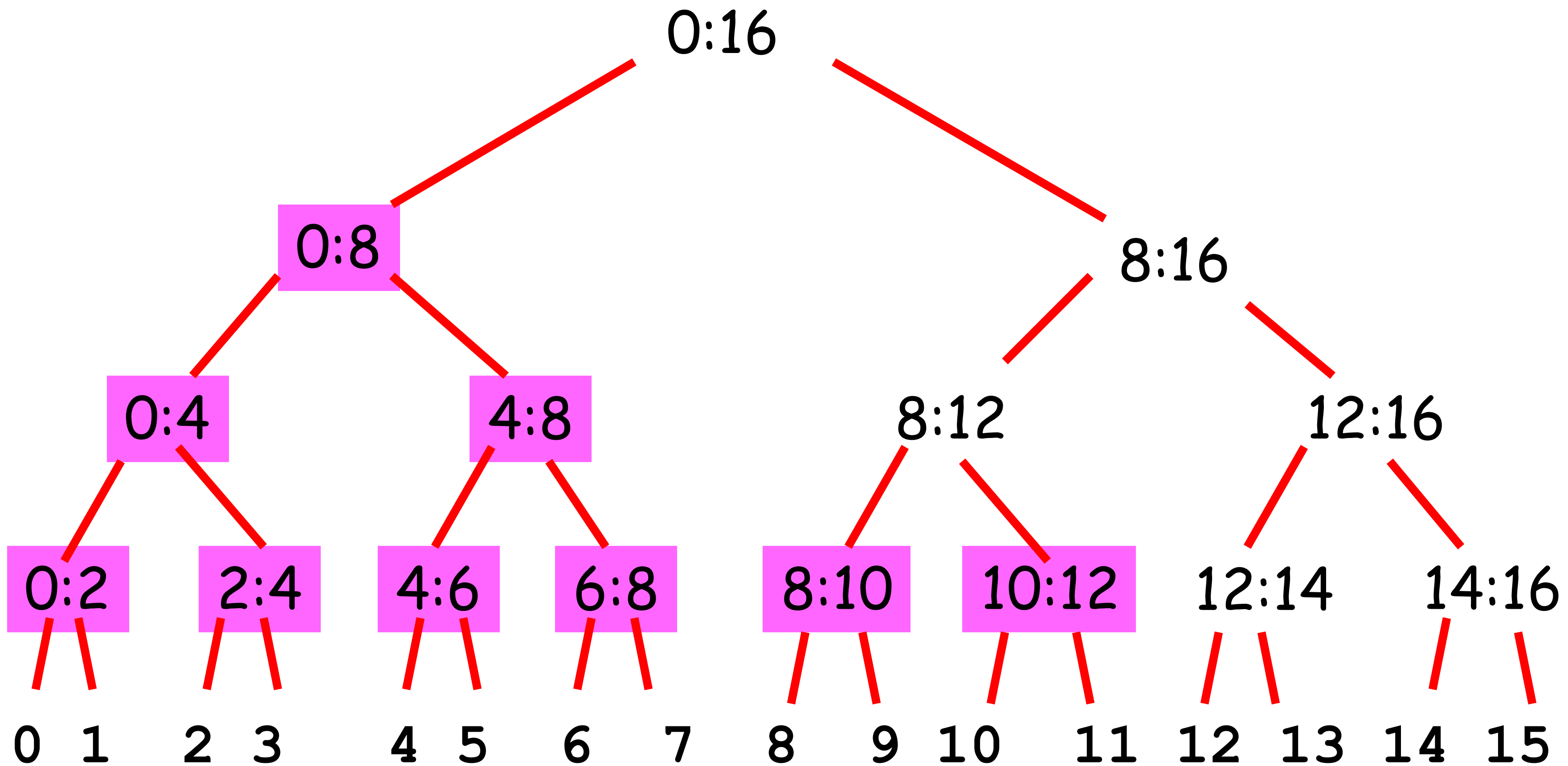
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



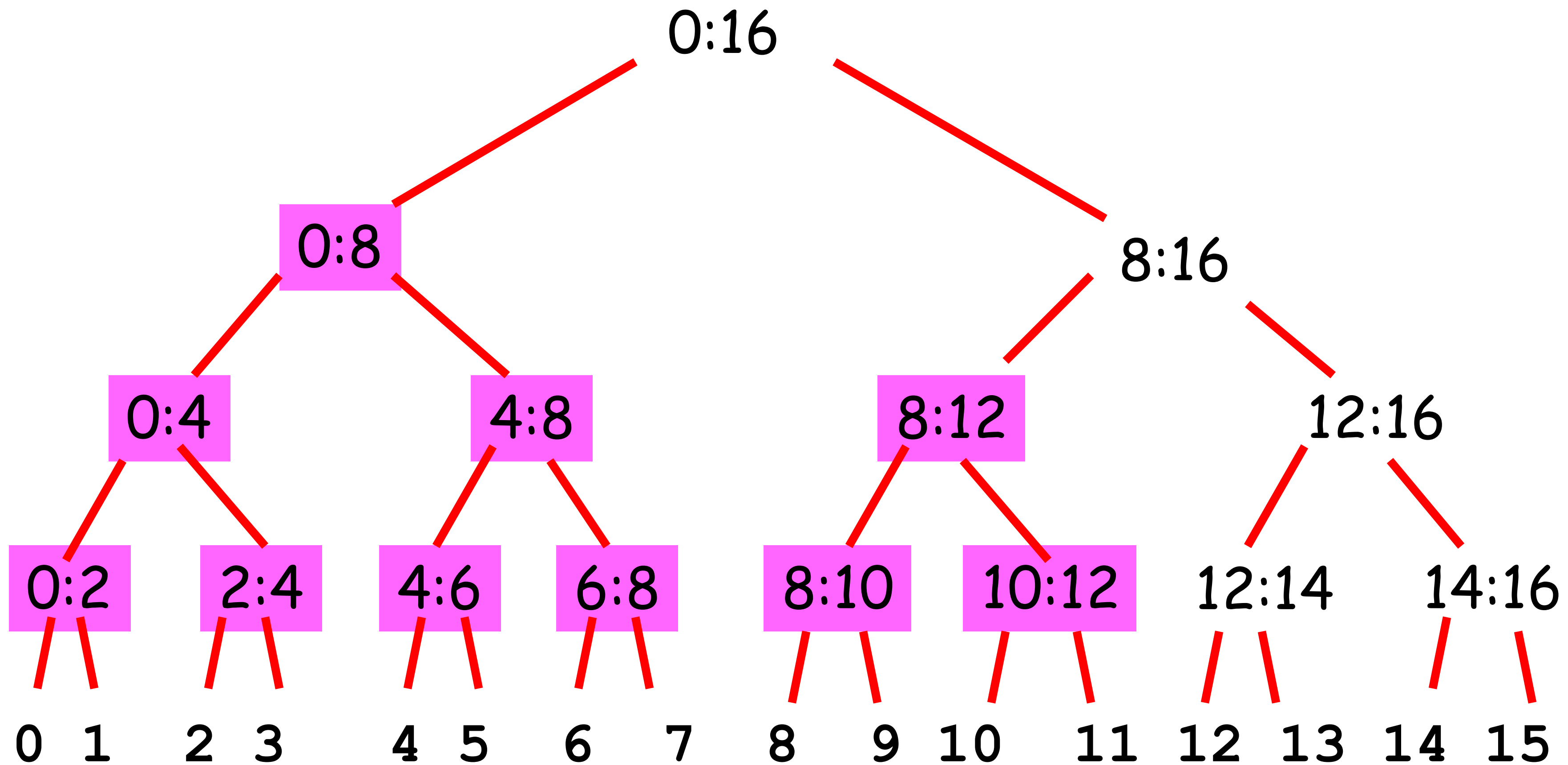
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



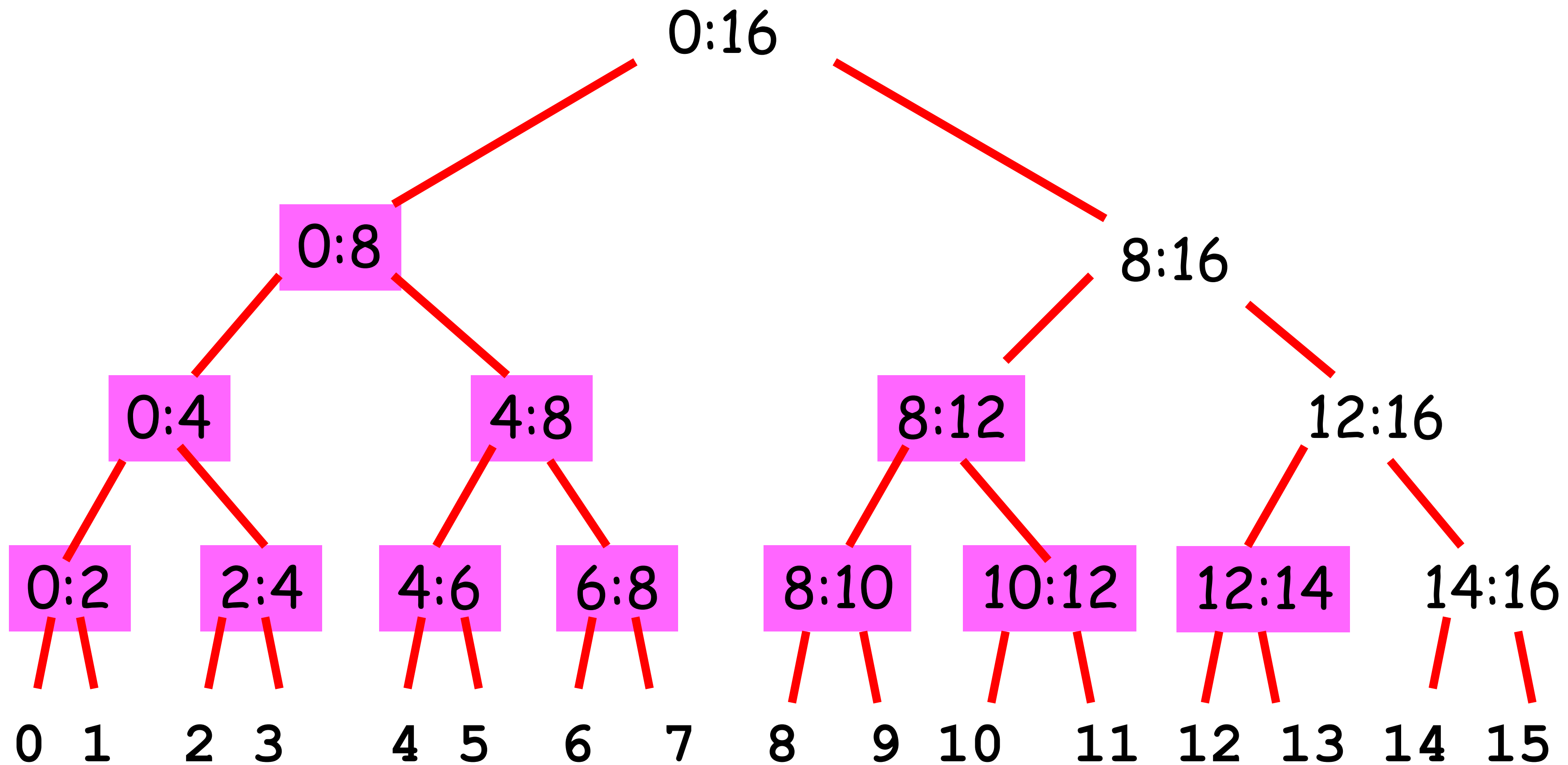
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



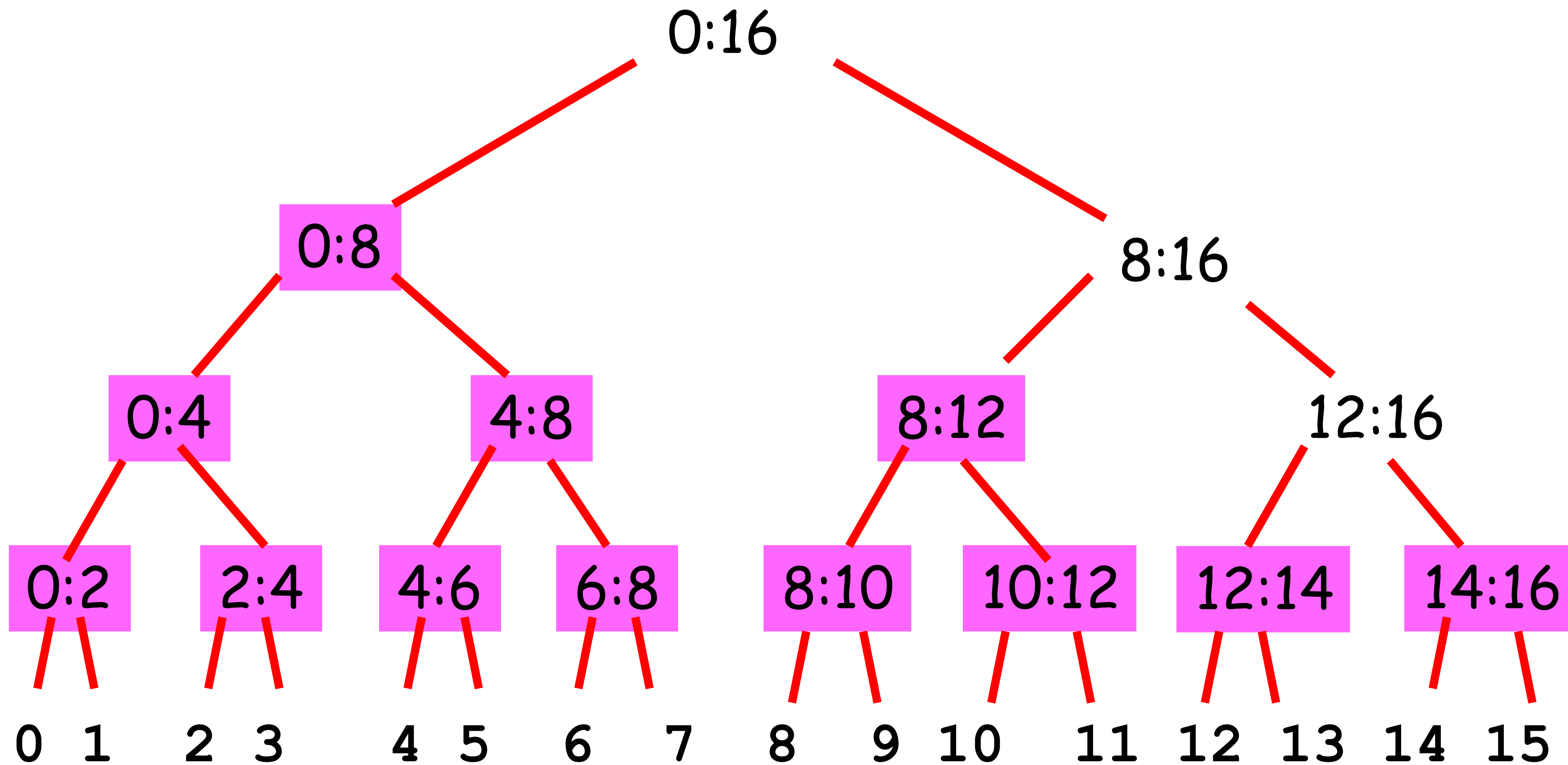
A Sorted List is produced at each ":"   Let's look at the order in which lists are sorted.

A Schematic



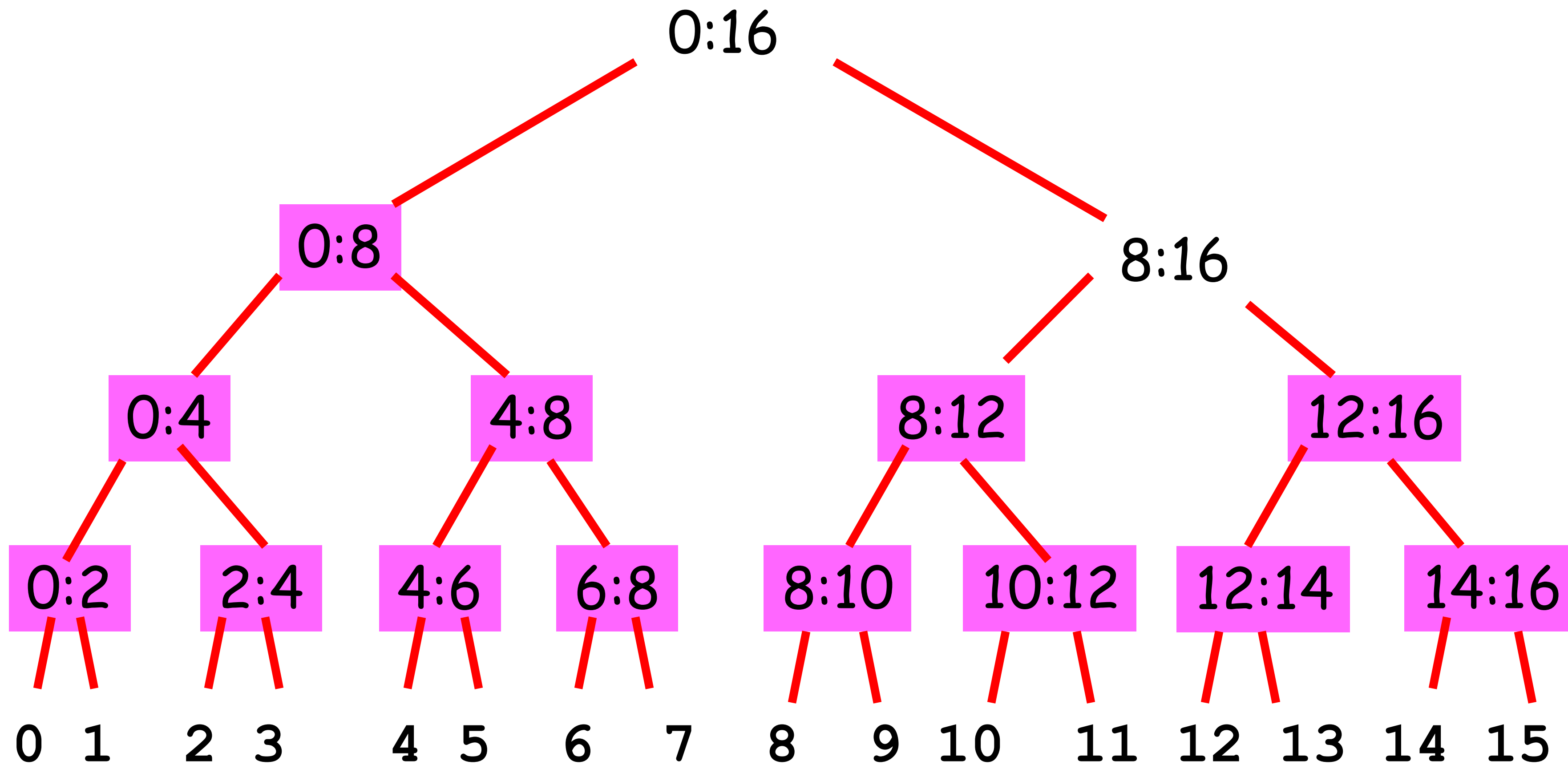
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



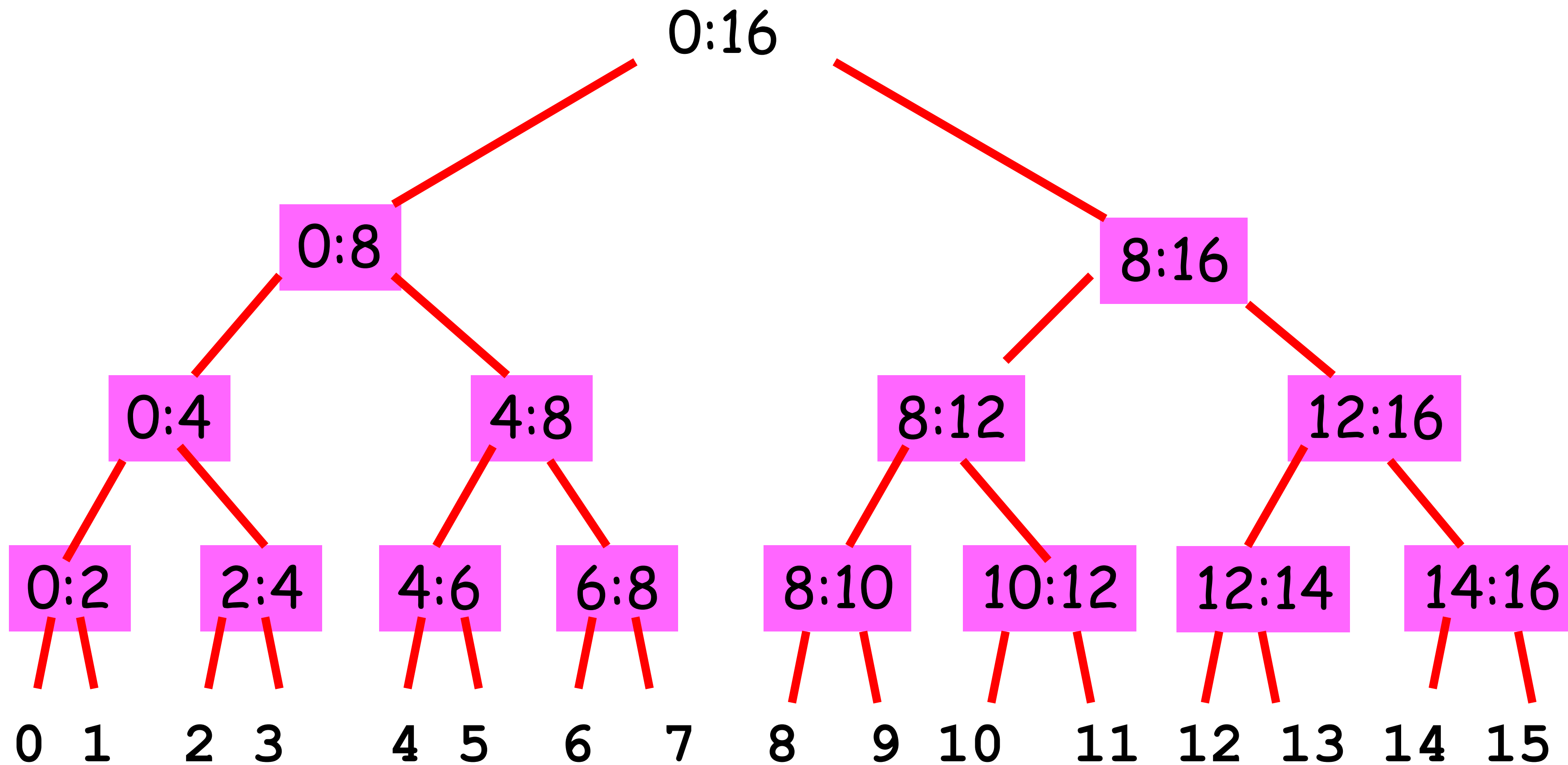
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



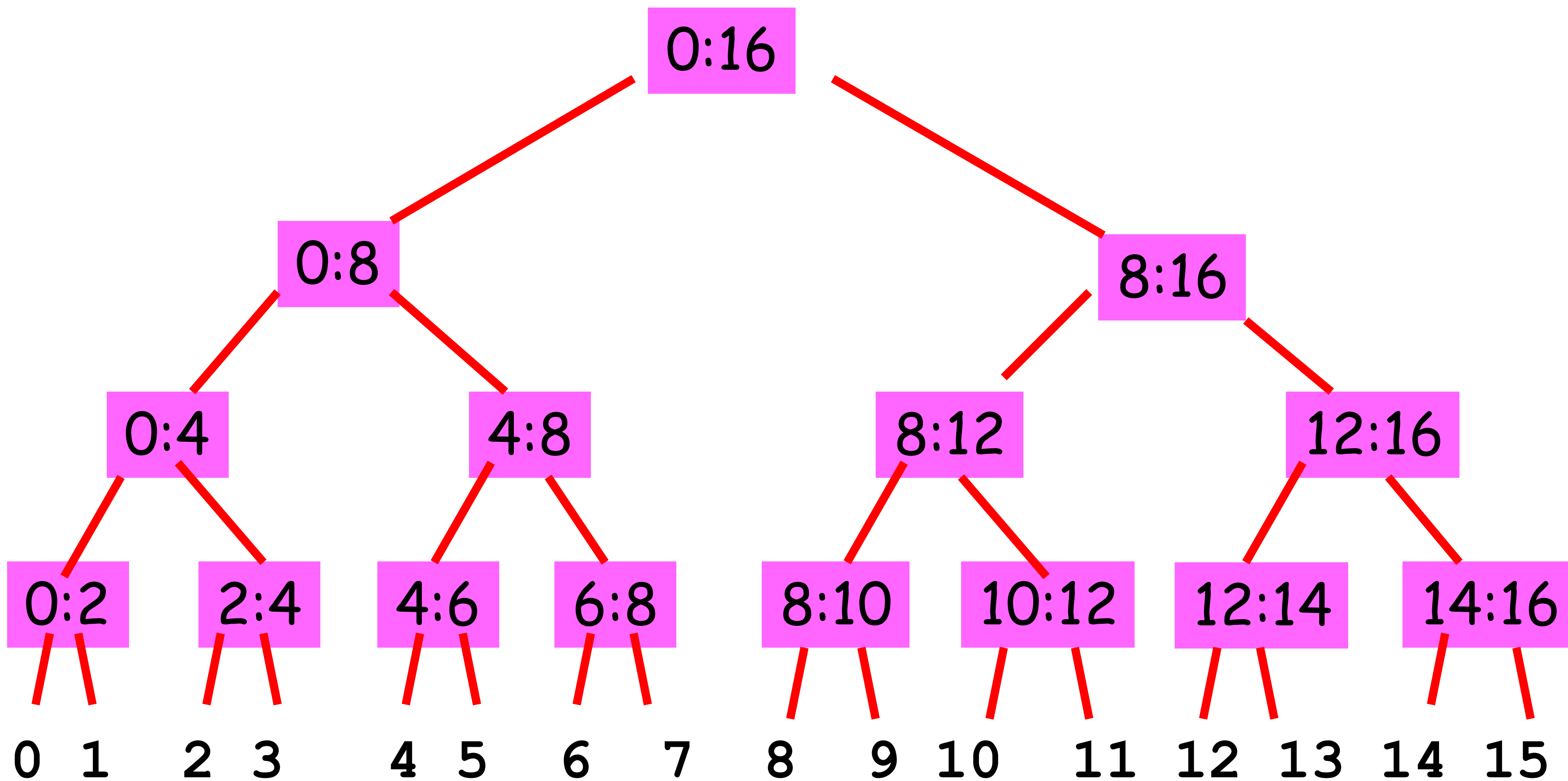
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



All Done!

Some Conclusions

Recursion is sometimes the simplest way to organize a computation.

It would be next to impossible to do the triangle partition problem any other way.

On the other hand, factorial computation is easier via for-loop iteration.

Some Conclusions

The function calls required by a recursive solution can be a significant overhead.

Some Conclusions

Infinite recursion (like infinite loops) can happen so careful reasoning is required.

Will we reach the "base case"?

Graphics examples: We will reach Level==0
Factorial: We will reach n==1
MergeSort: We will reach len(a) ==1