

١ وازرميات العودية

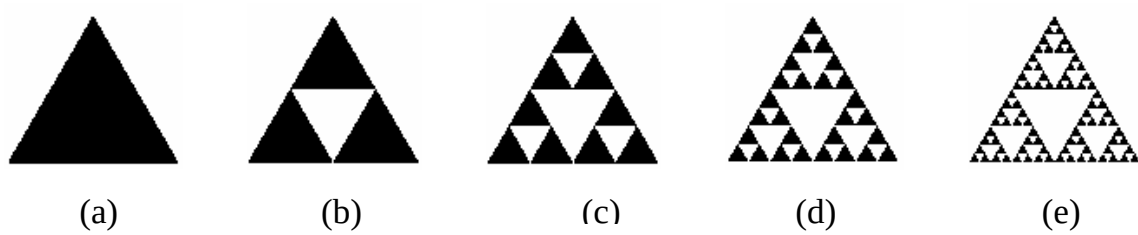
Recursive Algorithms

1. مقدمة

تُعتبر المسائل العودية من المسائل التي تتطلب منهجية تفكير خاصة، ولذلك سوف نسعى إلى عرض هذه المنهجية من خلال بعض الأمثلة.

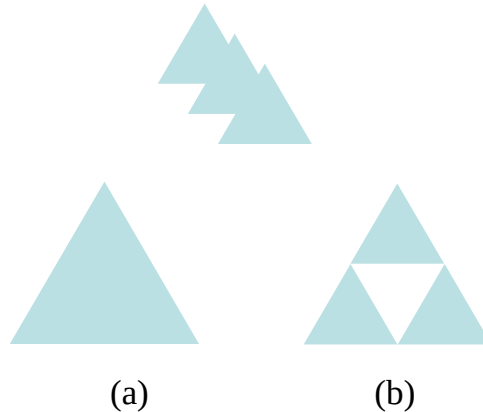
المثال الأول: مثلث سيربنسكي

انظر إلى الأشكال التالية :



ولنحاول إيجاد العلاقة بين هذه الأشكال.

لاحظ أن الشكل الأول (a) عبارة عن مثلث متساوي الأضلاع، وأن الشكل (b) عبارة عن ثلاث أشكال (مثلثات) من النوع (a) تم تصغيرها إلى النصف ووضعها بحيث يشترك كل شكلين (مثلثين) برأس واحد، وكذلك الشكل (c) يتألف من ثلاثة أشكال من النوع (b) تم تصغيرها إلى النصف ووضعها بحيث يشترك كل شكلين برأس واحد، وإذا تابعت بنفس الأسلوب ستكتشف أن الشكلين (d) و (e) تم رسمهما باستخدام القاعدة نفسها.



المثال الثاني: متتالية فيبوناتشي

في عام 1202 اهتم العالم فيبوناتشي Fibonacci بحساب عدد الأرانب التي ستكون في مزرعة ضمن الفرضيات التالية :

نبدأ بزوج واحد من الأرانب (ذكر وأنثى) حديثي الولادة

تصبح الأرانب قادرة على الإنجاب عندما يصبح عمرها شهراً واحداً

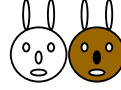
بعد أن يبدأ زوج الأرانب بالإنجاب فإنه ينجب في كل شهراً زوجاً واحداً من الأرانب (ذكر وأنثى)

الأرانب لاتموت خلال مدة الحساب

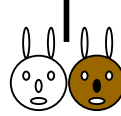
يبين الشكل التالي "أجيال" الأرانب وأعداد الأزواج في كل شهر

mon	Nb.
-----	-----

1	1
---	---



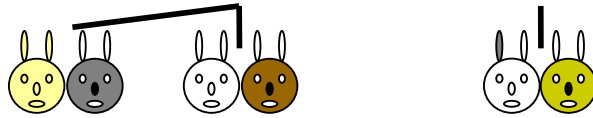
1	1
---	---



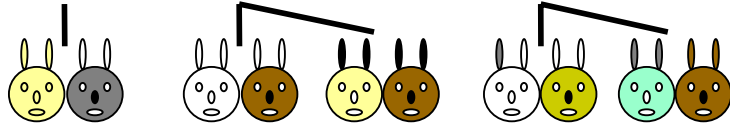
2	2
---	---



3	3
---	---



4	5
---	---



ولو تابعت لابد أنك ستلاحظ أن عدد الأزواج يأخذ القيم التالية :

عدد الأزواج	رقم الشهر
1	0
1	1
2	2
3	3
5	4
8	5
13	6
21	7
34	8
55	9
89	10
144	11
233	12

ومن هنا استنتج فيبوناتشي القاعدة العامة لحساب المتالية الشهيرة التي حملت اسمه، والتي تتلخص
بمايلي:

بفرض $F(n)$ هو عدد الأزواج في الشهر رقم n ، فإن

$$F(0) = 1$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

لاحظ أن تعريف $F(n)$ تم بدلالة $F(n-1)$ و $F(n-2)$ في الحالة العامة، وهذا التعريف هو تعريف
عودي.

2. تعريف

نقول عن شيء إنه ذو بنية عودّية (Recursive) إذا كان مؤلفاً من مجموعة من المكونات بعضها
معرف تعريف الشيء الأصلي.

نقول مثلاً إنّ الشجرة هي بنية عودّية لأنها مؤلفة من مجموعة من الفروع، كل منها يحوي فروعاً
جديدة، ومن ثمّ يكون له بنية الشجرة نفسها أو يحوي مجموعة من الأوراق.

من الأمثلة الشهيرة لاستخدام العودّية في التعاريف نورد ما يلي:

1- تعريف الأعداد الطبيعية: تُعرّف الأعداد الطبيعية كما يلي:

العدد صفر هو عدد طبيعي

إذا كان n عدداً طبيعياً فإن $n+1$ عدد طبيعي أيضاً.

2- تعريف بنية الشجرة الثنائية: تُعرّف الشجرة الثنائية كما يلي:

الشجرة $\emptyset$ لا تحوي أي عقدة، نسميها الشجرة الفارغة.

إذا كانت A و B شجرتين ثنائيتين فإن $\langle X, A, B \rangle$ هي شجرة ثنائية جذرها X وشجرتها الجزئية اليسارية A وشجرتها الجزئية اليمينية B .

3- تعريف العامل لعدد صحيح غير سالب: يعرف العامل كما يلي:

$$0! = 1$$

إذا كان $n > 0$ فإن $n! = n \cdot (n - 1)!$

4- تعريف متتالية أعداد فيبوناتشي (Fibonacci): تُعرّف هذه المتتالية بالشكل التالي:

$$\text{Fib}_0 = 0, \text{Fib}_1 = 1$$

إذا كان $n > 1$ فإن $\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$

تكمّن أهمية العودية في إمكان تعريف مجموعة غير منتهية من الأشياء، بواسطة مجموعة منتهية من التعليمات. بالطريقة نفسها يمكن تعريف عمليات حسابية غير منتهية بواسطة خوارزمية عودية، تكون الخوارزميات العودية أكثر ملاءمة عندما تكون المسألة المطلوب حلها أو بنية المعطيات الواجب معالجتها معروفة تعريفاً عودياً.

ملاحظة هامة:

في التطبيقات العملية يجب أن يكون عمق العودية منتهياً وصغيراً أيضاً، لأن كل استدعاء عودي يتطلب تخزين وسائط الإجرائية ومتحولاتها الموضعية وحالة الإجرائية، وذلك من أجل استعادة هذه المعطيات عند انتهاء الاستدعاء. على هذا، كلما كان عمق العودية صغيراً كان حجم التخزين صغيراً. وتجدر الإشارة إلى أنه في حالات عدم الانتباه لهذا العمق، يؤدي تنفيذ البرنامج العودي إلى طفح في مكدس التنفيذ (Stack Overflow) ومن ثم إلى توقف البرنامج فوراً، قبل الانتهاء من التنفيذ. من المفيد أن نلاحظ أيضاً أنه حتى في حالة التعريف العودي للمسألة أو بنية المعطيات، قد لا تكون العودية هي الحل الأفضل.

مثال: أعداد فيبوناتشي

التعريف عوديّ والحل بواسطة إجرائية عوديّة مباشر:

```
public static int fibrec(int n)
{
    if (n < 2)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

لاحظ أنه باستخدام هذه الإجرائية العودية لحساب fib_n نحتاج إلى $1-2^{n-1}$ استدعاء عوديّ، وإذا ينمو عدد الاستدعاءات أسياً ويعتبر هذا غير فعال من الناحية العملية، على حين نستطيع إيجاد إجرائية غير عوديّة (تكرارية) خطية للحل:

```
public static int fib(int n)
{
    int i=1, m=0;
    if (n<2)
        return n;
    else
        for (int j = 1 ; j<n;j++)
        {
            int temp = i+m;
            m = i;
            i=temp;
        }
    return i;
}
```

إذن يجب تجنّب استعمال العوديّة عندما يمكن استعمال خوارزمية تكرارية (Iterative Algorithm) مكافئة. ولكن ليس في كل الحالات يمكن تحويل العوديّة إلى تكرار بسيط بسهولة (انظر الأمثلة في فقرة "النتهاء تنفيذ إجرائية عوديّة")، ثم إن عملية التحويل العامة من عوديّة إلى تكرارية قد تكون معقّدة (انظر فقرة "أمثلة لبرامج عوديّة").

3. المخطط العام لخوارزمية عوديّة

يمكن بوجه عام التعبير عن خوارزمية عوديّة P بواسطة تركيب C لمجموعة عمليات أساسية Si (لا تحوي P) مع P نفسها:

$$P \equiv C[Si, P]$$

إنَّ الأداة اللازمة والكافية للتعبير عن برنامج معيّن تعبيراً عودياً هي الإجرائية (Procedure) أو التابع (Function) لأنها تسمح بإعطاء اسم معين لمجموعة تعليمات، وهذا ما يسمح باستدعاء هذه التعليمات استدعاءً عودياً.

يمكن التمييز بين نوعين من الإجرائيات العودية:

الإجرائيات ذات العودية المباشرة: نقول عن إجرائية P إنها عودية مباشرة إذا كانت تحوي استدعاءً صريحاً لنفسها.

الإجرائيات ذات العودية غير المباشرة: نقول عن إجرائية P إنها عودية غير مباشرة إذا كانت تستدعي إجرائية أخرى Q تستدعي P (بطريق مباشر أو غير مباشر).

4. انتهاء تنفيذ إجرائية عودية

في كل إجرائية تحوي تكراراً لتعليمات معينة، يجب الانتباه إلى أنَّ هذا التكرار سينتهي بعد حد معين. إنَّ أولى الاحتياطات الواجب اتخاذها هي جعل الاستدعاء العودي للإجرائية مشروطاً بتحقيق قضية معينة. بهذا يمكن أن نعتبر أن المخطط العام لإجرائية عودية يأخذ أحد الشكلين:

$$P \equiv \text{if } B \text{ then } C[\text{Si}, P]$$

أو

$$P \equiv C[\text{Si}, \text{if } B \text{ then } P]$$

للبرهان على انتهاء إجرائية عودية يعرف تابع $P(X)$ حيث X مجموعة معاملات الإجرائية. نعرف التابع $P(X)$ بحيث تكون القضية B مكافئة لـ $P(X) > 0$ وفي المرحلة التالية نبهرن أنَّ $P(X)$ يتناقص عند كل استدعاء عودي.

مثال

لنأخذ إجرائية حساب العامل:

```
public static int fact(int n)
{
    int f;
    if (n == 0 )
        f = 1;
    else
        f = n*fact(n-1);
    return f;
}
```

في هذه الحالة مجموعة المعاملات هي $X = \{n\}$ ، إذا عرّفنا التابع $P(n) = n$ فإن شرط الاستدعاء العودي هو $P(n) > 0$.

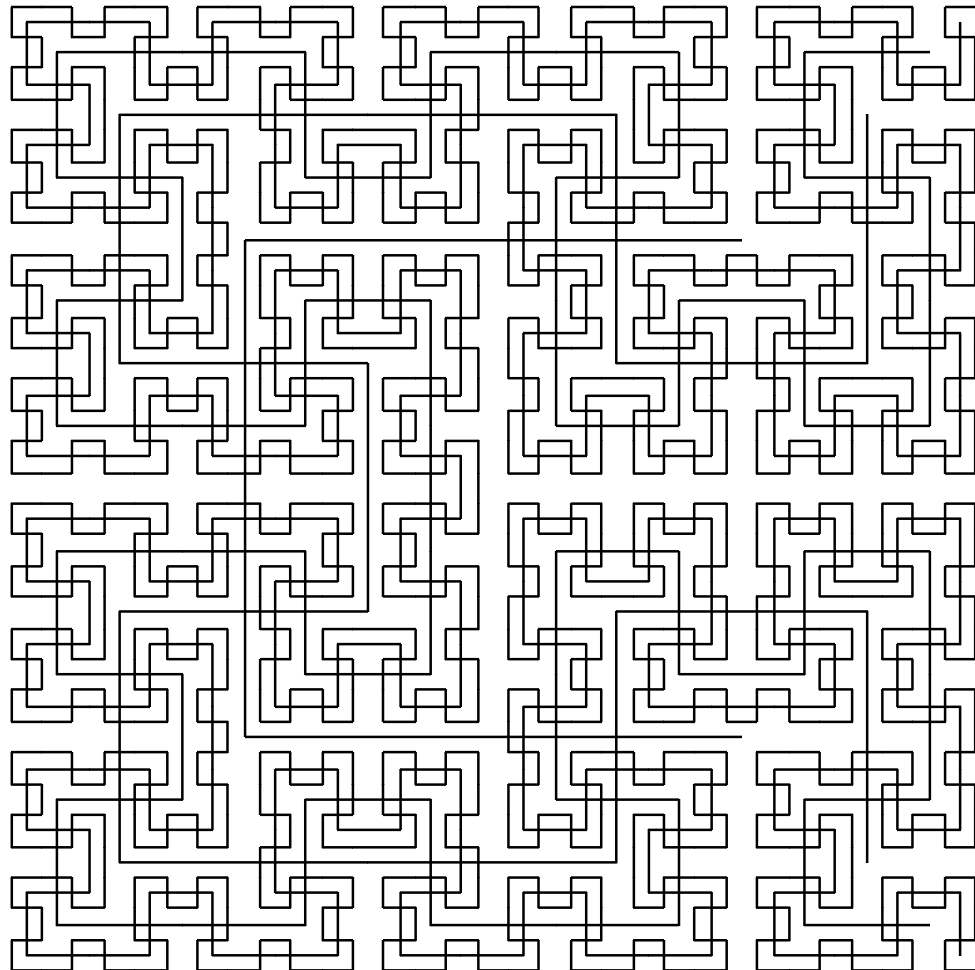
نلاحظ من نص الإجرائية أنّ التابع يتناقص بمقدار واحد عند كل استدعاء عودي.

5. أمثلة لبرامج عودية

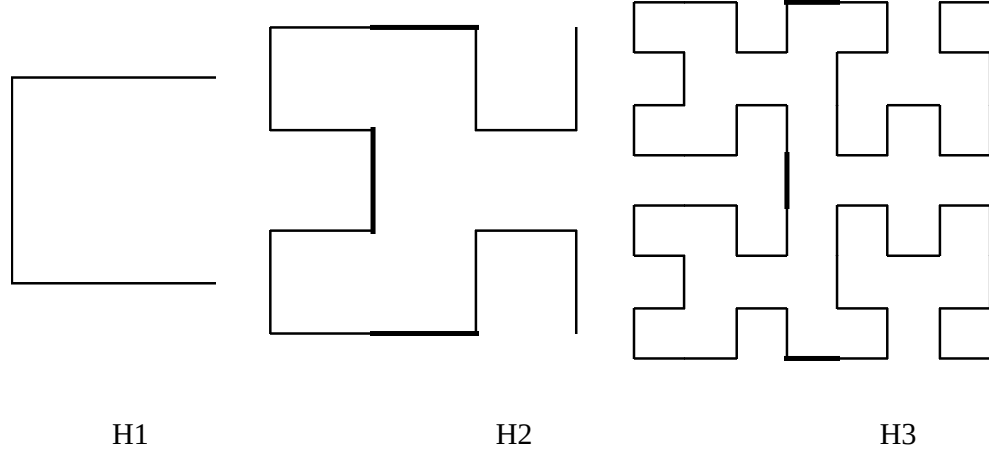
نعرض فيما يلي أمثلة على استخدام البرامج العودية في مسائل قد يكون من المعقد حلها ببرامج تكرارية.

5.1 منحنيات هيلبرت

إذا نظرنا بإمعان إلى الأشكال الزخرفية المبينة في الشكل التالي نرى أنها تتألف من خمسة خطوط منكسرة تتبع منهجاً في الرسم نستطيع برمجته بعد تحديده بدقة.



لو نظرنا مرة أخرى إلى هذه الخطوط ، وحاولنا عزل الثلاثة الأولى منها، نجد أن لها شكلاً مبيّناً بالشكل التالي نرمرز إلى هذه الأشكال بـ H_1 ، H_2 ، H_3 على الترتيب.

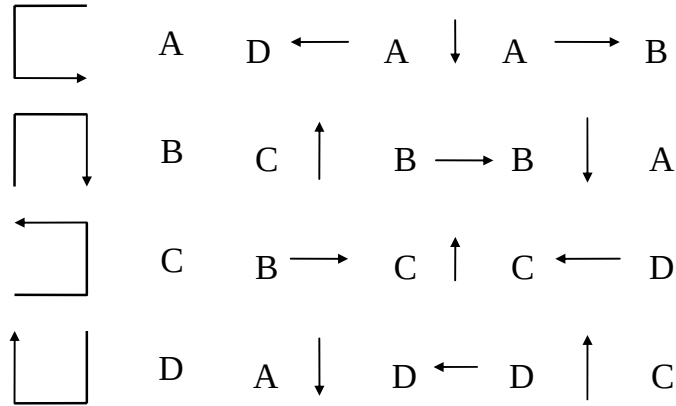


يظهر هذا الشكل كيف نستطيع الحصول على المنحني H_{i+1} بتركيب أربعة منحنيات H_i أصغر حجماً بعد تدويرها باتجاهات معينة، والوصل بينها بثلاث قطع مستقيمة رسمناها بخط عريض. لنفترض أن الأدوات الأساسية المتوفرة للرسم هي:

جملّة إحداثيات xOy

إجرائية $stepNorth()$ تقوم برسم قطعة مستقيمة من الموقع الحالي (x,y) إلى الموقع $(x,y+h)$
 إجرائية $stepSouth()$ تقوم برسم قطعة مستقيمة من الموقع الحالي (x,y) إلى الموقع $(x,y-h)$
 إجرائية $stepEast()$ تقوم برسم قطعة مستقيمة من الموقع الحالي (x,y) إلى الموقع $(x+h,y)$
 إجرائية $stepWest()$ تقوم برسم قطعة مستقيمة من الموقع الحالي (x,y) إلى الموقع $(x-h,y)$

لما كان كل منحني H_i يتألف من أربعة منحنيات H_{i-1} متصلة، فإننا نستطيع التعبير عن إجرائية رسم H_i كتركيب لأربع إجرائيات تقوم كل منها برسم H_{i-1} بالاتجاه والقياس المناسبين. إذا رمزنا إلى هذه الإجرائيات الأربع بالأحرف A, B, C, D وإذا رمزنا بأسهم إلى عمليات رسم الوصلات، فإننا نستطيع التعبير عن المخطط العودي لهذه الإجرائيات كما يلي:



إذا كانت h طول القطعة المستخدمة في رسم الخط المنكسر، فإن كتابة الإجراءية A ستأخذ التركيب التالي لـ D و B مع A:

```
private void HilbertNorth(int level)
{
    if (level <= 0)
    {
        HilbertLevel0();
        return;
    }

    int nextlevel = level - 1;

    HilbertEast (nextlevel);
    stepNorth();
    HilbertNorth(nextlevel);
    stepEast();
    HilbertNorth(nextlevel);
    stepSouth();
    HilbertWest (nextlevel);
}
```

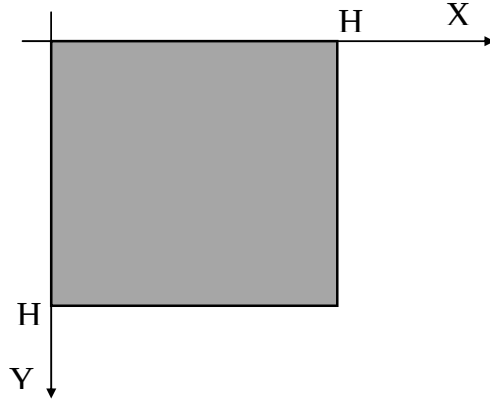
لتحديد نقطة بداية الرسم سنفترض أن منطقة الرسم عبارة عن مربع طول ضلعه h_0 ، عندئذ سنلاحظ أن رسم الخط H_i يبدأ من الموقع:

$$x = h_0 + h_0 / 2^i$$

$$y = h_0 - h_0 / 2^i$$

أما طول القطعة المستخدمة في رسم الخط H_i فهو:

$$h = h_0 / 2^i$$



بهذا نكمل تفصيل الخوارزمية التي تصبح كالتالي:

```
private void HilbertNorth(int level) {
    if (level <= 0) {
        HilbertLevel0();
        return;
    }

    int nextlevel = level - 1;

    HilbertEast (nextlevel);
    stepNorth();
    HilbertNorth(nextlevel);
    stepEast();
    HilbertNorth(nextlevel);
    stepSouth();
    HilbertWest (nextlevel);
}

private void HilbertEast(int level) {
    if (level <= 0) {
        HilbertLevel0();
        return;
    }

    int nextlevel = level - 1;

    HilbertNorth(nextlevel);
    stepEast();
    HilbertEast (nextlevel);
    stepNorth();
    HilbertEast (nextlevel);
    stepWest();
    HilbertSouth(nextlevel);
}

private void HilbertSouth(int level) {
    if (level <= 0) {
        HilbertLevel0();
    }
}
```

```

        return;
    }

    int nextlevel = level - 1;

    HilbertWest (nextlevel);
    stepSouth();
    HilbertSouth(nextlevel);
    stepWest();
    HilbertSouth(nextlevel);
    stepNorth();
    HilbertEast (nextlevel);
}

private void HilbertWest(int level) {
    if (level <= 0) {
        HilbertLevel0();
        return;
    }

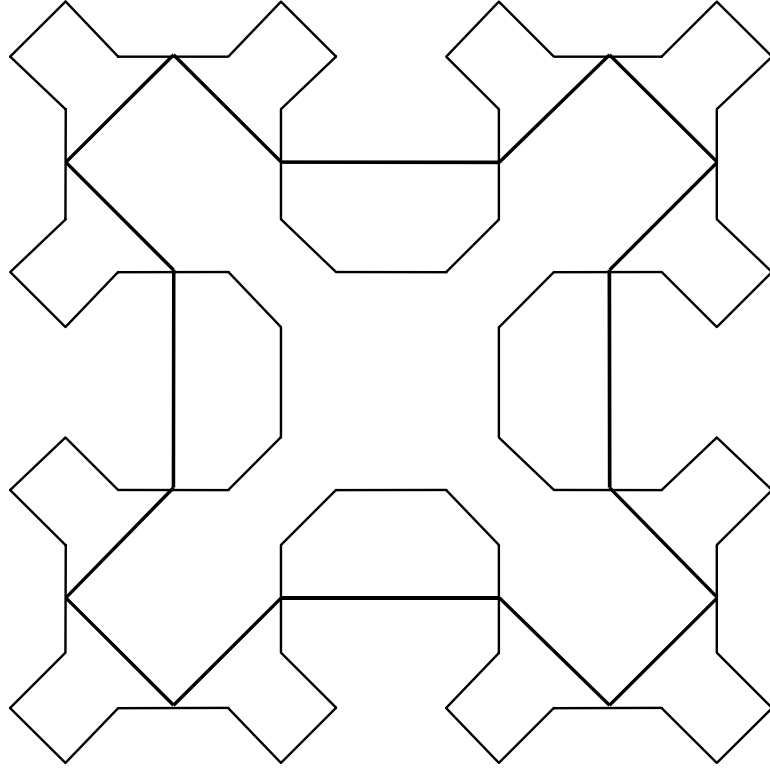
    int nextlevel = level - 1;

    HilbertSouth(nextlevel);
    stepWest();
    HilbertWest (nextlevel);
    stepSouth();
    HilbertWest (nextlevel);
    stepEast();
    HilbertNorth(nextlevel);
}
}

```

5.2 منحنيات سيرينسكي

نعرض في هذه الفقرة مثالاً لمنحنيات تشبه من حيث التركيب منحنيات هيلبرت، لكنها أكثر تعقيداً وأثاقاً. يبين الشكل التالي المنحنيين S1 و S2.

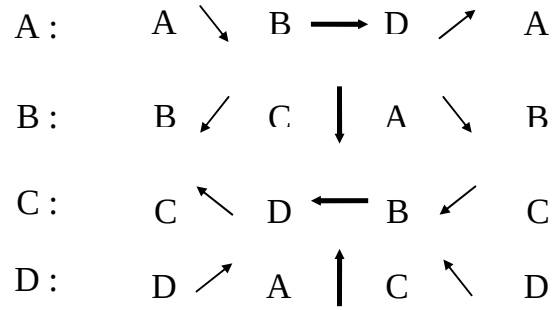


من ملاحظة هذين الرسمين نجد أننا لا نستطيع الاعتماد على S_1 في بناء S_2 حتى لو حذفنا أحد أضلاعه. هذا ناتج من كون منحنيات سيرينسكي منحنيات مغلقة، ويجب اعتماد مخطط عودّي لخوارزمية الرسم، بحيث نحصل على منحنيات مفتوحة يؤدي وصلها إلى الحصول على الرسم المطلوب.

سنعتبر أن منحنى سيرينسكي من الدرجة n هو تركيب لأربع إجراءات A, B, C, D تجمعها أربع قطع مستقيمة لا تنتمي إلى هذه المنحنيات ويعطي المنحنى المطلوب. إن المنحنيات الأربعة متماثلة وينتج كلٌّ منها من الآخر بدوران قدره 90° . المخطط الأساسي لرسم منحنى سيرينسكي هو:

$S: \quad A \searrow B \longleftarrow C \nearrow D$

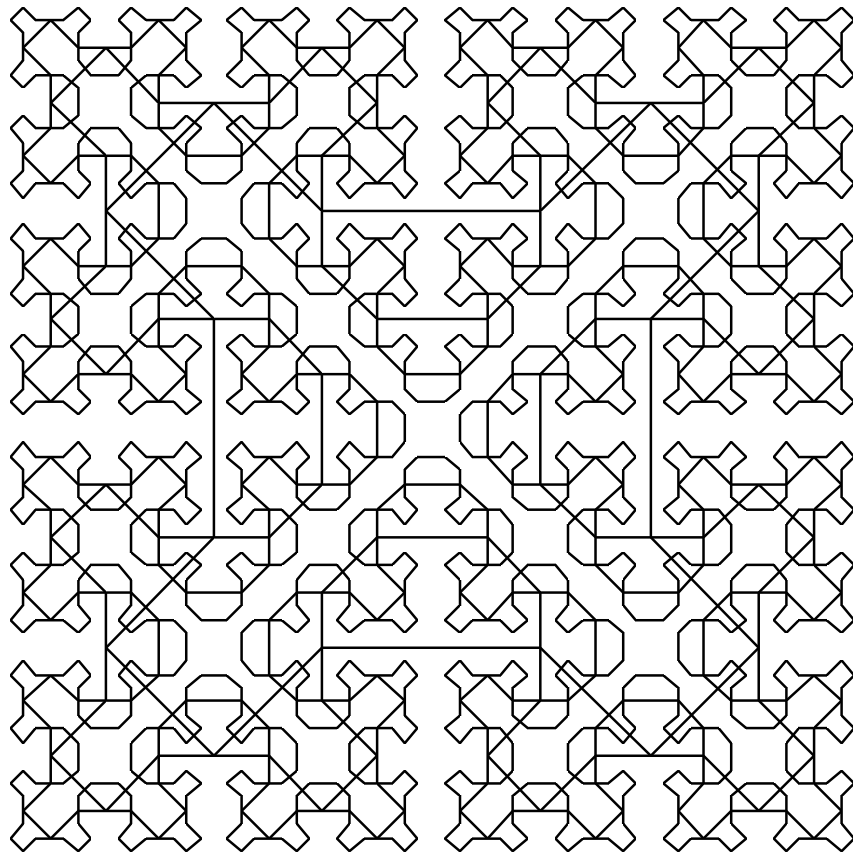
أما المخطط العودّي للإجراءات الأربع A, B, C, D التي تقوم برسم المنحنيات الجزئية فهو:



حيث يرمز السهم العريض إلى قطعة مستقيمة مضاعفة الطول. باتباع نفس الخطوات التي أوردناها لرسم منحنيات هيلبرت، يمكن التعبير عن الجزء A بالإجرائية التالية:

```
public void paint(Graphics g) {
    int level = 3;
    dist = dist0;
    for (int i=level;i>0;i--) dist /= 2;
    sg.goToXY(2*dist,dist);
    sierpA(level); // start recursion
    sg.lineRel(+dist,+dist);
    sierpB(level); // start recursion
    sg.lineRel(-dist,+dist);
    sierpC(level); // start recursion
    sg.lineRel(-dist,-dist);
    sierpD(level); // start recursion
    sg.lineRel(+dist,-dist);
}
```

يبين الشكل التالي منحنيات سيربينسكي من S1 حتى S4.



```

package siepr;

import java.awt.*;
import java.applet.*;

public class SierpinskiCurve extends Applet {
    private SimpleGraphics sg=null;
    private int dist0 = 128, dist;

    public void init() {
        sg = new SimpleGraphics(getGraphics());
        dist0 = 128;
        resize ( 4*dist0, 4*dist0 );
    }

    public void paint(Graphics g) {
        int level = 3;
        dist = dist0;
        for (int i=level;i>0;i--) dist /= 2;
        sg.goToXY(2*dist,dist);
        sierpA(level);      // start recursion
        sg.lineRel(+dist,+dist);
        sierpB(level);      // start recursion
        sg.lineRel(-dist,+dist);
        sierpC(level);      // start recursion
        sg.lineRel(-dist,-dist);
    }
}

```

```

        sierpD(level);          // start recursion
        sg.lineRel(+dist,-dist);
    }

    private void sierpA (int level) {
        if (level > 0)
        {
            sierpA(level-1); sg.lineRel(+dist,+dist);
            sierpB(level-1); sg.lineRel(+2*dist,0);
            sierpD(level-1); sg.lineRel(+dist,-dist);
            sierpA(level-1);
        }
    }

    private void sierpB (int level) {
        if (level > 0) {
            sierpB(level-1); sg.lineRel(-dist,+dist);
            sierpC(level-1); sg.lineRel(0,+2*dist);
            sierpA(level-1); sg.lineRel(+dist,+dist);
            sierpB(level-1);
        }
    }

    private void sierpC (int level) {
        if (level > 0) {
            sierpC(level-1); sg.lineRel(-dist,-dist);
            sierpD(level-1); sg.lineRel(-2*dist,0);
            sierpB(level-1); sg.lineRel(-dist,+dist);
            sierpC(level-1);
        }
    }
}

```