Fractals

Fractals are a paradox. Amazingly simple, yet infinitely complex. Benoit Mandelbrot (unconventional 20th century mathematician) created the term fractal from the Latin word *fractus* (meaning irregular or fragmented) in 1975. At their most basic, fractals are a visual expression of a repeating pattern or formula that starts out simple and gets progressively more complex.

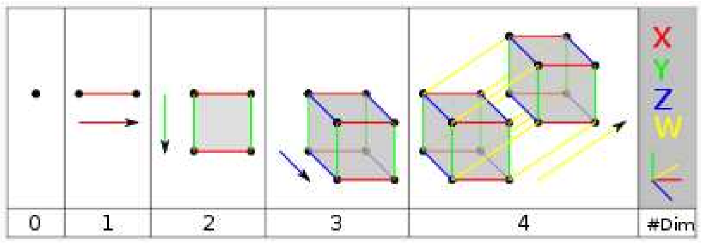
All fractals show a degree of what's called **self-similarity**. This means that as you look closer and closer into the details of a fractal, you can see a replica of the original pattern.

These self-similar patterns are the result of a simple equation, or mathematical statement. Fractals are created by repeating this equation through a feedback loop in a process called **iteration**, where the results of one iteration form the input value for the next. For example, if you look at the interior of a nautilus shell, you'll see that each chamber of the shell is basically a carbon copy of the preceding chamber, just smaller as you trace them from the exterior to the interior.

Fractals are also **recursive,** regardless of scale. So if you ever went into a store's dressing room and find yourself surrounded by mirrors, you maybe didn't realise it, but you were looking at an infinitely recursive image of yourself.

As they are visualy complex, it is interesting what do we get when we observe fractals by means and premises of classic mathematical discipline like geometry. After all, geometry studies the shape and size of visual forms (patterns).

And all of us know that length, width and height are the three dimensions, thus time as fourth and that's that. So a quick reminder:



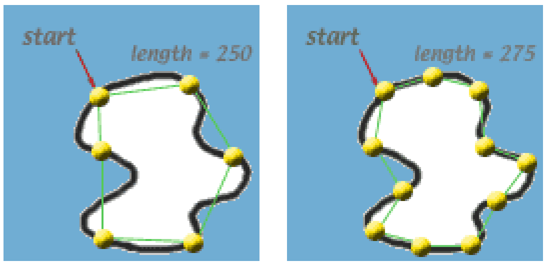
It seems as solid concept.

Fractal dimension

The concept of "fractal dimension" is attributed to Benoit Mandelbrot. His fractal theory was developed in order to try to more precisely quantify the immense complexity of nature in relatively simple equations.

His favourite example of such complexity was the craggy coast of Britain, which, seen from far above, looks somewhat wrinkled and convoluted. Yet, as an observer gets closer and closer to the shore, the complexity of the coastline increases - smooth lines become jagged and more complex, until the observer is so close that she is observing the minute variation in the positions of each individual grain of sand along the shore.

Moreover, we can imagine this observer measuring the length of the coastline with smaller and smaller rulers. As observer measures with increasingly smaller and more precise resolution (meaning replacing meter ruler with centimeter ruler and so on), observer's approximation of the length of the coast will keep increasing. In fact, observer might well conclude that the length diverges to infinity!

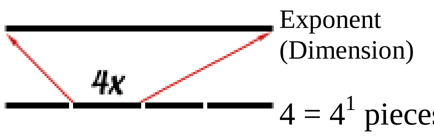


And yet, it is obvious that this "infinitely long" coast of Britain encapsulates only a finite area. Just as a circle drawn on the globe can contain all of Britain . In some way, we believe that the coast of Britain is more "substantial" than a simple circle, and perhaps more interesting than a 1-dimensional line which defines a circle's circumference. Fractal dimension was developed as a way to quantify this contradictory complexity.

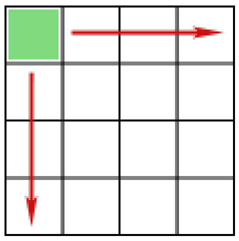
Fractal dimension of a shape is a way of measuring that shape's complexity.

Let's start from classic dimensions like line segment, area and volume.

Notice that a line segment is **self-similar**. It can be separated into **4 smaller replicas of itself**, each 1/4th the size of the original. **Each looks exactly like the original** figure when magnified (scaled) by a factor of 4.

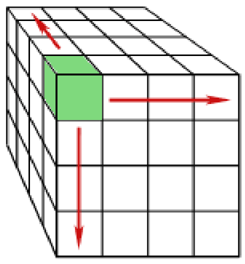


A square can similarly be separated into miniature squares. If the smaller square is scaled 4 times, then it is identical to the larger square; however, observe that here we need $4^2 = 16$ copies of the smaller figure to fill up the original square figure.



$16 = 4^2$ pieces

Following along in this pattern, a cube can be broken down into smaller cubes, each of which can be scaled by a factor of 4 to obtain the larger cube -- but now it takes $4^3 = 64$ smaller cubes to fill up the larger cube.



$64 = 4^3$ pieces

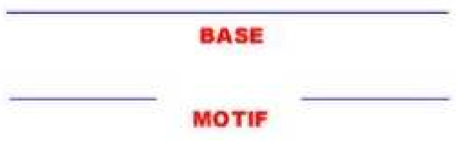
From this, we begin to see a pattern emerge: N, the number of smaller pieces required to "fill up" the original figure is equal to S, the scaling factor, raised to the D power, where D is the dimension of the figure. In the above examples, it is easy to find the dimension simply by reading the exponent:

$$\mathbf{N = S^D}$$

This simple concept can be generalized to measure non-integral dimensions as well. Figures with such fractional dimensions are called (I bet you will guess) - "fractals".

So, give it a try. For example, let there be a line segment. In second step we will split it in 3 segments each one og 1/3 of original lenght. But, instead of just dividing original (base) shape, we shall erase a middle segment. By that we ended 1st iteration, and we ended up with a 2 smaller line segment and a gap – and tha is our motif.

Formulas for creating fractals often involve removing all or part of the previous fractal image.

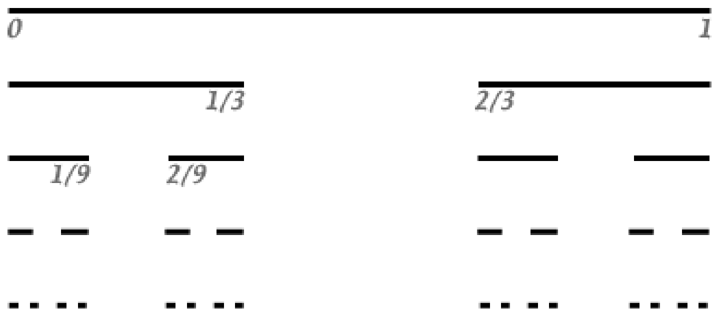


Root

1st iteration

If we continue to iterate on and on we'll get a shape that is not a line segment, neither is a dot.

It is obvious that there are gaps between every smaller and smaller segments. So, intuitively we can conclude that it's dimension is somewhere between 0 and 1. As it is not a integer number we call it fractal dimension.



Level 0

Level 1 – 1st Iteration

Level 2 – 2nd Iteration

Level3 – 3rdt Iteration

Level 4 – 4tht Iteration

By applaying equation D= log(N)/log(S) wich is derived from $N=S^D$ we can calculate that fractal dimension:

1st iteration

• number of smaller segments left in motif	N = 2,
• scaling factor scaled to base	S = 3,
• dimension is	D = log(2) / log(3) = 0.63093

2st iteration

• number of smaller segments left in motif	N = 4,
• scaling factor scaled to base	S = 9,
• dimension is	D = log(4) / log(9) = 0.63093

3st iteration

• number of smaller segments left in motif	N = 8,
• scaling factor scaled to base	S = 27,
• dimension is	D = log(8) / log(27) = 0.63093

Etc..

And by that we've just constructed the Cantor's Dust fractal. Nice and simple example of fractal with a dimension that's very easy to determine. With more complex fractals, determination of dimension will get more complex, and in many cases that will be hard task if not imposible.

So, let's try to determine a dimension of a bit more complex form.

One of the earliest applications of fractals came about well before the term was even used. Lewis Fry Richardson was an English mathematician in the early 20th century studying the length of the English coastline. He reasoned that the length of a coastline depends on the length of the measurement tool. Measure with a yardstick, you get one number, but measure with a more detailed foot-long ruler, which takes into account more of the coastline's irregularity, and you get a larger number, and so on. This was Mandelbrot favorite example, as mentioned earlier.

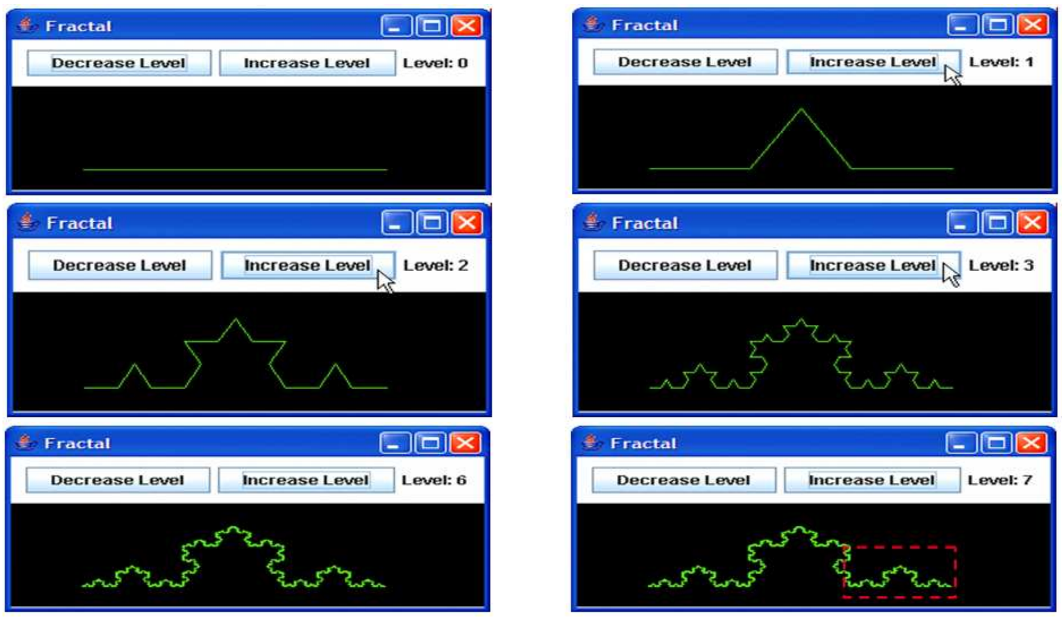
Carry this to its logical conclusion and you end up with an infinitely long coastline containing a finite space, the same paradox put forward by Helge von Koch in the Koch Snowflake. his fractal involves taking a triangle and turning the central third of each segment into a triangular bump in a way that makes the fractal symmetric. Each bump is, of course, longer than the original segment, yet still contains the finite space within. Weird, but rather than converging on a particular number, the perimeter moves towards infinity. Mandelbrot saw this and used this example to explore the concept of fractal dimension, along the way proving that measuring a coastline is an exercise in approximation .

Example – Koch Snowflake

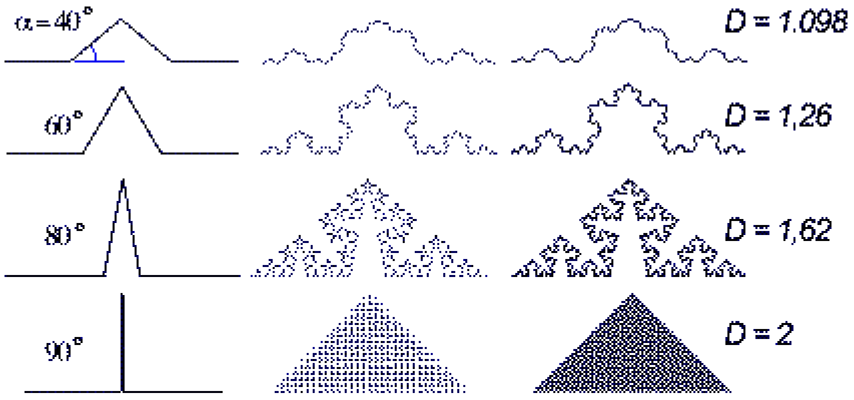
This is popular, strictly self-similar fractal. Fractals have self-similar property in the case that, when subdivided into parts, each resembles a reduced-size copy of the whole. If part is exact copy of original, fractal is said to be strictly self similar.

Koch Snowflake is formed by removing the middle third of each line in the drawing and replacing it with two lines that form a point, such that if the middle third of the original line remained, an equilateral triangle would be formed.

We start with a straight line and apply the pattern, creating a triangle from the middle third. We then apply the pattern again to each straight line, resulting in Level 2 image. Each time the pattern is applied, we say that the fractal is at a new level (or depth, though sometimes the term order is also used). Fractals can be displayed at many (detail) levels. For instance, a fractal at level 3 has had three iterations of the pattern (Level 0) applied.



And it’s is also important to point out that we can get different results even just with a little motif changing. Motif selection is totally at your will, so there is always something to explore even with fractals already explored or published on the web for million times. So let’s see Koch snowflake with different motifs.



Here we can see that a fractal dimension depends on the motif applied, and that is jut about enough about that.

A pure fractal is a geometric shape that is self-similar through different magnification scale, constructed through infinite iterations in a recursive pattern and with complex form measured in fractal dimension that manifest through infinite detail.

(Scary thing is how that actualy make sense....)

Iteration & Recursion

Iteration means the act of repeating a process usually with the aim of approaching a desired goal or target or result. Each repetition of the process is also called an "iteration," and the results of one iteration are used as the starting point for the next iteration.

Recursion is the process of repeating items in a self-similar way. In mathematics and computer science, a class of objects or methods exhibit recursive behavior when they can be defined by two properties:

1) A simple base case (or cases), and
2) A set of rules which reduce all other cases toward the base case.

Iteration and recursion, both :

- are based on a control statement:
  - Iteration uses a repetition structure;
  - recursion uses a selection structure.
- involve repetition:
  - Iteration explicitly uses a repetition structure;
  - recursion achieves repetition through repeated function calls.
- involve a termination test
  - Recursion terminates when a base case is reached
  - Iteration terminates when the loop-continuation condition fails
- Both may occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false;
  - infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; **recursion produces simpler versions of the original problem until the base case is reached** (does this ring any bell?).
- Both techniques have the same complexity - $O(k^n)$,and there exist other algorithms with less complexity (Greedy algorithm, Dynamic programming etc.) - just for info.

Iterative solution

Iteration may refer to the process of iterating a function i.e. applying a function repeatedly, using the output from one iteration as the input to the next. Speaking in general:

- Any problem that can be solved recursively can be solved iteratively
- Iterative solution is easier to construct
- More code
- Accidentally having a nonrecursive method call itself either directly or indirectly through another method can cause infinite recursion.
- Extra non-trivial data structure required (enumerator)

Recursive solutions

Inductive Approach

You have been given a big task, on assumption that you know how to tackle a small one - the basic task. And it is also assumed that you know how to transform a task into a smaller one (that transformation is seen as "some extra action" - usually, that action is defined inductively).

Then, the solution is to successively reduce the task, doing the extra action for every step, until you reach the basic task. In programming, inductive definitions can be directly used to define recursive methods.

Non-Inductive Problems

There are problems without an inductive definition which can best be mapped to an inductive solution. They are of the type "try all possibilities" – "full backtracking".

Recursion vs. Iteration

- A recursive approach is normally preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug.
- It can often be implemented with fewer lines of code. Code much shorter & more elegant
- No extra Enumerator required
- Another reason to choose a recursive approach is that an iterative one might not be apparent.
- Recursion can be expensive in terms of processor time and memory space, but usually provides a more intuitive solution
- Recursion is much harder to understand and verify
- Avoid using recursion in situations requiring high performance. Recursive calls take time and consume additional memory.
- Lots of examples on PSC (shufflin leters, palindromes tester, Prime Factorization Algorithm, folder tree search algorithms etc)

Recursion examples are numerous:

- Find a path through a maze
- Find the minimal roundtrip through a network
- Safely place 8 queens on a chessboard
- Pay a sum with a minimum number of coins, etc

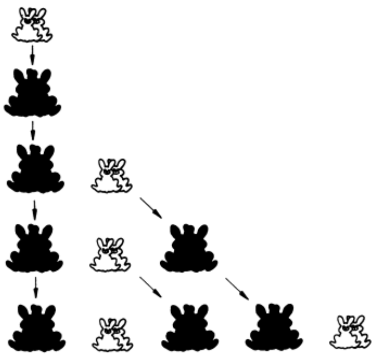
REMEMBER – recursion shoul always converge to a BASE CASE (aiming to / decreasing to simpler or smaller problem).

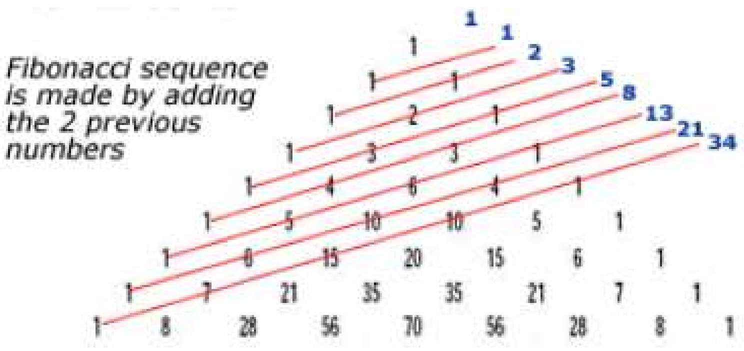
Example -Fibonacci series

Around the year 1200, Leonardo da Pisa (ca. 1175-1250)—better known as "Fibonacci" that is, son of Bonacci (figlio Bonacci) - was considering the problem of how rabbits multiply, something rabbits were obviously very good at even then.

In best modern style he postulated a highly simplified model of the procreation process:

- starting with one immature pair of rabbits
- each season every adult pair of rabbits begets a young pair,
- young pair will be mature one generation later.
- and assuming that rabbits go on living forever, the rabbit population grows rapidly, as shown





More formally, Fibonacci was considering the iterated map 0 -> 1 and 1 -» 10, where 0 stands for an immature rabbit pair and 1 for a mature pair. Thus, the first six generations are represented by the binary sequences

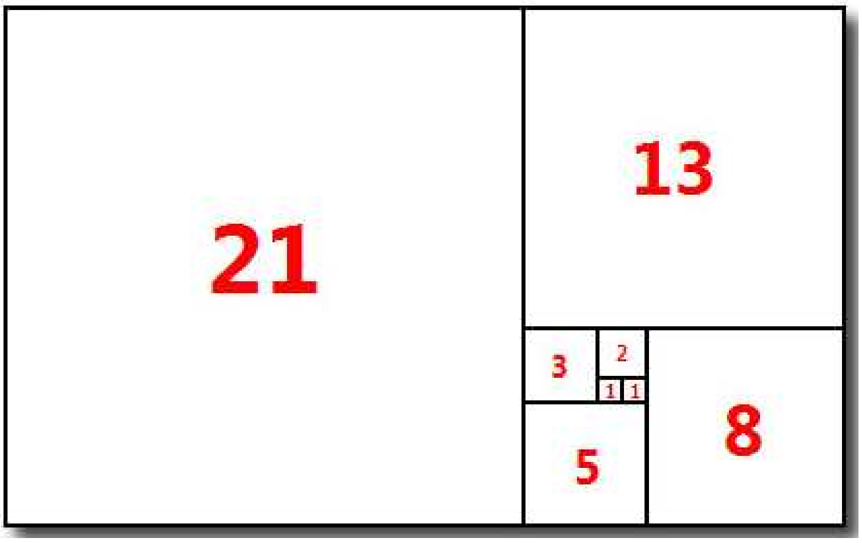
0	
1	-1st pair gets mature, and will produce offspring (“0”) in every furher step)
10	- “10” will by that became constant part of map
101 (1)	- 2nd generation get mature and will produce youngs in every next step
10110	….
10110101	

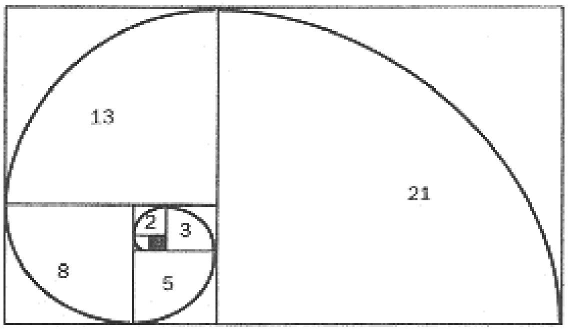
The n-th generation has precisely F_n pairs of rabbits, where F_n is the n-th Fibonacci number defined by $F_1 = F_2 = 1$ and the recursion $F_{n+2} = F_{n+1} + F_n$. This yields the well- known **Fibonacci sequence 0,1,1,2, 3, 5, 8,13, ..(number of mature rabbits)**.

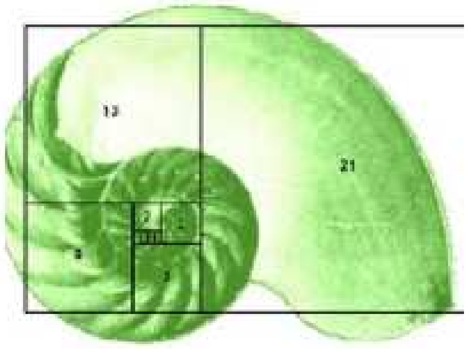
Fibonacci series begins with 0 and 1 and has property that each subsequent Fibonacci number is the sum of previous two Fibonacci numbers. The next number in the sequence is computed simply by adding together the last two numbers.

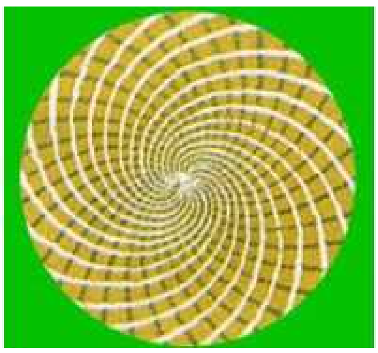
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

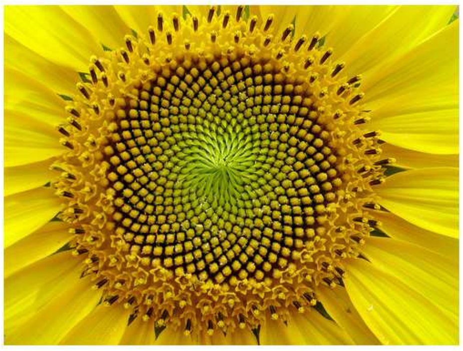
Series occurs in nature, ratio of successive Fibonacci numbers converges on golden ratio or golden mean.


21
13
3
2
1
1
8
5


13
21
2
3
8
5


13
21





Fibonacci – iterative approach version 1

```
Option Explicit

Dim n As Integer:       Dim i As Integer:       Dim Fibonacci() As Single

Private Sub Form_Load()

    n = InputBox("Input the number of member to display ", "Iterative Fibonacci Sequence", 25)

        Label1.Caption = "":        Form1.Caption = "First " & n & " members of Fibonacci"

    ReDim Fibonacci(n):         Call Fibonacci_iter(n)

    For i = 0 To (n - 1)
        Label1.Caption = Label1.Caption & "#" & (i + 1) & "   Fibonacci= " & Fibonacci(i) & vbCrLf
    Next i
End Sub

Private Function Fibonacci_iter(ByVal n As Integer) As Integer

    Fibonacci(0) = 0:   Fibonacci(1) = 1
    'iteration - summation of members up to input limit
    For i = 2 To n
        Fibonacci(i) = Fibonacci(i - 1) + Fibonacci(i - 2)
    Next i
End Function
```

Fibonacci – iterative approach version 2

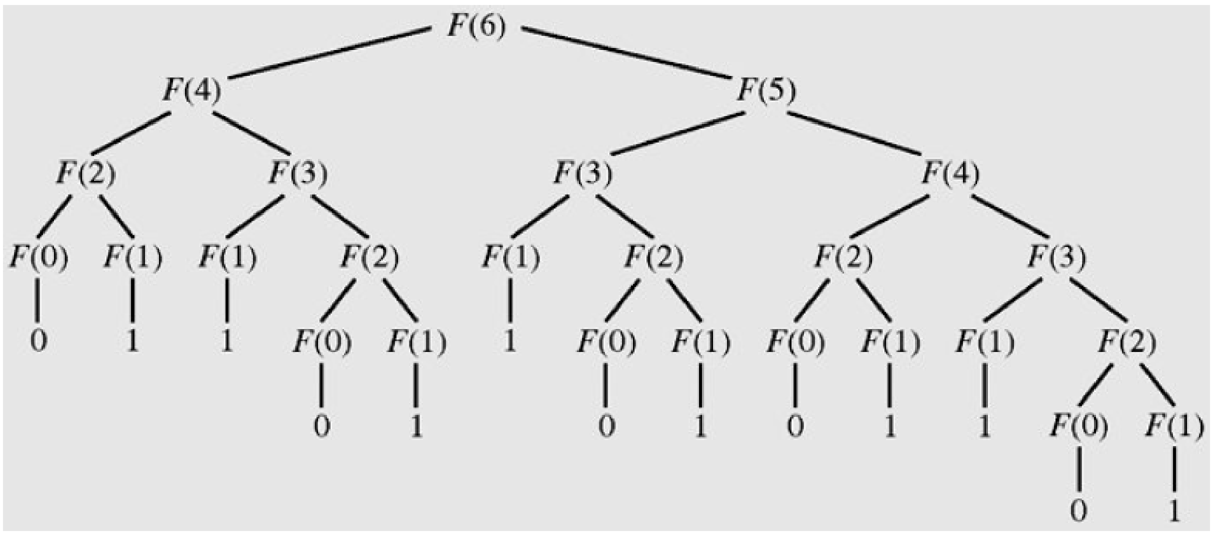
```
Option Explicit

Dim n As Integer:   Dim Fibonacci() As Single

Private Sub Form_Load()

    n = InputBox("Input the number of member to display ", "Iterative Fibonacci Sequence", 25)

    ReDim Fibonacci(n)

    Label1.Caption = "":        Form1.Caption = "First " & n & " members of Fibonacci"

    Call Fibonacci_iter(n)
End Sub

Private Function Fibonacci_iter(ByVal n As Integer) As Integer
    Dim i, p, p0, p1, p2 As Long

    p = 0:      p0 = 0:     p1 = 1

    For i = 1 To n
        Label1.Caption = Label1.Caption & "#" & (i) & "   Fibonacci>= " & p & vbCrLf

        p = p0 + p1

        If (i > 1) Then
            p0 = p1
            p1 = p
        End If
    Next i
End Function
```

Fibonacci – recursive approach

Recursive solution for calculating Fibonacci values results in explosion of recursive method calls. Avoid Fibonacci-style recursive programs, because they result in an exponential “explosion” of method calls.

```
Option Explicit

Dim n As Long:          Dim nr As Single:       Dim FibMember As Single
_______________________________________________________________________________

Private Sub Form_Load()
    Dim x As Integer

    nr = 0:     x = 0:         Label1.Caption = ""
    n = InputBox("Input the number of member to display ", "Recursion Fibonacci Sequence", 25)

    Do Until x = n
        'doing recursion call for each member (recursion for 1st, new recursion for 2nd etc.)
        'explosion of calls and proccessing time is observable for n > 32
        Call Fib(x)
        x = x + 1
        Label1.Caption = Label1.Caption & "#" & x & "  " & FibMember & vbCrLf
    Loop
End Sub
_______________________________________________________________________________

Private Function Fib(n) As Long

    If n <= 1 Then
        Fib = n
    Else
        Fib = Fib(n - 1) + Fib(n - 2)
    End If

    FibMember = Fib
End Function
```

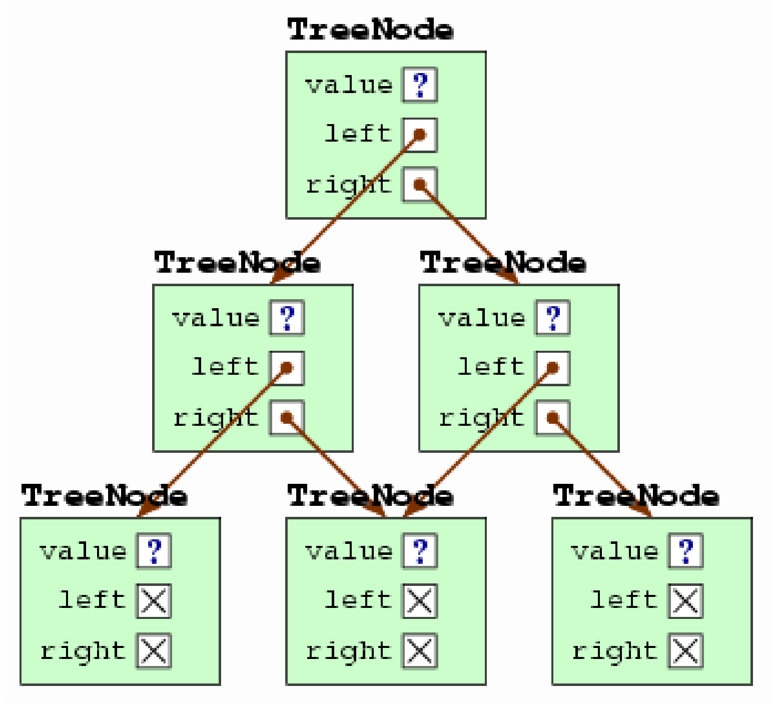
All recursive calls (call tree) for 6th member of Fibonacci sequence.



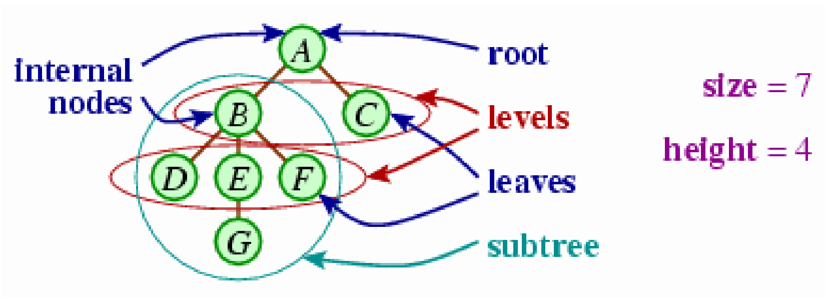
All 3 examples are posted on Planet-Source-Code site.

Trees

A tree is any structure of nodes where no node has multiple references to it. The following figure, for example, is not a tree, since the center node at bottom has two references to it.



General terminology



The node at the tree's top (*A* in the diagram) is called **the tree's root**. (This makes sense when you realize that this is where a natural tree's root system would occur if they didn't grow upside-down.) Each node not connected to anything below it — ***C*****,** ***D*****,** ***F*****, and** ***G*** **in this example — are called leaves**. **All the non-leaf nodes —** ***A*****,** ***B*****, and** ***E*** **here — are called internal nodes.**

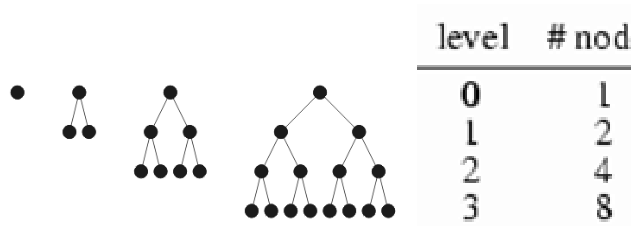
The terminology for talking about relationships between nodes derives from thinking of the tree as a diagram of a person's descendants. We would say that *B* and *C* are children of *A* since the links to them come from *A*. Conversely, *A* is the parent of *B* and *C*. We can extend the terminology to ancestors and descendants: *D*'s ancestors are *A* and *B*, while *B*'s descendants are *D*, *E*, *F*, and *G*.

We'll sometimes talk about the **subtree rooted at a node**, which would consist of that node and all its descendants. The subtree rooted at *B*, for example, would include the nodes in the dashed circle above. For a node with two children, we'll occassionally speak of a node's left subtree or right subtree, which would be the subtree rooted at the relevant child. We would say that the subtree enclosed in the dashed circle is the *left subtree* of *A*.

We will sometimes quantify characteristics of a tree. The most basic measure of a tree is its size, which would be the total number of nodes in the tree (7 here). But we can also talk about the depth of each node, which is the number of ancestors it has. In our diagram, *A* has depth 0, while *B* and *C* have depth 1, and *D*, *E*, and *F* have depth 2. A level is the set of nodes who have the same depth. The overall tree's height is the number of levels — 4 in our example.

Binary tree

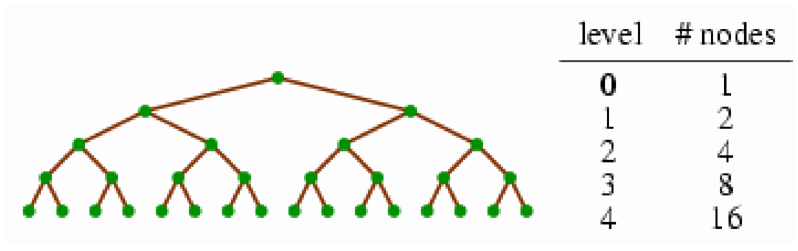
A binary tree is one in which every node has at most two children. When we have a binary tree, we often distinguish a node's two children, calling one the left child and the other the right child.



level	# nod
0	1
1	2
2	4
3	8

Folowing part include some of math, but it is essential to understand algorithms for drawing fractals. Don't be mislead, trees do not govern branching of fractal, that is done by transformation formulas. Trees are governing general shape, meaning number sub-branches that derive from parent level. So if we talk about iterating a point, binary tree will produce 2x more points in child level, and transformations will position them according to fractal's inherent transfomation formula. A ternary tree has 3 time more nodes, but we can use tree of any other higher-order. Of course all tree equations use must be altered accordingly.

Let us define M_h to be the size (total number of nodes) of the complete binary tree with height *h*. We can tabulate the first few values of M_h to get a feeling for how it works.



level	# nodes
0	1
1	2
2	4
3	8
4	16

h	$\boldsymbol{M_h}$
1	1
2	3=1+2
3	7=1+2+4
4	15=1+2+4+8

In a complete (perfect) binary tree, each level has twice as many nodes as the previous level. Thus, we assert, level *k* of a complete binary tree has 2^k nodes (numbering levels starting at root - 0)

We can convince ourselves of this by observing that it is true for $k = 0$ — the root level — where there is just $M_h = 1 = 2^0$ node. And then we observe that if level $k - 1$ has 2^{k-1} nodes, then level k will have $2 \cdot 2^{k-1} = 2^k$ nodes, since each node in level $k - 1$ has exactly two children in level k.

This technique of argument, called an **induction proof**, is particularly convenient when talking about data structures: An induction proof starts by establishing that the assertion is true for a starting case (the **base case**), and then proving the **induction step** — that if the assertion holds starting with the base case and going up to $k - 1$, then the assertion would also hold for k. With both steps completed, then we can conclude that it holds for all integers beyond the base case.

By another induction proof we can prove that number of nodes can be calculated using equaton

$$M_h = 2^h - 1.$$

First the base case: Note that M_1 is 1, which is indeed $2^1 - 1$. Now, for the induction step: Suppose that we already knew that $M_{h-1} = 2^{h-1} - 1$. The last level of the height-h complete tree is level $h - 1$, which we just showed contains 2^{h-1} nodes. Thus, M_h will be 2^{h-1} more than M_{h-1}, and starting from here and simplifying we can conclude that M_h is $2^h - 1$, as we want for the induction step:

$$\begin{aligned} M_h &= M_{h-1} + 2^{h-1} \\ &= (2^{h-1} - 1) + 2^{h-1} \\ &= (2^{h-1} + 2^{h-1}) - 1 \\ &= 2 \cdot 2^{h-1} - 1 \\ &= 2^h - 1 \end{aligned}$$

Since $M_h = 2^h - 1$ is the *maximum* number of nodes in a binary tree with height h, it must be that for any tree, $2^h - 1$ is greater than or equal to n, where h is the height of that tree and n is the number of nodes in that tree:

$$2^h - 1 \geq n$$

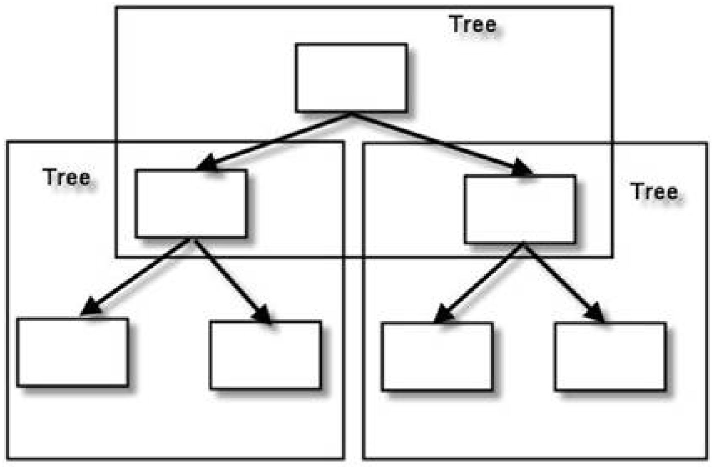
We'll solve for n:

$$2^h \geq n + 1$$
$$h \geq \log_2 (n + 1)$$

This says that a binary tree with n nodes has a height of at least $\log_2 (n + 1)$.

Recursive definition of a perfect binary tree

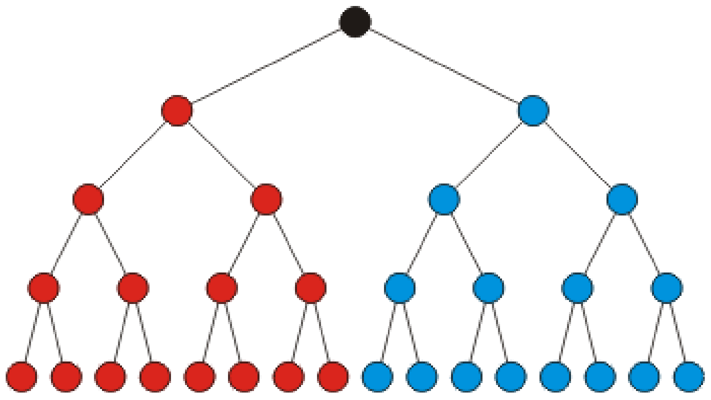
Recursion is generally appropriate and useful when we have a structure that has a similar repeated structural form. As we can see from any tree diagram, a binary tree consists of multiple subtrees (smaller number of nodes comparing to original) which have a similar structure to a binary tree.



A recursive definition of a non-empty perfect binary tree is

- A binary tree of height 1 is perfect, and
- A binary tree of height $h > 1$ is perfect if both the left and right sub-trees are perfect binary trees of height $h - 1$.

Figure bellow gives an example of this recursive defintion.



The number of nodes in a perfect binary tree of height h is given by the formula

$$2^{h+1} - 1$$

Using induction, we can proof that the statement is true when $h = 0$. Assume the statement is true for an arbitrary value h.

If a binary tree of height $h + 1$ is a perfect binary tree, then both subtrees must be of height h. Therefore, by assumption, both sub-trees have $2^{h+1} - 1$ nodes.

Therefore, the number of nodes in a perfect binary tree of height $h + 1$ must be:

$$2(2^{h+1} - 1) + 1$$

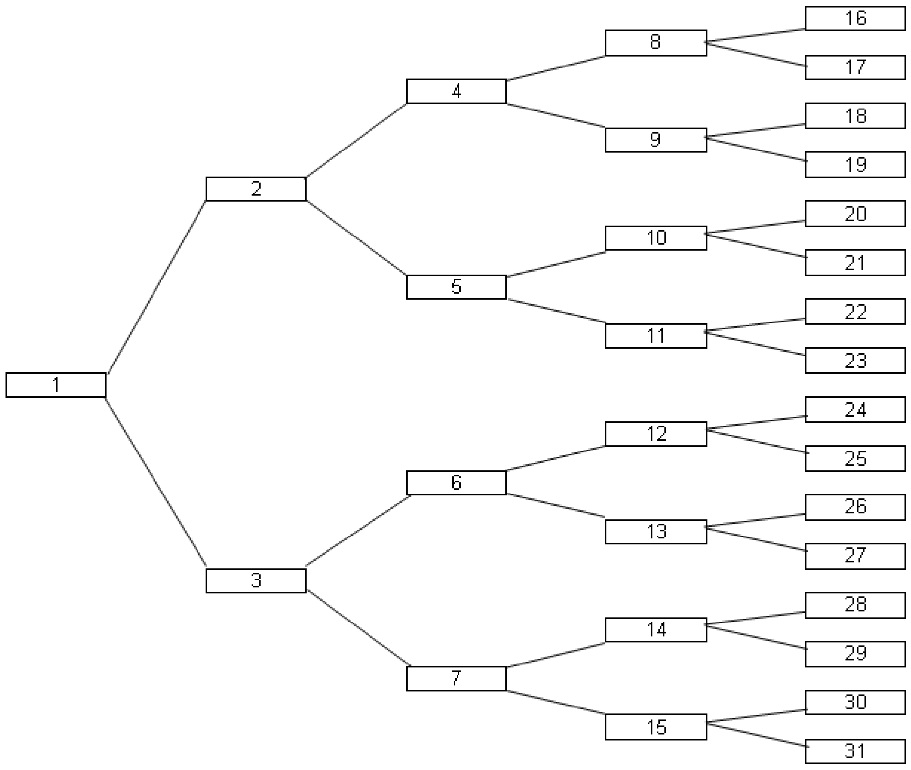
the 1 counting the root node. Simplifying this expression, we get:

$$(2 \cdot 2^{h+1} - 2) + 1 = 2^{h+2} - 1 = 2^{(h+1)+1} - 1$$

Thus, by the process of mathematical induction, the statement is true for $h \geq 0$. Conversely, a binary tree with n nodes must have height $\log_2(n + 1) - 1$.

Recursive Backtracking

Some fractals like Dust or Cloud fractal sets make use of tree. If point 1 is start point of fractal, the all other points (2-31) are points of fractal too (take a closer look on the node numbering).

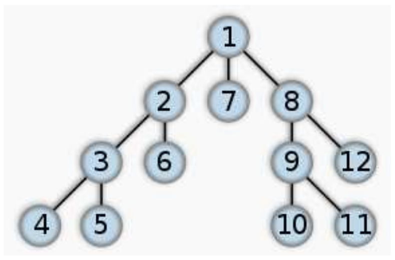


Fractal based on perfect binary tree

order of fractal - number of levels				
p=0	p=1	p=2	p=3	p=4
total points	$k=2^{p+1}-1$			
of fractal to draw	3	7	15	31
Height of tree	$h=\log_2(k+1)$			
h=1	h=2	h=3	h=4	h=5
number of	$M_h=2^h-1$			
nodes	3	7	15	31

Depth-first search and backtracking

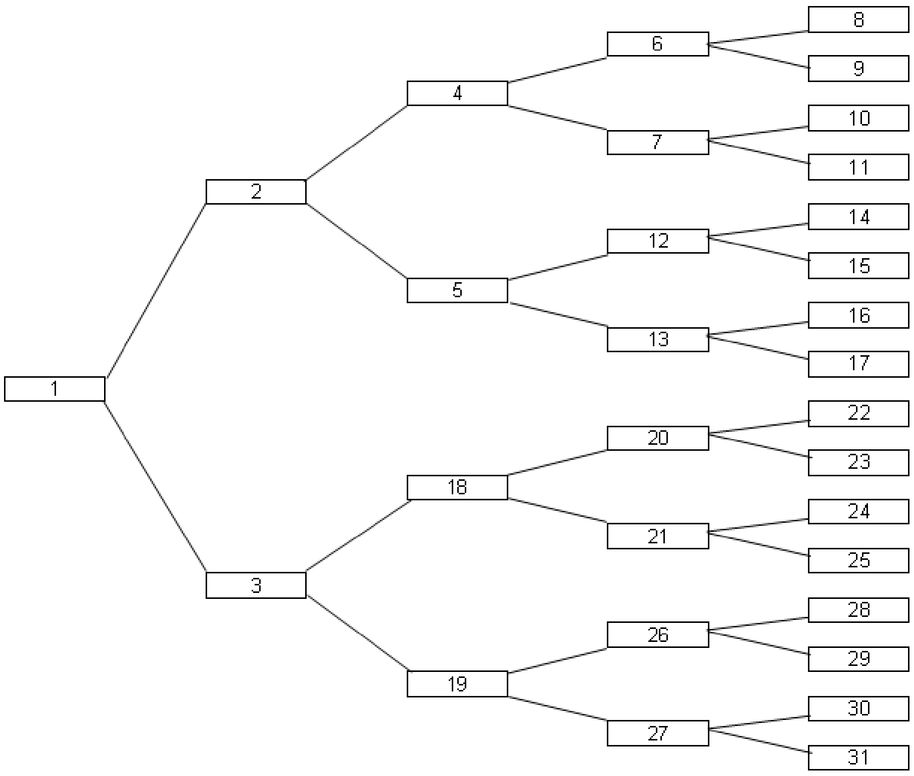
Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking. Each node is numbered according to its access position (1 is accessed 1st, 2 i accessed 2nd et.).



Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Backtracking algorithm

Backtracking is the idea to search all possible solutions paths (or "configurations"), always returning from a dead end to try another path. The backtracking algorithm traverses search tree recursively, from the root down, in depth-first order. Take a look at node numbering again - see a difference.

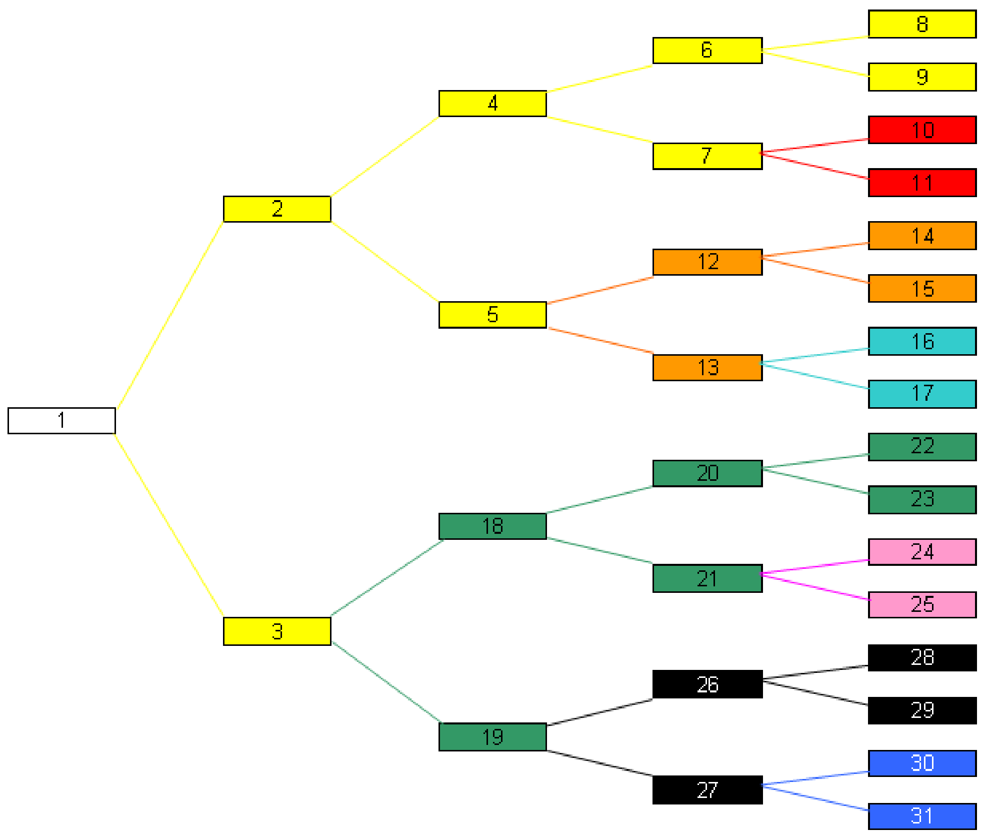


Backtracking is a form of recursion. Conceptually, you start at the root of a tree. At each node, beginning with the root, you choose one of its children to move to, and you keep this up until you get to a leaf.

Recursive Backtracking is a process of using recursion to return to earlier decision point. If one set of recursive calls does result in dead end (leaf), program backs up to previous decision point and makes different decision, often resulting in another set of recursive calls.

The backtracking algorithm is simple but important. You should understand it thoroughly. Another way of stating it is as follows - to search a tree:

1. If the tree consists of a single leaf, test whether it is a goal node,
2. Otherwise, search the subtrees until you find one containing a goal node, or until you have searched them all unsuccessfully.



Fractal based on perfect binary tree

order of fractal - number of levels				
p=0	p=1	p=2	p=3	p=4
total points	$k=2^{p+1}-1$			
of fractal to draw	3	7	15	31
Height of tree	$h=\log_2(k+1)$			
h=1	h=2	h=3	h=4	h=5
number of	$M_h=2^h-1$			
nodes	3	7	15	31
number of groups	$m=2^{p-1}-1$	note - each group is in different color		
[0 - #]	0	1	3	7

(Group contains pairs of points)

Lots of examples on recursion implementation on PSC

- n-Quens,
- Tower of Hanoi,
- Sudoku solver
- Maze solvers, etc..)

Non-recursive backtracking, using a stack

Starting from the root, the only nodes that can be pushed onto the stack are the children of the node currently on the top of the stack, and these are only pushed on one child at a time; hence, the nodes on the stack at all times describe a valid path in the tree. Nodes are removed from the stack only when it is known that they have no goal nodes among their descendents. Therefore, if the root node gets removed (making the stack empty), there must have been no goal nodes at all, and no solution to the problem.

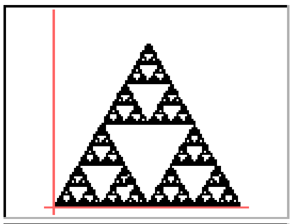
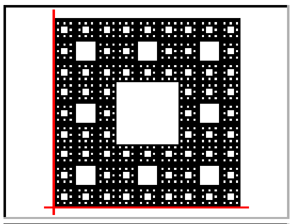
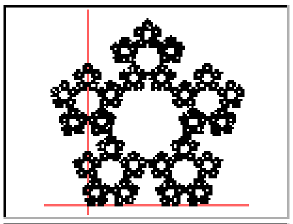
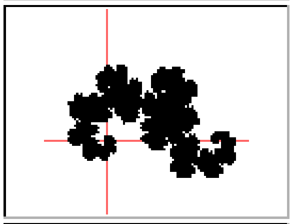
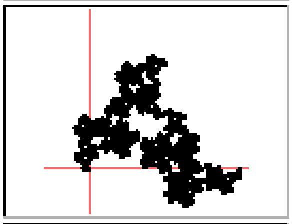
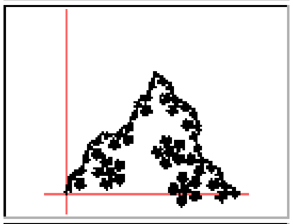
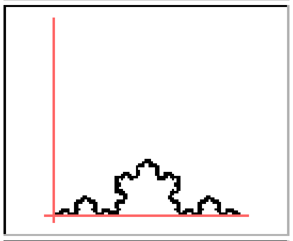
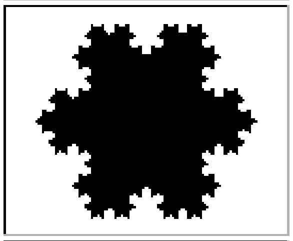
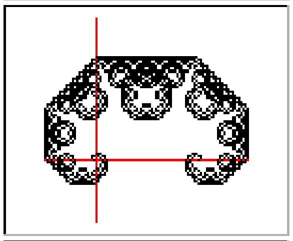
When the stack algorithm terminates successfully, the nodes on the stack form (in reverse order) a path from the root to a goal node.

Similarly, when the recursive algorithm finds a goal node, the path information is embodied (in reverse order) in the sequence of recursive calls. Thus as the recursion unwinds, the path can be recovered one node at a time, by (for instance) printing the node at the current level, or storing it in an array.

Iterated function system

One of the most common ways of generating fractals is as the fixed attractor set of an iterated function system. Each IFS consists of affine transformations involving rotations, scalings by a constant ratio, and translations, and the resulting constructions are always self-similar.

IFS fractals, as they are normally called, can be of any number of dimensions, but are commonly computed and drawn in 2D. The fractal is made up of the union of several copies of itself, each copy being transformed by a function (hence "function system").

		
Sierpinski Gasket	Sierpinski Carpet	Sierpinski Pentagon
		
Heighway Dragon	McWorter Pentigree	Pentadentrite
		
Koch Curve	Koch Snowflake	Levy Dragon

An initial image is transformed by a set of affine transformations (functions) producing a new image. The new image is then transformed by the same affine transformations producing another new image. Thus, each time the image is transformed, an iteration occurs. If the transformation is contractive--that is, the transformation brings points closer together--, then the image will begin to converge. After infinitely many iterations, assuming a contractive transformation, the image will converge to what is called an attractor.

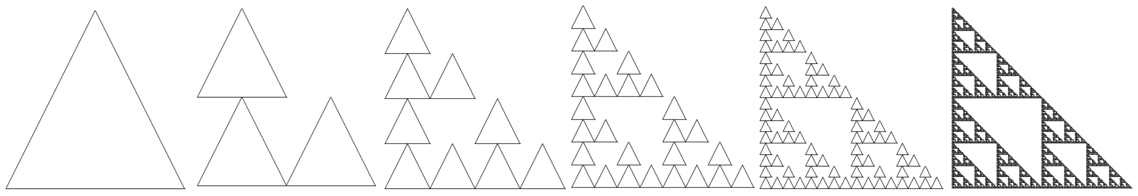
The canonical example is the **Sierpinski gasket** also called the Sierpinski triangle. The functions are normally contractive which means they bring points closer together and make shapes smaller. Hence the shape of an IFS fractal is made up of several possibly-overlapping smaller copies of itself, each of which is also made up of copies of itself, ad infinitum. This is the source of its self-similar fractal nature.

The Sierpinski Gasket can be formed as follows:

1. Begin with an equilateral triangle (although, we can begin with any figure as we will see later).
2. Divide the triangle into four equal-sized triangles.
3. Remove the middle triangle.
4. Go to Step #2.

The above method will form a Sierpinski Gasket. However, it is not an Iterated Function System yet. Let us express this as an Iterated Function System.

1. Begin with an equilateral triangle (again, this is arbitrary)
2. **Reduce the image by one-half.**
3. Make three copies of the reduced image.
4. Align them in the shape of an equilateral triangle.
5. Translate the top copy to the left above the lower-left copy.
6. Go to step #2.



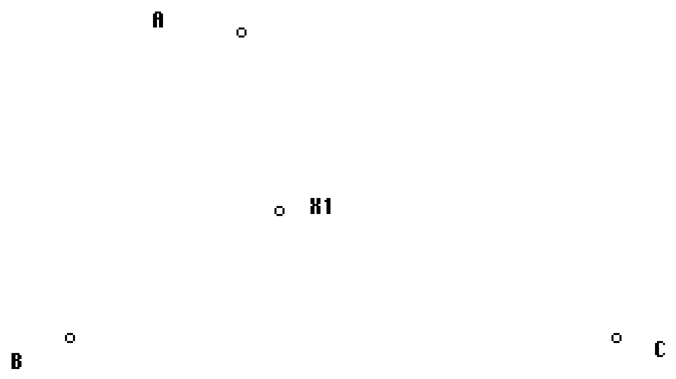
An initial image is transformed by a set of affine transformations (functions) producing a new image. The new image is then transformed by the same affine transformations producing another new image. Thus, each time the image is transformed, an iteration occurs. If the transformation is contractive--that is, the transformation brings points closer together--, then the image will begin to converge. After infinitely many iterations, assuming a contractive transformation, the image will converge to what is called an attractor.

An affine transformation is a recursive transformation of the type

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

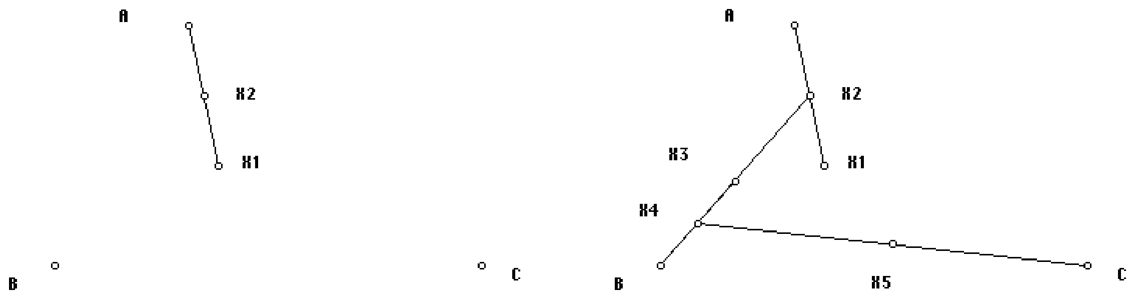
Each affine transformation will generally yield a new attractor in the final image. The form of the attractor is given through the choice of the coefficients *a* through *f*, which uniquely determine the affine transformation. To get a desire shape, the collage of several attractors may be used (i.e. several affine transformations). This method is referred to as an Iterated Function System (IFS).

A number of years ago, Michael Barnsley introduced viewers of public television to the *Chaos Game*. Michael instructed his viewers to plot three arbitrary points A, B, and C on a piece of paper. In addition, he asked his viewers to plot an arbitrary initial point X1, as shown in Figure 1.

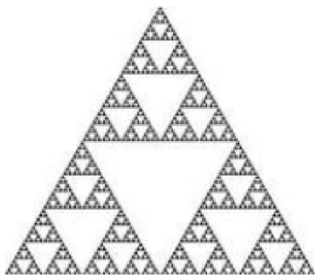


Michael then explained the rules of the game, as follows:

- Roll a three-sided die whose sides are imprinted with the letters A, B, and C.
- If A comes up, plot the midpoint of the segment joining X1 to A. If the point B comes up, plot the midpoint of the segment joining X1 to B. If the point C comes up, plot the midpoint of the segment joining X1 to C. Call the newly plotted point X2 (as shown in Figure 2).
- Roll the die again. Plot the midpoint of the segment joining X2 with either A, B, or C, depending on the outcome of the die toss. Call this new point X3.
- Roll the die again. Plot the midpoint of the segment joining X3 with either A, B, or C, depending on the outcome of the die toss. Call this new point X4.
- Repeat this process indefinitely.



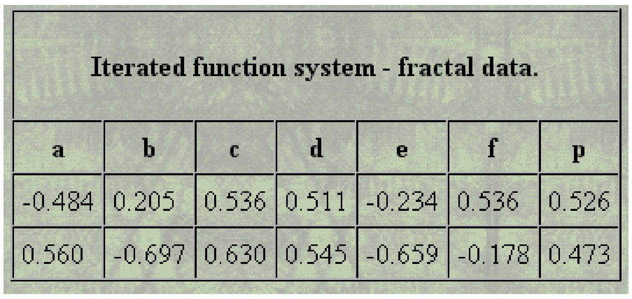
Repeating this iterative process a large number of times, selecting the vertex at random on each iteration, and throwing out the first few points in the sequence, will often (but not always) produce a fractal shape. Using a regular triangle and the factor 1/2 will result in the Sierpinski triangle.



The "chaos game" method plots points in random order all over the attractor. This is in contrast to other methods of drawing fractals, which test each pixel on the screen to see whether it belongs to the fractal. The general shape of a fractal can be plotted quickly with the "chaos game" method, but it may be difficult to plot some areas of the fractal in detail.

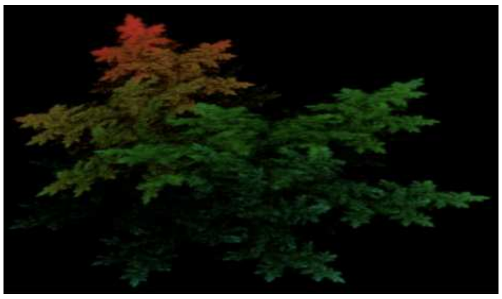
IFS (Iterated Function System) fractals are defined as a set of affine transformations (usually), each assigned a probability value.

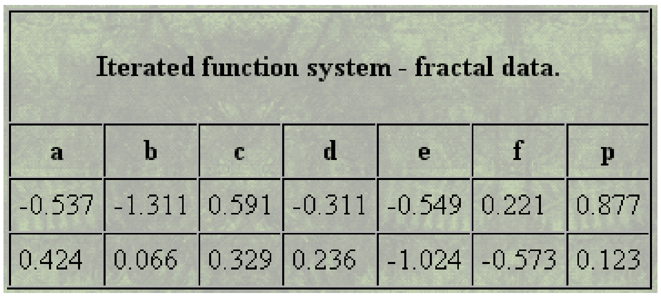
During the fractal iteration, one of the transformations is selected at random based on the assigned probabilities, and the current orbit point is passed through the selected transformation to obtain the next orbit point. In general, in order to produce a fractal, the transformations should be contractions; i.e., when applied to any 2 points, the transformation should reduce the Euclidean distance between the points.



Iterated function system - fractal data.

a	b	c	d	e	f	p
-0.484	0.205	0.536	0.511	-0.234	0.536	0.526
0.560	-0.697	0.630	0.545	-0.659	-0.178	0.473





Iterated function system - fractal data.

a	b	c	d	e	f	p
-0.537	-1.311	0.591	-0.311	-0.549	0.221	0.877
0.424	0.066	0.329	0.236	-1.024	-0.573	0.123