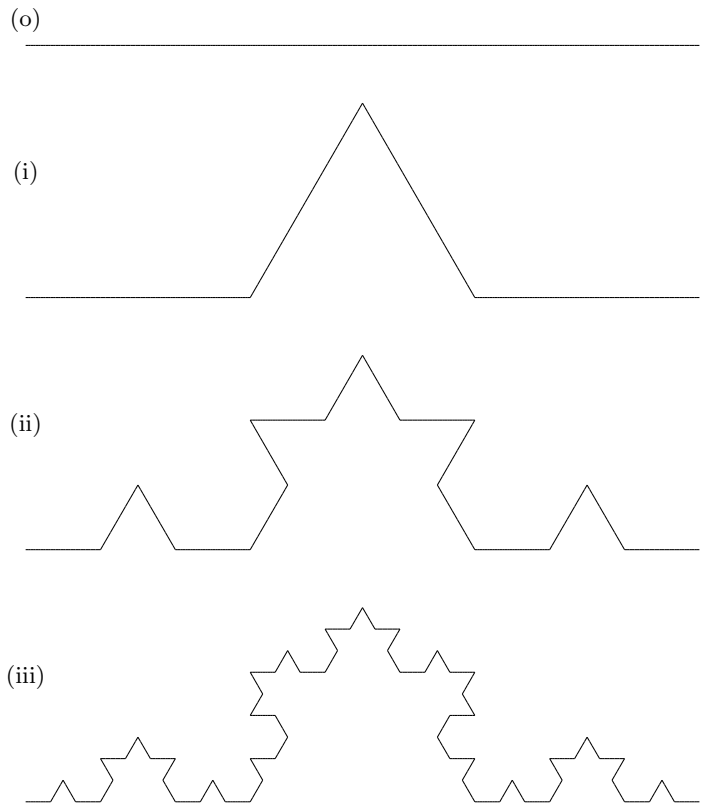
ペアノ曲線

相似な図形の基本は拡大・縮小である。それを少し発展させて、図形の一部を一定の規則で変形することを繰り返すと**自己相似形**と呼ばれる図形ができる。**コッホ曲線**[1]はそのような図形の一つである。コッホ曲線は次のようにして作成される。



一定の長さの線分 (o) を 3 等分し、中央に 3 等分線の長さを一辺とする正三角形を描き、下辺を取り除く`..(i)`。(i) の新たな 4 辺それぞれに対し、(i) の操作を行う`..(ii)`。(ii) の新たな 16 辺それぞれに対し、(i) の操作を行う`..(iii)`。以下、新たな辺に対し (i) の操作を行うことを続ける。この操作を無限に繰り返してできる図形がコッホ曲線である。

このように自己相似形となる図形は**フラクタル図形**ともいう。自己相似であることは、たとえば (ii) の図形全体を 1/3 に縮小した図形が、(iii) の「左–中央左–中央右–右」へ複製されていることからわかる。このことは (iv) 以降も繰り返される。多くは再帰関数を利用して描画できるが、そうでないフラクタル図形もある。フラクタル図形はさまざまなものが考案されているので、興味があれば調べてみるとよいだろう。

[1] ヘルゲ・フォン・コッホ (1870–1924)：スウェーデンの数学者。

2

さて、コッホ曲線はその作成の仕方から、再帰を利用してプログラムが書ける。ただ、正三角形の頂点を求めるのは三角比が関連するので、ここでは座標計算がより簡単な**ペアノ曲線**[2]に取り組むことにしたい。ただしペアノ曲線は、最初の線の引き方によりいくつかのタイプがあるようだ。また再帰を考える際、4 通りの状況を考える必要があるため、多少込み入った部分もあることに注意されたい。

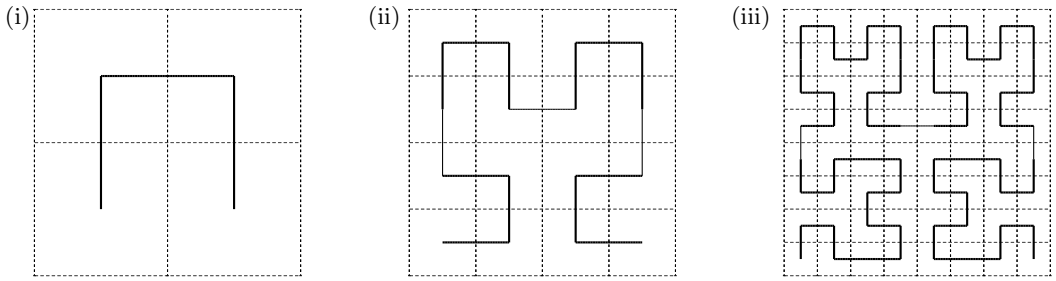
[peanocurve.playground]

```
 1 import SwiftUI
 2
 3 struct ContentView: View {
 4     // @State private var end = 0.0            // for animation
 5
 6     var body: some View {
 7         Path { path in
 8             @State var SIZE  = 80.0
 9             @State var nodes = divlayer(n: 3)
10
11             for i in 0 ..< (nodes.count-1) {
12                 path.move(   to: CGPoint(x: SIZE * CGFloat(nodes[i  ].0),
                                            y: SIZE * CGFloat(nodes[i  ].1)))
13                 path.addLine(to: CGPoint(x: SIZE * CGFloat(nodes[i+1].0),
                                            y: SIZE * CGFloat(nodes[i+1].1)))
14             }
15         }
16         // .trim(from: 0.0, to: end)           // for animation
17         .stroke(lineWidth: 1)
18         .fill(Color.red)
19         .frame(width: 1, height: 1)
20         // .onAppear{ end = 1.0 }              // for animation
21         // .animation(.linear(duration: 3), value: end)    // *
22     }
23
24     func divlayer(n: Int) -> [(Float, Float)] {
25         var h4: [(Float, Float)] = []
26
27         if n > 0 {
28             let hf = half(c: divlayer(n: n-1))
29             for i in (0 ..< hf.count).reversed() {
                                        h4.append(( hf[i].1-1, -hf[i].0-1)) }
30             for i in  0 ..< hf.count             {
                                        h4.append(( hf[i].0-1,  hf[i].1+1)) }
31             for i in  0 ..< hf.count             {
                                        h4.append(( hf[i].0+1,  hf[i].1+1)) }
32             for i in (0 ..< hf.count).reversed() {
                                        h4.append((-hf[i].1+1,  hf[i].0-1)) }
33             return h4
```

[2]ジュゼッペ・ペアノ (1858–1932)：イタリアの数学者。

```
34          } else {
35              return [(0, 0)]
36          }
37      }
38
39      func half(c: [(Float, Float)]) -> [(Float, Float)] {
40          var h: [(Float, Float)] = []
41
42          for i in 0 ..< c.count { h.append((c[i].0 / 2, c[i].1 / 2)) }
43          return h
44      }
45  }
```

プログラムの説明の前に、このプログラムが想定しているペアノ曲線がどのように作られるかを述べよう。まず、均等に 4 分割された正方形の領域において、各領域の中心を結ぶ線分を「左下–左上–右上–右下」へ描く`..(i)`。



各領域をさらに 4 分割すると、各々の領域は (i) の 1/2 の大きさになるので、(i) で描いた図形を 1/2 の大きさで「(−90° 回転で) 左下–左上–右上–(90° 回転で) 右下」へ描く。この際、各々の領域の図形は線分（図では細い線で描いた）で結ばれる`..(ii)`。

同様に、各領域をさらに 4 分割すると、各々の領域は (ii) の 1/2 の大きさになるので、(ii) で描いた図形を 1/2 の大きさで「(−90° 回転で) 左下–左上–右上–(90° 回転で) 右下」へ描く。この際、各々の領域の図形は線分で結ばれる`..(iii)`。(iv) 以降も同様に繰り返す。

この手順をプログラムしたのが `peanocurve.playground` である。まず、領域を分割するごとに大きさが半分になるので、そのときの座標が一つ前の座標の半分になることを反映した関数 `func half(c: [(Float, Float)]) -> [(Float, Float)]` を用意している。この関数は単に座標データを半分にするもので、もとのデータを一度だけ受け取る仕様であるから再帰関数ではない。

再帰関数は `func divlayer(n: Int) -> [(Float, Float)]` で、4 分割の階層を整数値 `n` で受け取って、一つ前の階層をもとに 4 分割された領域にそれぞれ 1/2 サイズの複製の座標データを格納する。

最初は $-90°$ 回転をして左下の領域に複製する。ドラゴンカーブで述べたことを参照すれば、$-90°$ 回転した座標 $(\dot{x},\ \dot{y})$ は $(\dot{x},\ \dot{y}) = (y,\ -x)$ の関係にある。関数 `divlayer` が、座標を格納する配列 `h4` に与える初期値は `(0, 0)` で、ここに `h4.append(( hf[i].1-1, -hf[i].0-1))` としてデータを追加する理由は、(i) の図形の描き始めの座標を原点から $(-1,\ -1)$ の位置にあるものとして見ているからである。つまり、(i)–(無限) のいずれにおいても全体領域の中心座標は $(0,\ 0)$、左下 $1/4$ の領域の中心座標は $(-1,\ -1)$ ということである。このとき、`for` 文が `.reversed()` メソッドで逆読みしているのは、ドラゴンカーブの扱い同様、回転したことによる。

続く 3 個の `for` 文が、順に左上、右上、右下であることは、x 座標、y 座標の ± 1 の具合でわかるであろう。

このままでは実際の図形表示は 4×4 の大きさになってしまうので、全体領域の `SIZE` を `80.0` にしたのはドラゴンカーブでの処置と同じである。蛇足ながら、単に `80` で代入すると `CGFloat` 型にならないので、SwiftUI から修正を促される。`divlayer(n: 3)` の値を変えて、ペアノ曲線が自己相似形で増えていく様子を見るとよいだろう。

このプログラムをもとにすればコッホ曲線なども表示できるので、いろいろなフラクタル図形を描画するのは君たちの仕事となる。

◇　◇　◇

プログラム `peanocurve.playground` には、コメントアウトした行（4, 16, 20, 21 行目）がある。これはアニメーション用に用意したもので、4 行すべてを有効にすればペアノ曲線が描画される際、アニメーションで見られるようになる。描画する速度は `duration:` の値で調整してもらいたい。

ドラゴンカーブのプログラムにはアニメーション用のコードを入れてないので、このプログラムと同じように 4 行のコードを追加しよう。ドラゴンカーブの描画の様子がアニメーションとして見られるようになる。