الفصل الثالث : الخوارزميات العودية
Recursive Algorithms

3-1 مفهوم العودية

نقول عن شيء إنه ذو بنية عودية إذا كان مؤلفاً من مجموعة من المكونات بعضها معرف تعريف الشيء الأصلي .

إن أكثر من اهتم بموضوع العودية هم الرياضيون الذين استخدموا أداة فعالة في التعاريف الرياضية و خاصة للمجموعات غير المنتهية . و من الأمثلة الشهيرة لاستخدام العودية نورد مايلي :

1. **تعريف الأعداد الطبيعية :** تعرف الأعداد الطبيعية كما يلي :
   - العدد صفر هو عدد طبيعي
   - إذا كان n عدداً طبيعياً فإن n+1 عدد طبيعي أيضا .

2. **تعريف العاملي لعدد صحيح غير سالب :** يعرف العاملي كما يلي :
   - 0! = 1
   - إذا كان n>0 عددا فإن n!=n(n-1) .

3. **تعريف دالة Ackermann :** تعرف هذه الدالة بالصيغة الآتية:

$$A(m,n)=\begin{cases} n+1 & if\ m=0 \\ A(m-1,1) & if\ n=0 \\ A(m-1,A(m,n-1)) & Otherwise \end{cases}$$

4. **تعريف متتالية أعداد فيبوناتشي :** تعرف هذه المتتالية بالشكل التالي :
   - $f_0=0,\ f_1=1$
   - $f_i=f_{i-1}+f_{i-2}\ ,\ i>1$

تكمن أهمية العودية في إمكانية 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 ملائمة عندما تكون المسألة المطلوب حلها أو بنية المعطيات الواجب معالجتها معرفة تعريفاً عودياً .

كما نقول عن برنامج جزئي (دالة / إجرائية) بأنه عودي عندما يستدعي نفسه (recursive call). وبالتالي يتم استدعاء هذه الدالة بشكل مكرر (أي يتم تنفيذ نفس التعليمات بشكل مكرر).

من المفيد في بعض المسائل أن نستخدم مفهوم العودية بدلاً من حلقات تكرارية, خاصة في مسائل الذكاء الصنعي ومسائل الترتيب...الخ . نلاحظ أن تكرار الاستدعاء العودي لا نهائي ، لذلك يجب بالضرورة وضع شرط توقف لإنهاء التكرار.

فائدة الخوارزميات العودية:

عندما نتعامل مع مسائل معقدة تقوم العودية بتقسيم هذه الحالة إلى حالات أبسط على التتالي حتى إن تصل إلى حالة بدائية. ومن ثم تعود خطوة – خطوة وتعوض القيم الناتجة إلى أن تصل إلى الحالة الأصلية فنحصل على الناتج النهائي.

سلبيات استخدام العودية:

1- الاستدعاءات العودية تسبب ضياعاً في الوقت وتستهلك ذاكرة إضافية (لأن كل استدعاء يُعتبر تابعاً جديداً يتم حجز متحولاته المحلية و وسطائه من جديد في المكدس, مما يؤدي في الأخير إلى الطفحان (Stack Overflow).
2- لا تقبل بعض اللغات البرمجية التعريف العودي للتوابع ومنها مثلاً: لغة الآلة , Fortran , Cobol ...

نتيجة: يجب أن يكون عمق العودية (أي عدد تكرار الاستدعاءات العودية للتابع نفسه) منتهياً و صغيراً.

ملاحظة: أحياناً اللجوء إلى العودية ليس هو الحل الأمثل بل يُفضل عليه استخدام الخوارزميات التكرارية.

3-2 أنواع الخوارزميات العودية

يمكن التعبير عن خوارزمية عودية P بواسطة تركيب C لمجموعة عمليات أساسية S_i (لا تحوي P) مع P نفسها : $P \equiv C[S_i , P]$

إن الأداة اللازمة و الكافية للتعبير عن برنامج معين تعبيراً عودياً هي الدالة (function) لأنها تسمح بإعطاء اسم لمجموعة تعليمات تنجز عملية منطقية ، قد تكون معقدة و طويلة ، و هذا ما يسمح باستدعاء هذه التعليمات استدعاء" عودياً .

يمكن التمييز بين نوعين من الدوال العودية : دوال ذات عودية مباشرة أو ذات عودية غير مباشرة, حيث يبيّن الجدول (3-1) الفرق بينهما:

الجدول (3-1): العودية المباشرة و العودية غير المباشرة

الدوال ذات العودية المباشرة	الدوال ذات العودية غير المباشرة
هي توابع تحوي استدعاءً صريحاً لنفسها. مثال:	هي توابع تحوي استدعاءً لتوابع أخرى و التي بدورها تستدعي التابع الأب. مثال: إذا كان لدينا التابع

int A(void) {
…….
A( ); // direct recursive call
…….
}
	void B(void) {
	…….
	A( );
	…….
	}
	
	int A(void) {
	…….
	B( ) ; // indirect recursive call
	…….
	}

3-3 انتهاء تنفيذ دالة عودية:

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

$$P \equiv if\ B\ then C\ [S_i, P]$$

أو

$$P \equiv C[S_i, if\ B\ then\ P]$$

للبرهان على انتهاء إجرائية عودية نعرف تابع P(X) حيث X مجموعة معاملات الدالة الإجرائية . نعرف التابع P(X) بحيث تكون القضية B مكافئة لـــ P(X) >0 و في المرحلة التالية يتناقص عند كل استدعاء عودي .

مثال:

لتكن دالة حساب العاملي :

```
long int fact(int n)
        {
  if (n==1||n==0)
     return 1;
  else
     return n*fact(n-1);
         }
```

في هذه الحالة مجموعة المعاملات هي X={n} ، إذا عرفنا التابع P(n) = n فإن شرط الاستدعاء العودي هو P(n) > 0 . كما نلاحظ من نص الدالة الإجرائية أن التابع يتناقص بمقدار واحد عند كل استدعاء عودي .

لنكتب دالة حساب العاملي بالطريقة التكرارية :

```
long int fact(int n)
{
  Long int  f=1;
  if (n==1||n==0)
     return 1;
  else
  {
     for (int i=1;i<=n;i++)
        f*=i;
     return f;
  }
}
```

3-4 أمثلة لبرامج عودية

3-4-1 حساب قيمة متتالية فيبوناتشي عند عدد ما - Fibonacci :

إن الشكل العام لمتتالية فيبوناتشي هو :

$F_n = F_{n-1} + F_{n-2}$; $n \geq 2$ $(F_0 = 0 , F_1 = 1)$

و بالتالي لدينا مثلاً:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	……..
0	1	1	2	3	5	8	13	……..

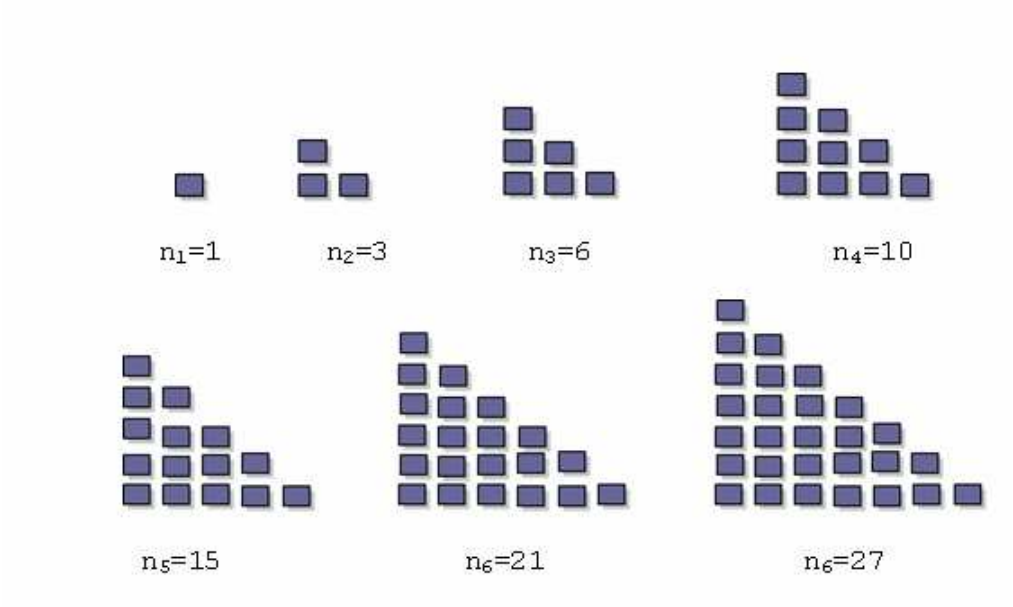
إذاً لدينا حالتان بدائيتان (0 و 1) بينما من أجل أي عدد آخر لا يمكن حساب القيمة إلا بالاعتماد على القيم المسبقة, هذا يعني أنه لدينا تكرار حساب تابع فيبوناتشي ولكن في كل مرة لأعداد أصغر حتى نصل إلى أبسط قيم ممكنة (أي 0 و 1). ويكتب التابع العودي كما يلي:

```
unsigned long fib(unsigned  int  n)
{
 if (n<=1) return n;
 else
 if (n==1) return 1;
 else    return fib(n-1) + fib(n-2);
}
```

3-4-2 الأعداد المثلثية (Triangular Numbers) :

لاحظ اليونانيون القدماء أن ثمة علاقة وطيدة بين مجموعة الأعداد المتسلسلة:

... 1, 3, 6, 10, 15, 21, 28 فيما تعرف بأعداد فيثاغورث (Pythagoreans) . في هذه المتتالية العدية يتم الحصول على العنصر (n th) من خلال إضافة (n) إلى العنصر السابق . و بذلك ، يمكن إيجاد العنصر الثاني بإضافة (2) إلى العنصر الأول و هو العدد واحد ليكون الناتج ثلاثة . و ينتج العنصر الثالث من إضافة 3 إلى العنصر الثاني ليكون 6 ، و هكذا . و تسمى الأعداد – في هذه الحالة – بالأعداد المثلثية و ذلك لأنها تأخذ في النهاية شكل المثلث – كما هو موضح في الشكل (3-1).



الشكل(3-1): الأعداد المثلثية

بفرض أننا نرغب بإيجاد عدد في سلسلة عددية ؛ و ليكن هذا العنصر – مثلاً – الرابع في التسلسل (و قيمته عشرة) ؛ و يتم ذلك من خلال جمع عدد المربعات الموجودة في كل الأعمدة . ففي حالة العنصر الرابع – مثلا – يحتوي العمود الأول على أربعة مربعات و الثاني على ثلاثة و هكذا . و بإضافة 4 إلى 3 إلى 2 إلى 1 يكون الناتج عشرة . و هي

القيمة الموجودة في الرقم الرابع في التسلسل .

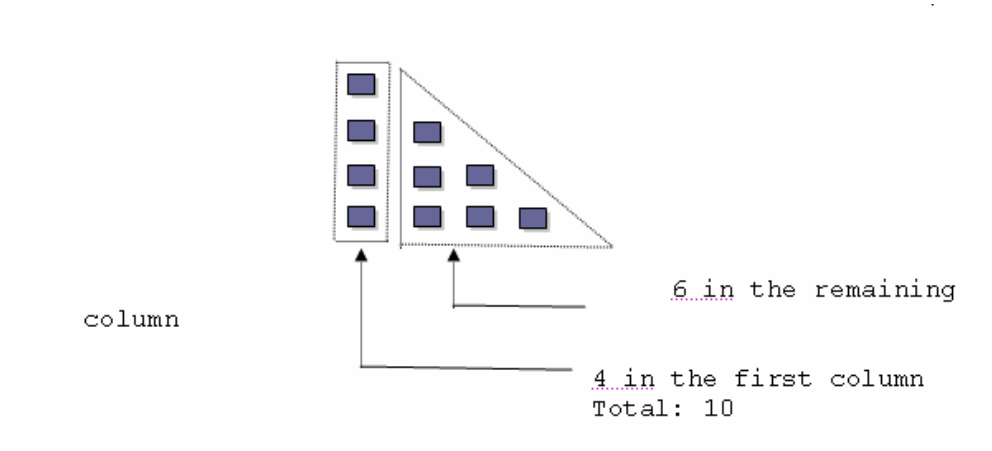
الدالة الاجرائية ()triangle تستخدم هذه التقنية المبنية على فكرة الأعمدة لإيجاد عدد مثلثي ، و تقوم على جمع عدد المربعات في الأعمدة كلها بدءا" من العمود ذي الارتفاع n و حتى العمود ذي الارتفاع 1 .

```
int triangle (int n)
  {
  int total = 0;
  While (n>0){
     total = total +n;
     --n;
     }
  return total;
 }
```

لنستخدم الآن مفهوم العودية في إيجاد قيمة العدد n من الأعداد المثلثية ، و الذي هو جمع جزأين اثنين من عناصر التركيب بدلا" من التركيب كله . فهناك :

1. عدد المربعات في العمود الأول (الأكثر ارتفاعا") و يساوي n .
2. مجموع المربعات في الأعمدة الأخرى .

و هذا ما يوضحه الشكل (3-2):



الشكل(3-2): تنظيم الأعداد المثلثية في شكل عمود منفصل و مثلث

و يكتب التابع العودي كما يلي :

```
int triangle (int n)
   {
  if(n == 1) return 1;
  else
    return(n + triangle(n-1));
     }
```

نلاحظ أن قيمة عنصر الدخل n الموجود في الدالة العودية triangle يتناقص بمقدار واحد في كل مرة تتكرر فيها عملية الاستدعاء . و هو الأمر الذي يؤدي في النهاية إلى تضييق نطاق المشكلة . و من ثم ، يكون التعامل معها أسهل . و يستمر ذلك حتى تصل قيمة عنصر الدخل إلى الحد الأدنى و هي القيمة واحد و التي تحقق شرط التوقف في الدالة ، فتنتج القيمة دون تكرار عملية الاستدعاء مرة أخرى .

ملاحظة:

يقابل مفهوم العودية في البرمجة مفهوم الاستقراء الرياضي (Mathematical Induction) و كون الاستقراء الرياضي عبارة عن أسلوب تعريف الشيء بنفسه . و باستخدام هذا الأسلوب يمكن حساب الأعداد المثلثية رياضيا على هذا النحو :

$$tri(n) = \begin{cases} 1 & if\ n = 1 \\ n + tri(n-1) & if\ n > 1 \end{cases}$$

3-4-3 توليد التباديل (Permutation Generator)

يقصد بتوليد التباديل : تغيير يجرى في ترتيب أحرف كلمة ما بغية تشكيل تراكيب جديدة . إذا طبقنا هذا المفهوم على كلمة cat ، سيكون الناتج على النحو التالي :

- cat
- cta
- act
- atc
- tac
- tca

بتنفيذ هذه العملية على كلمة أخرى مكونة من n حرف ، سنجد أن عدد الاحتمالات مساويا دائما لـــ n! . ففي حال الكلمة المكونة من ثلاثة أحرف ، سيكون عدد التراكيب الممكنة ستة و في حال الكلمة المكونة من أربعة أحرف ، سيكون عدد التراكيب الممكنة أربعة و عشرين، و هكذا .

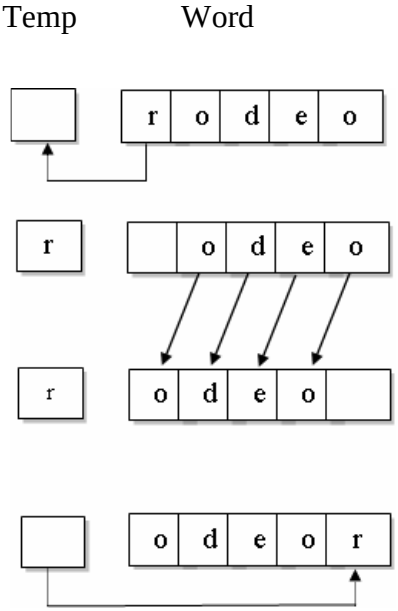
ملاحظة: يفترض أن تكون الحروف المكونة للكلمة مختلفة لأن وجود الحرف نفسه أكثر من مرة سيؤدي إلى وجود احتمالات أقل لتركيب هذه الكلمة .

خطوات تنفيذ عملية توليد التباديل

يتم تنفيذ خطوات عملية توليد التباديل على كلمة مكونة من n حرف بالشكل :

1. انقل n-1 حرف إلى أقصى اليمين
2. انقل جميع الحروف نحو اليسار
3. كرر هذه الخطوات بعدد العناصر نفسه .

تعريف : يقصد بمصطلح تدوير (rotate): عملية نقل حروف الكلمة بمقدار خانة واحدة نحو اليسار باستثناء الحرف الأخير الموجود يمين الكلمة لأنه سينتقل مرة أخرى نحو اليمين .كما هو واضح في الشكل (3-3) .



الشكل (3-3): عملية التدوير

تسمح عملية التدوير (rotation) بإعطاء فرصة لبدء الكلمة بكل حرف من هذه الحروف . و سيتم إعطاء جميع الاحتمالات للتراكيب الممكنة للحروف عندما يكون الحرف المختار موجودا في أول الكلمة . وبالنسبة لكلمة cat – مثلا – التي لا تحتوي إلا على ثلاثة أحرف ، ستتم عملية النقل لحرفين فقط . و يوضح الجدول(3-2) ذلك .

الجدول(3-2) : إيجاد التراكيب المختلفة لكلمة cat

Word	Word?	Display letter	First letters	Remaining Action
cat	Yes	c	at	Rotate at
cta	Yes	c	ta	Rotate ta
cat	No	c	at	Rotate cat
atc	Yes	a	tc	Rotate tc
act	Yes	a	ct	Rotate ct
atc	No	a	tc	Rotate atc
tca	Yes	t	ca	Rotate ca
tac	Yes	t	ac	Rotate ac
tca	No	t	ca	Rotate tca
cat	No	c	at	Done

نلاحظ أنه لابد من نقل الحروف مرة أخرى إلى نقطة البداية بحرفين قبل تنفيذ عملية النقل للأحرف الثلاثة معا" . و ينتج عن ذلك التراكيب : cat, cta, cat أما التراكيب الأخرى العشوائية فلا تظهر . و بذلك نكون أوجدنا جميع التراكيب الممكنة لهذه الكلمة باستدعاء عودي للدالة ()Perm و التي تحوي عنصر دخلٍ واحدٍ وهو فقط طول الكلمة أو عدد الحروف المكونة لها . و يقصد بطول الكلمة هنا عدد n من الحروف الموجودة في أقصى اليمين و منها تتكون الكلمة كلها ، و في كل مرة تستدعى فيها الدالة ()perm عوديا ، تتكرر هذه العملية بهذا الشكل حيث يقل طول الكلمة بمقدار حرف واحد في كل مرة .

و تتوقف عملية الاستدعاء العودي للدالة عندما لا يكون هناك سوى حرف واحد في الكلمة ، و هو التركيب التالي الممكن لها . فليس ثمة وسيلة لإعادة تنظيم حرف واحد هو كل ما في الكلمة . و لو لم يكن الأمر كذلك ، فسيتم تنظيم جميع الحروف عدا الحرف الأول في الكلمة، ثم تنتقل جميع أحرف الكلمة بعد ذلك إلى أقصى اليسار. و تحدث هذه العملية n مرة عندما يكون عدد الأحرف في الكلمة مساوياً لـــ n .
الدالة ()perm مسؤولة عن تنفيذ جميع العمليات السابق ذكرها .

```
void perm (int new-size)
    {
  if(new-size==1) return;
  for(int j=0; j<newSize; j++) {
    perm(newSize-1);
    if(newSize==2)
      displayWord();
    rotate(newSize);
     }
   }
```

يوضح المثال التالي نموذج البرنامج الكامل perm.cpp . و يستخدم فيه الصف class لتمثيل الكلمة التي سيتم تنظيمها . يحتوي هذا الصف على دوال لعرض التراكيب و عملية التدوير .

```
#include <iostream . h>
#include <string . h>
 ////////////////////////////////
  class word{
    private:
    int size ;
    int count ;
    string workStr;
    void rotate(int);
    void display-word();
    Public:
    word(string);
    void perm(int);
     };
//---------------------------------------
// constructor
word ::word(string inpStr):workStr(inpStr),count(0)
{
Size=inpStr.length();
}
//---------------------------------------
void word::perm(int newSize)
    {
if(newSize == 1)
```

```
 return ;
for(int j=0; j<newSize; j++)
 {
  perm(newSize-1);
  if(newSize == 2)
   displayWord();
  rotate(newSize);
  }
 }
//--------------------------------------------
//rotate left all chars from position to end

void word::rotate(int newSize)
 {
  int j;
  int position = size-newSize;
  char temp=workStr[position];
  for(j=position+1; j<size;j++)
    workStr[j-1]=workStr[j];
  workStr[j-1]=temp;
}
//-------------------------------------------------

void word::displyWord()
  {
  if(count<99)
  cout<< " ";
  if(count<9)
   cout<< " ";
  cout<<++count<<" ";
  cout<<workStr<<" ";
  if(count%==0)
    cout<<end;
}
```

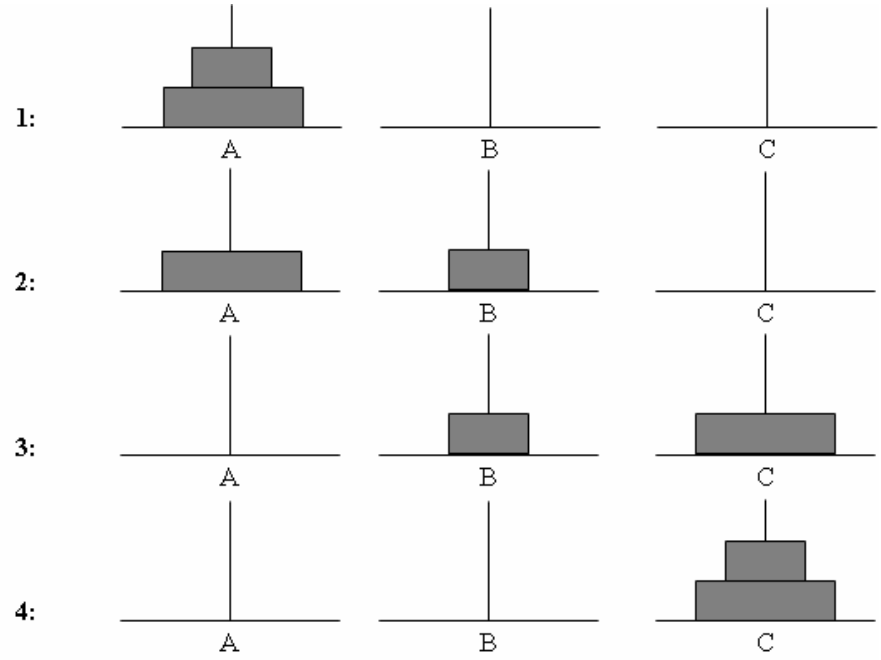
3-4-4 مسألة أبراج هانوي - Towers of Hanoi :

ليكن لدينا n قرصاً من أقطار مختلفة. ولكل قرص ثقب بحيث يمكن إدخاله إلى عامود. والمطلوب:

بناء برج هرمي من هذه الأقراص على عامود C علماً بأنها كانت متوضعة على شكل برج هرمي على عامود A مع التقييد بالشروط التالية :

- يُسمح في كل خطوة بنقل قرص واحد فقط من عمود لآخر.
- يُمنع وضع قرص أكبر على قرص أصغر.
- يمكن استخدام العمود المساعد B مرحلياً.

(كتنفيذ يُطلب فقط طباعة مراحل النقل, أي معرفة نقل أي قرص وإلى أي عمود على التتالي). فمثلاً من أجل n = 2 لدينا:



الحل:

```
#include <iostream.h>
int n;
void hanoi(int n, char A, char B, char C);

void main()
{
  do{
     cout<<"Enter n= ";
          cin>>n;
```

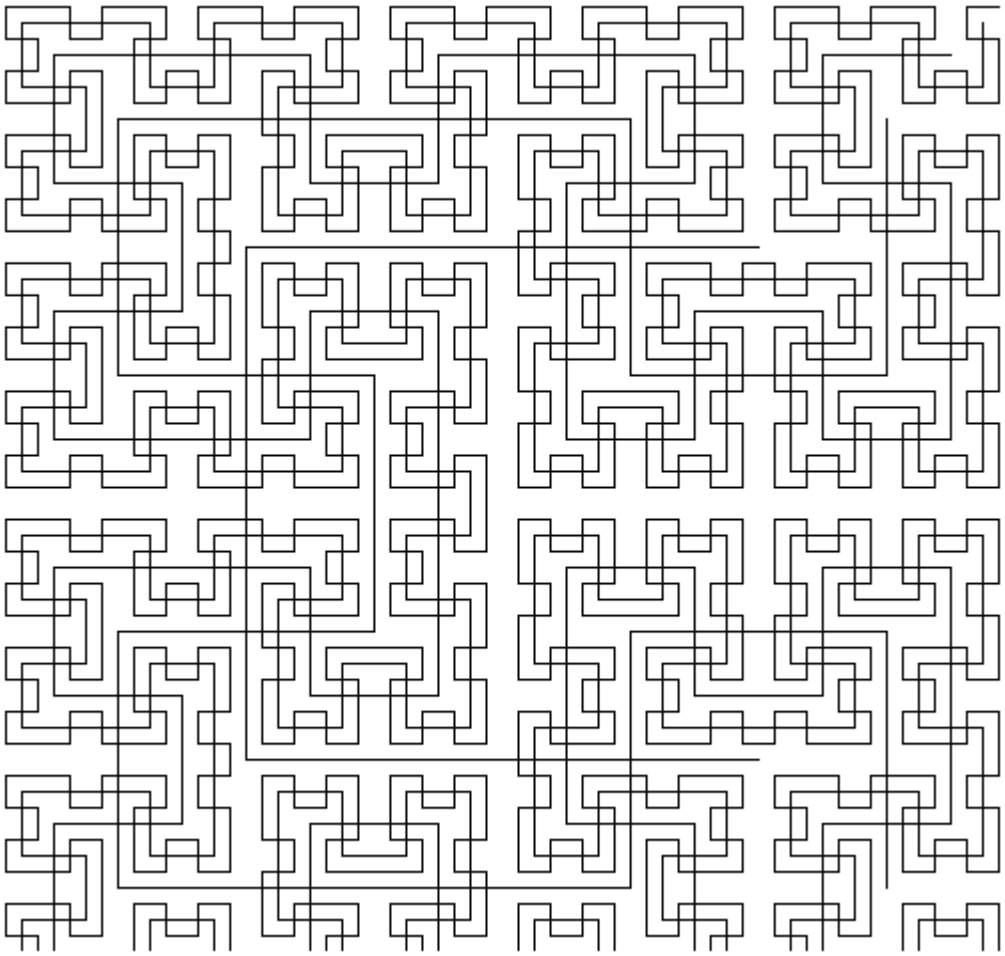
```
 }while(n<=0);
 hanoi(n,'A','B','C');
}

void hanoi(int n, char A, char B, char C)
{
 if (n==1) cout<<A<<" --> "<<C<<endl;
 else
 {
    hanoi(n-1,A,C,B);
    cout<<A<<" --> "<<C<<endl;
        hanoi(n-1,B,A,C);
 }
}
```

حيث إنه يعطي تنفيذ البرنامج السابق النتائج التالية (من أجل قــرص أو قرصــين أو ثلاثــة أقراص):

n = 1	n = 2	n = 3
A → C	A → B A → C B → C	A → C A → B C → B A → C B → A B → C A → C

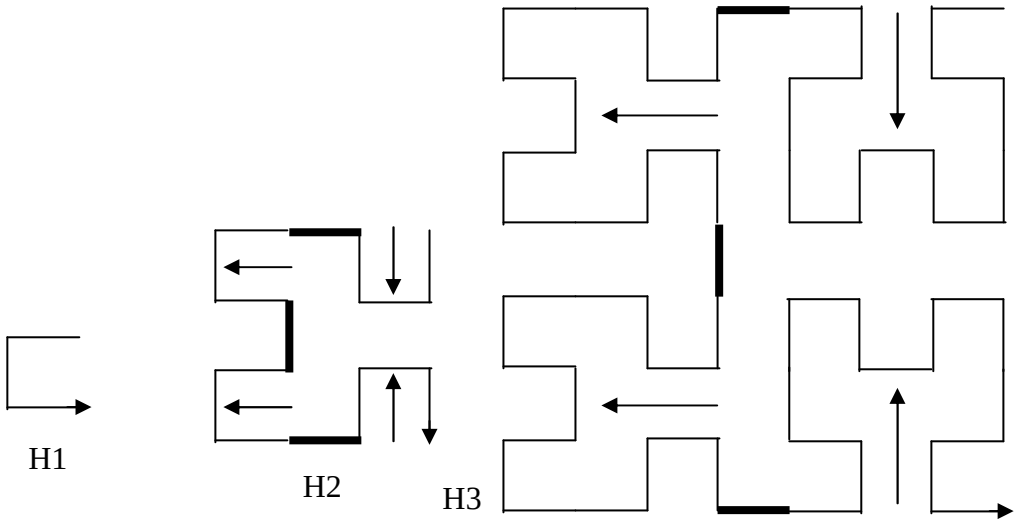
3-4-5 منحنيات هيلبرت - Hilbert :

إن منحنيات هيلبرت مثال عن الزخرفة المتناسقة, حيث إنها عبارة عن تركيب عدة خطــوط منكسرة تتبع منهجاً في الرسم:



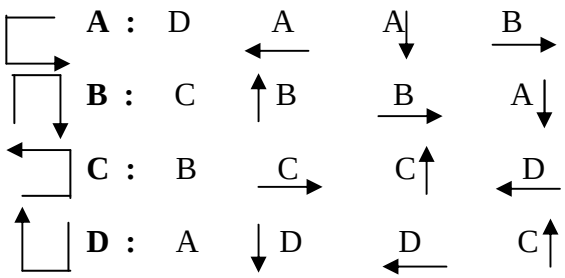
(منحنيات هيلبرت من H1 إلى H5)

نجد بعد دراسة الشكل السابق جيداً أنه مؤلف من خمسة أشكال يتوضع بعضها فوق بعض, ويتألف كل من الثلاثة الأولى منها من خط منكسر من إحدى الأشكال الثلاثة الموضحة كالتالي , حيث نرمز لهذه الخطوط بالرموز H1 و H2 و H3 .(خطوط هيلبرت من المستويات H1 و H2 و H3)



نلاحظ أنه يمكن الحصول على المستوى H_{i+1} من تركيب أربعة أشكال من المستوى H_i التي تعد أصغر حجماً بمرتين, ومتوافقة من حيث الشكل, وتوجه هذه الأشكال الأربعة إلى جميع الاتجاهات المختلفة, ويتصل بعضها ببعض بثلاث قطع مستقيمة. يمكن اعتبار أنه المستوى H_1 مؤلف من أربعة أشكال H_0 فارغة أي أنها أربع نقاط تم وصلها بوساطة ثلاثة خطوط.

بما أن كل منحن 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. لنرمز إلى هذه الإجرائيات الأربعة بـ A,B,C,D ولنرمز بأسهم إلى عمليات رسم الوصلات, وبالتالي نستطيع التعبير عن المخطط العودي لهذه الإجرائيات كما يلي:




```
include <graphics.h>
#include <conio.h>  // --> getch()

const int n=5,h0=512;
int h,x,y,x0,y0;

void A(int i);
void B(int i);
void C(int i);
void D(int i);
/*********************************/

void main()
{
int gdriver = DETECT, gmode;
initgraph(&gdriver, &gmode, "");
int i=0;
h=h0;
x0=h/2;
y0=x0;
do{
i++;
h/=2;
setcolor(i);
x0=x0+(h/2);
y0=y0-(h/2);
moveto(x0,y0);
```

```
x=x0;
y=y0;
A(i);
getch();
}while (i<n);
closegraph();
}
/*********************************/

void A(int i)
{
if (i>0){
D(i-1); x-=h; lineto(x,y);
A(i-1); y+=h; lineto(x,y);
A(i-1); x+=h; lineto(x,y);
B(i-1);
}
}
/*********************************/

void B(int i)
{
if (i>0){
C(i-1); y-=h; lineto(x,y);
B(i-1); x+=h; lineto(x,y);
B(i-1); y+=h; lineto(x,y);
A(i-1);
}
}
/*********************************/
void C(int i)
{
if (i>0){
B(i-1); x+=h; lineto(x,y);
C(i-1); y-=h; lineto(x,y);
C(i-1); x-=h; lineto(x,y);
D(i-1);
}
}
/*********************************/
```

```
void D(int i)
{
if (i>0){
A(i-1); y+=h; lineto(x,y);
D(i-1); x-=h; lineto(x,y);
D(i-1); y-=h; lineto(x,y);
C(i-1);
}
}
```

الحل لمنحنيات هيلبرت بطريقة أخرى من أجل التنفيذ بـــ Borland C++ Builder :

1- في Borland C++ Builder نختار من قائمة File الخيار "New Application"

2- ننتقل إلى نافذة التحرير لـــ Unit1.cpp

3- نترك الكتابات الافتراضية من دون أي تغيير ونضيف من بعدها الكود التالي:

```
int i=0,n=5,x0=256,y0=256,x,y,h=512;

void A(int i);
void B(int i);
void C(int i);
void D(int i);
void delay();  // تابع لتأخير متابعة التنفيذ ببعض الثواني
//-----------------------------------------
void A(int i)
{
   if (i>0){
      D(i-1); x-=h; Form1->Canvas->LineTo(x,y);
      A(i-1); y+=h; Form1->Canvas->LineTo(x,y);
      A(i-1); x+=h; Form1->Canvas->LineTo(x,y);
      B(i-1);
   }
}
```

```
//-----------------------------------------------------
void B(int i)
{
   if (i>0){
      C(i-1); y-=h; Form1->Canvas->LineTo(x,y);
      B(i-1); x+=h; Form1->Canvas->LineTo(x,y);
      B(i-1); y+=h; Form1->Canvas->LineTo(x,y);
      A(i-1);
   }
}
//-----------------------------------------------
void C(int i)
{
   if (i>0){
      B(i-1); x+=h; Form1->Canvas->LineTo(x,y);
      C(i-1); y-=h; Form1->Canvas->LineTo(x,y);
      C(i-1); x-=h; Form1->Canvas->LineTo(x,y);
      D(i-1);
   }
}
//-----------------------------------------------
void D(int i)
{
   if (i>0){
      A(i-1); y+=h; Form1->Canvas->LineTo(x,y);
      D(i-1); x-=h; Form1->Canvas->LineTo(x,y);
      D(i-1); y-=h; Form1->Canvas->LineTo(x,y);
      C(i-1);
   }
}
//---------------------------------------
void delay()    // تابع لتأخير متابعة التنفيذ ببعض الثواني
{
 for (int wait=1; wait<999999999; wait++);      // حلقة لضياع الوقت
}
//---------------------------------------
```

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
   Form1->WindowState=wsMaximized;
   do{
      i++;
      h/=2;
      x0+=h/2; x=x0;
      y0-=h/2; y=y0;
      Form1->Canvas->MoveTo(x0,y0);
      Form1->Canvas->Pen->Color=i*100;
      A(i);
      delay();
   }while(i<n);
}
//---------------------------------------------------------------------------
```

4- نذهب إلى نافذة الـــ Object Inspector و نحدد الـــ Events Tab ثم نقرة مزدوجة على خانة النص الفارغة بجانب الحدث OnPaint

5- نعود إلى نافذة التحرير ونحذف الأسطر الجديدة المضافة, لأننا قمنا بكتابتها سابقاً بالشكل المطلوب.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
        {
                }
```

6- نحفظ المشروع من خلال Save All , ولا ضرورة لتغيير أسماء الملفات الافتراضية.

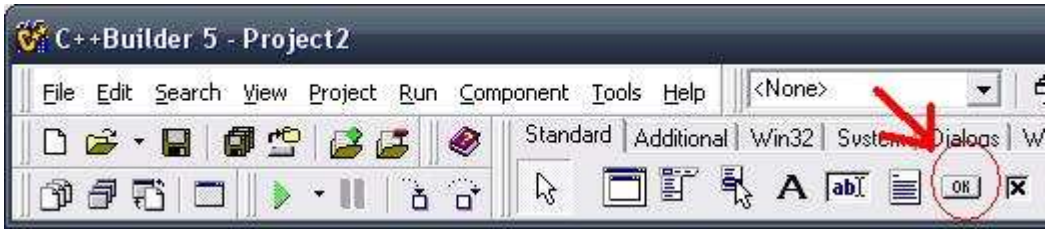
7- لتنفيذ البرنامج نضغط F9 .

ملاحظة:

يمكن تحسين البرنامج بإضافة زر إلى الفورم بحيث إنه يتم رسم منحني هيلبرت جديد كلما

نقرنا على الزر (بهذه الحالة لا نضيف الحدث OnPaint إلى الفورم ولا داعي للتابع delay بعد الآن).

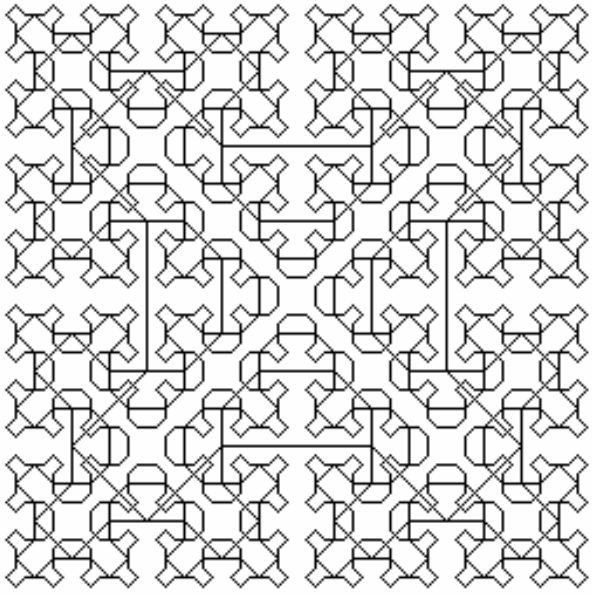
نحدد رمز الزر من شريط الأدوات Standard ثم نرسمه على الفورم. ثم نقرة مزدوجة على الزر المرسوم وإضافة الكود التالي:



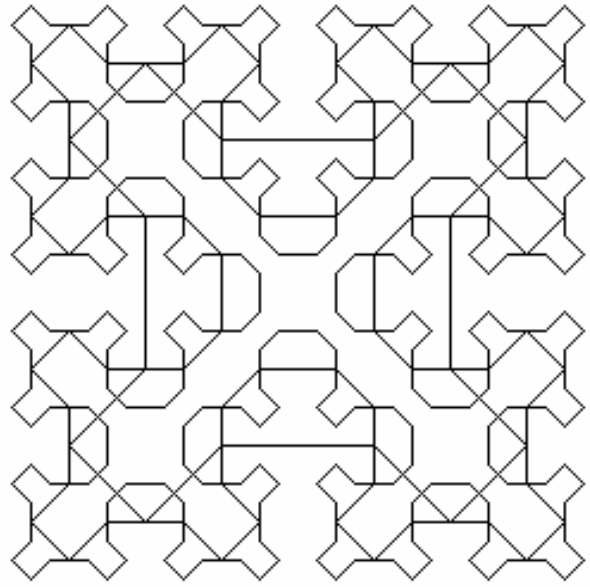

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   if (i<n){
      Form1->WindowState=wsMaximized;
      i++;
      h/=2;
      x0+=h/2; x=x0;
      y0-=h/2; y=y0;
      Form1->Canvas->MoveTo(x0,y0);
      Form1->Canvas->Pen->Color=i*100;
      A(i);
   }
}
```

6-4-3 منحنيات سيربنسكي - Sierpinsky :

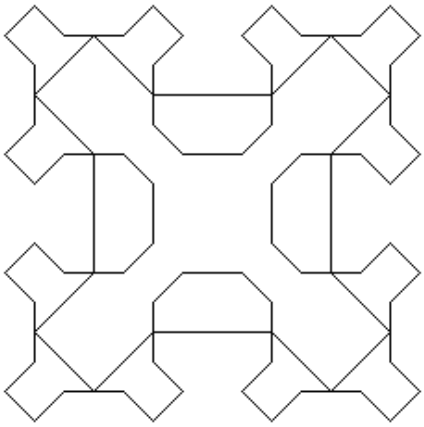
منحنيات سيربنسكي تشبه من حيث التركيب منحنيات هلبرت, لكنها أكثر تعقيداً وأناقة ! فمثلاً لدينا:



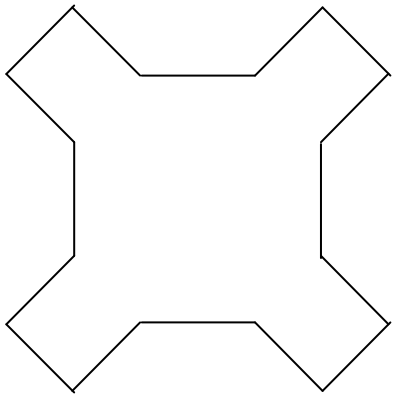
منحنيات سيربنسكي من S1 - S4



منحنيات سيربنسكي من S1 - S3



منحنيات سيربنسكي من S1 – S2

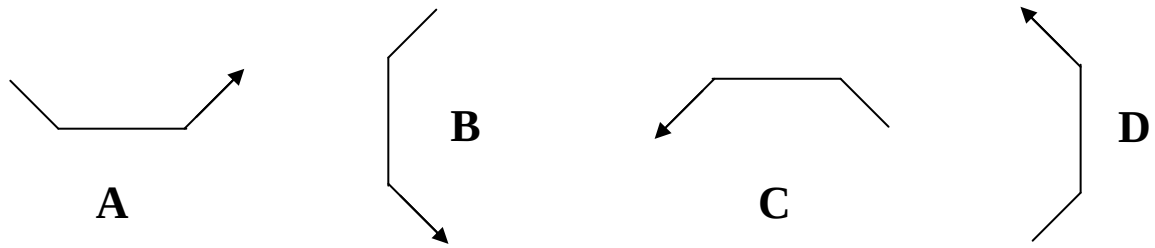


منحني سيربنسكي من المستوى S1

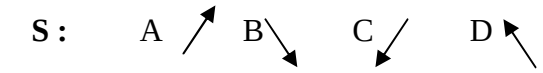
نلاحظ أنه لا يمكن رسم S2 بالاعتماد على S1 لكون منحنيات سيربنسكي منحنيات مغلقة, لذلك يجب اعتماد مخطط عودي لخوارزمية الرسم, بحيث نحصل على منحنيات مفتوحة يؤدي وصلها إلى الحصول على الرسم المطلوب.

سنعتبر أن منحني سيربنسكي من الدرجة n هو تركيب لأربع إجرائيات A,B,C,D تجمعها أربع قطع مستقيمة لا تنتمي إلى هذه المنحنيات ويعطي المنحني المطلوب. إن المنحنيات الأربعة متماثلة وينتج كل منها من الآخر بدوران قدره 90^o .

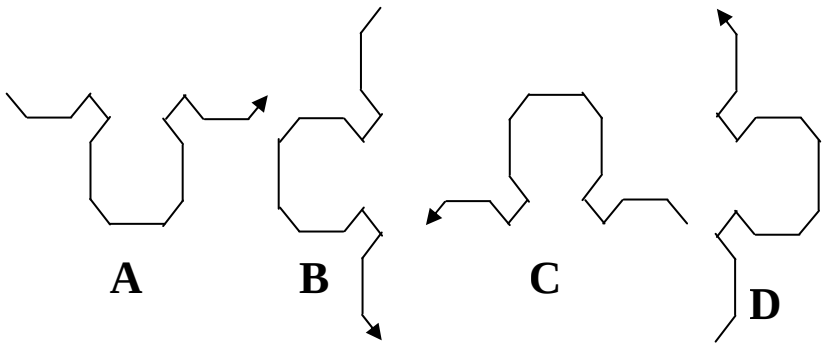
فمثلاً في أثناء رسم S1 اعتمدنا على القطع الأربع التالية مع الوصل بينها بقطع مستقيمة مائلة وذلك بدءاً من عند رسم A :



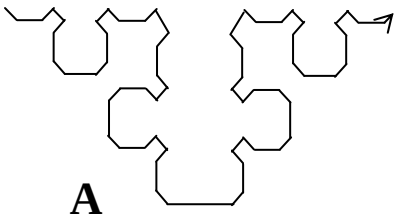
فالمخطط الأساسي لرسم منحني سيربنسكي (مهما كان المستوى) هو:



أما لرسم S2 فيجب الاعتماد على القطع الأربع التالية مع الوصل بينها بقطـــع مـــستقيمة مائلة وذلك بدءاً من عند رسم A من المستوى الثاني:

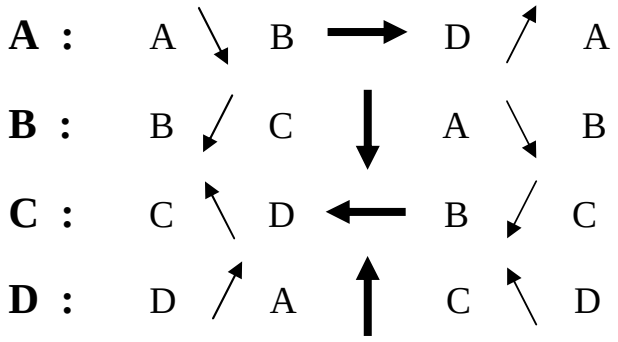


أما لرسم S3 فيجب الاعتماد على القطع الأربع التالية مع الوصل بينها بقطع مستقيمة مائلة وذلك بدءاً من عند رسم A من المستوى الثالث:



. . . . . .

حيث إن منحني سيربنسكي من الدرجة صفر أي S0 هو عبارة عن أربع نقاط تم الوصل بينها بقطع وصل. أما المخطط العودي للإجرائيات الأربعة A,B,C,D التي تقوم برسم المنحنيات الجزئية فهو:



حيث يرمز السهم العريض ⟶ إلى قطعة مستقيمة مضاعفة الطول. بإتباع الخطوات التي أوردناها نفسها لرسم منحنيات هيلبرت نكتب البرنامج اللازم لرسم منحنيات سيربنسكي:

يتم تطبيق البرنامج في Borland Turbo C++ under Dos بالشكل الأتي :

```
#include <graphics.h>
#include <conio.h>

int i=0,h=64,n=4,x,y,x0,y0;

void A(int i);
void B(int i);
void C(int i);
void D(int i);

void main()
{
 int gdriver = DETECT, gmode;
 initgraph(&gdriver, &gmode, "");

 x0=2*h;
 y0=h;

 do{
    i++;
```

```
    x0-=h;
    h/=2;
    y0-=h;
    x=x0;
    y=y0;
    moveto(x0,y0);
    setcolor(i);
    A(i); x+=h; y+=h; lineto(x,y);
    B(i); x-=h; y+=h; lineto(x,y);
    C(i); x-=h; y-=h; lineto(x,y);
    D(i); x+=h; y-=h; lineto(x,y);
    getch();
 }while(i<n);
 closegraph();
}
/*********************************************/

void A(int i)
{
 if (i>0){
    A(i-1);  x+=h;    y+=h; lineto(x,y);
    B(i-1);  x+=2*h;       lineto(x,y);
    D(i-1);  x+=h;    y-=h; lineto(x,y);
    A(i-1);
 }
}
/************************************/

void B(int i)
{
 if (i>0){
    B(i-1);  x-=h;  y+=h;    lineto(x,y);
    C(i-1);         y+=2*h;  lineto(x,y);
    A(i-1);  x+=h;  y+=h;    lineto(x,y);
    B(i-1);
 }
}
/************************************/

void C(int i)
{
```

```
 if (i>0){
    C(i-1); x-=h;   y-=h; lineto(x,y);
    D(i-1); x-=2*h;       lineto(x,y);
    B(i-1); x-=h;   y+=h; lineto(x,y);
    C(i-1);
 }
}
/*************************************************/
```

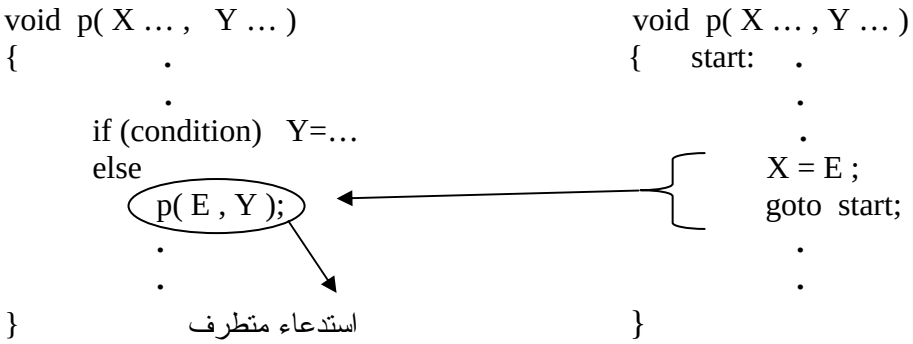
```
void D(int i)
{
 if (i>0){
    D(i-1); x+=h; y-=h;   lineto(x,y);
    A(i-1);       y-=2*h; lineto(x,y);
    C(i-1); x-=h; y-=h;   lineto(x,y);
    D(i-1);
}
}
```

3-5 تحويل الخوارزميات العودية إلى خوارزميات تكرارية

3-5-1 طريقة حذف الاستدعاء العودي المتطرف:

نقول عن استدعاء عودي إنه متطرف إذا كان هذا الاستدعاء ينهي تنفيذ الإجرائية, أي لا يوجد أي تعليمة أخرى بعد تنفيذ الاستدعاء ضمن نص الإجرائية (بغض النظر عن موضع الاستدعاء العودي ضمن الإجرائية).

في حالة كون الاستدعاء العودي متطرفاً يمكن تحويله إلى خوارزمية تكرارية بالاستعانة بتعليمة goto أو حلقة تكرارية مثل حلقة while فيصبح الشكل العام للتحويل كالمخطط التالي :



حيث إن X قائمة من متحولات الدخل, Y قائمة من متحولات الخرج, والاستدعاء العودي p(E,Y) متطرف.

3-5-2 أمثلة على طريقة حذف الاستدعاء العودي المتطرف:

مثال (1): حساب العاملي لعدد صحيح:

الخوارزمية العودية	استخدام goto label	استخدام حلقة while

long fact(int n)
{
if (n==1||n==0)
return 1;
else
return n*fact(n-1);
}
استدعاء عودي متطرف
	long fact(int n)
	{ long f=1;
	Start:
	if (n==1||n==0)
	return f;
	else{
	f*=n;
	--n;
	goto Start;
	}
	}
		long fact(int n)
		{
		long f=1;
		while(n!=0){
		f*=n;
		--n;
		}
		return f;
		}

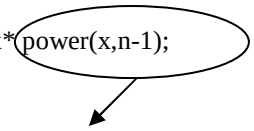
مثال (2): إيجاد القاسم المشترك الأكبر لعددين صحيحين:

الخوارزمية العودية	استخدام goto label	استخدام حلقة while

int gcd(int x,int y)
{
if (y==0)
return x;
else
return gcd(y,x%y);
}
 
استدعاء عودي متطرف
	int gcd(int x,int y)
	{
	int h;
	Start:
	if (y==0) return x;
	else{
	h=x; x=y;
	y=h%y;
	goto Start;
	}
	}
		int gcd(int x,int y)
		{
		int h;
		while(y!=0){
		h=x;
		x=y;
		y=h%y;
		}
		return x;
		}

مثال (3): حساب رفع قوة لعدد صحيح:

الخوارزمية العودية	استخدام goto label	استخدام حلقة while

int power(int x,int n)
{
if (n==0)
return 1;
else
return x*power(x,n-1);
}
 
استدعاء عودي متطرف
	int power(int x,int n)
	{ int p=1;
	Start:
	if (n==0) return p;
	else{
	p*=x;
	--n;
	goto Start;
	}
	}
		int power(int x,int n)
		{
		int p=1;
		while(n!=0){
		p*=x;
		--n;
		}
		return p;
		}

مثال (4): مسألة أبراج هانوي:

الخوارزمية العودية

```
void Hanoi(int n,char a,char b,char c)
{
 if (n==1) cout<<a<<"  -->  "<<c<<endl;
 else{
     Hanoi(n-1,a,c,b);
          cout<<a<<"  -->  "<<c<<endl;
          Hanoi(n-1,b,a,c);
 }
}
```

استدعاء عودي متطرف

goto label

```
void Hanoi(int n,char a,char b,char c)
{ char h;
Start:
 if (n==1) cout<<a<<"  -->  "<<c<<endl;
 else{
     Hanoi(n-1,a,c,b);
          cout<<a<<"  -->  "<<c<<endl;
          n--;
          h=a; a=b; b=h;
          goto Start;
 }
}
```

While

```
void Hanoi(int n,char a,char b,char c)
{ char h;
 while(n!=1){
   Hanoi(n-1,a,c,b);
   cout<<a<<"  -->  "<<c<<endl;
   n--;
   h=a; a=b; b=h;
 }
 cout<<a<<"  -->  "<<c<<endl;
}
```

مثال (5): إيجاد ناتج جداء عددين صحيحين:

الخوارزمية العودية	استخدام goto label	استخدام حلقة while

int multi(int x,int y)
{
if (y==0) return 0;
else
return x+multi(x,y-1);
}
استدعاء متطرف
	int multi(int x,int y)
	{ int m=0;
	Start:
	if (y==0) return m;
	else{
	m+=x;
	y--;
	goto Start;
	}
	}
		int multi(int x,int y)
		{
		int m=0;
		while(y!=0){
		m+=x;
		y--;
		}
		return m;
		}

3-5-3 الطريقة العامة لحذف الاستدعاء العودي غير المتطرف:

نقول عن استدعاء عودي إنه غير متطرف إذا كان يوجد بعد هذا الاستدعاء تعليمة واحدة على الأقل.

إن الشكل العام للتحويل كالتالي (حيث إنه الاستدعاء العودي P( ) ليس متطرفاً, A و B و C كتل من التعليمات وحيث إنه لا توجد وسطاء تمرير للإجراء):

إن لم يوجد وسطاء تمرير (= معطيات جديدة) فيتم في كل استدعاء عودي إعادة تنفيذ التعليمات نفسها تماماً دون أي فرق, لذلك يمكن تحديد label في نقطة معينة تناسب المسألة المطروحة, ولكن بما أننا نذهب في كل مرة إلى هذا الـــ label مقابل كل استدعاء عودي فمن المستحيل متابعة تنفيذ التعليمات الموجودة ما بعد الاستدعاء العودي (B) لأن تعليمة goto تقطع لنا في كل مرة الطريق الواصل إلى تلك التعليمات (B), لذلك يجب إضافة label آخر يعطي مكان هذه التعليمات المقطوعة. من الواضح أنه يجب تنفيذ (B) لأول مرة بعد أن نصل إلى شرط التوقف (أي بعد تنفيذ (C) التي لا تنفذ إلا مرة واحدة فقط !!!). عدد مرات تنفيذ (B) يتوقف على عدد الاستدعاءات العودية, لذلك قمنا بإضافة عدّاد في بداية الخوارزمية يقوم بعملية العد وبالتالي فإن (B) تنفذ بقيمة العداد مرة.

طريقة ثانية للحذف	حذف الاستدعاء العودي	الخوارزمية بشكل عودي

void P()
{
int i=0;
while(condition){
A ;
++i;
}
 	
C;
 	
while(i>0){
B ;
--i;
}
}
	void P()
	{
	int i=0;
	Start:
	if (condition){
	A ;
	++i;
	goto Start;
	L1: B ;
	--i;
	}
	else
	C ;
	
	if (i>0) goto L1;
	}
		void P()
		{
		if (condition){
		A ;
		P();
		B ;
		}
		else
		C ;
		}

ملاحظة: في حال وجود عدة استدعاءات عودية يمكن استخدام عدة labels وتوزيعها بالشكل المناسب, بالإضافة إلى استخدام مكدس نضع فيه أرقاماً تدل على الاستدعاءات العودية, ونستعمل هذه القيم في تحديد آخر استدعاء مؤجل.

ملاحظة: عندما توجد وسطاء تمرير للتابع العودي فإن في كل استدعاء عودي يتم تنفيذ التعليمات نفسها ولكن من أجل معطيات مختلفة, فإذا كان لدينا تعليمات ما بعد الاستدعاء العودي متعلقة بهذه المعطيات, يجب علينا حفظ هذه المعطيات قبل تغييرها من أجل إمكانية استرجاعها في أثناء طريق العودة من استدعاء ابن إلى استدعاء أب. إذاً هنا لا يكفينا عدّاد يقوم بعد عدد مرات الاستدعاءات العودية, بل يمكن استخدام مفهوم المكدسات لتخزين المعطيات مؤقتاً فيها. إن المثال التالي يبيّن هذه الفكرة:

3-6 تمارين الفصل الثالث

تمرين 1: تستخدم قاعدة هورنر (Horner's rule) العدد الأدنى للجداءات في تقييم كثير حدود عند نقطة a . ليكن كثير الحدود $A(x)=\sum_{i=0}^{n} a_i x^i$ ، قاعدة هورنر هي :

$$A(x_0)=(..... (a_n x_0 + a_{n-1})x_0 + ... + a_1)x_0 + a_0$$

اكتب دالة إجرائية تقيم كثير حدود مستخدماً قاعدة هورنر.

تمرين 2: اكتب دالتين إجرائيتين الأولى تكرارية و الثانية عودية لحساب التوافيق $\binom{n}{m}$ المعطى بالشكل:

$$\binom{n}{m}=\binom{n}{m}+\binom{n}{m}=\frac{n!}{m!(n-m)!}, \quad where \ \binom{n}{0}=\binom{n}{n}=1$$

تمرين3: لتكن S مجموعة مكونة من n عنصراً ، تعرف قوة المجموعة S (power set) بأنها مجموعة كل المجموعات الجزئية من S . على سبيل المثال ، إذا كانت S={a, b, c} ، عندئذ قوة المجموعة S تعطى بالشكل :

Power set (S)={{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}

المطلوب : اكتب دالة عودية تحسب power set (S) .

تمرين 4: اكتب دالتين إجرائيتين الأولى عودية و الثانية تكرارية لحساب دالة Ackermann المعرفة على مجموعة الأعداد الطبيعية كما يلي :

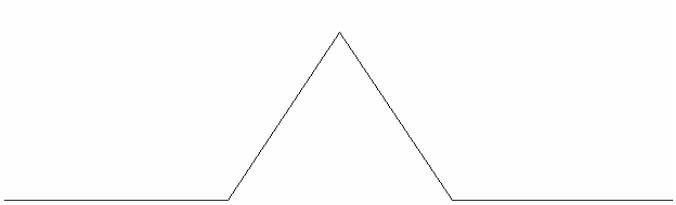
$$A(m,n)=\begin{cases} n+1 & if \ m=0 \\ A(m-1,1) & if \ n=0 \\ A(m-1,A(m,n-1)) & Otherwise \end{cases}$$

تمرين 5: اكتب دوال إجرائية لرسم المنحنيات التالية اعتماداً على مفهوم العودية.

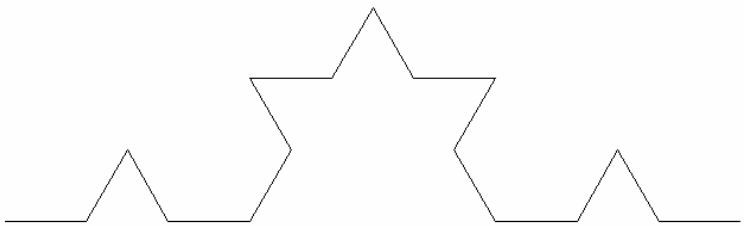
(ابدأ برسم الأشكال الأساسية التي تتألف منها المنحنيات, ثم حدد المخطط العام لكل شكل)

S1:

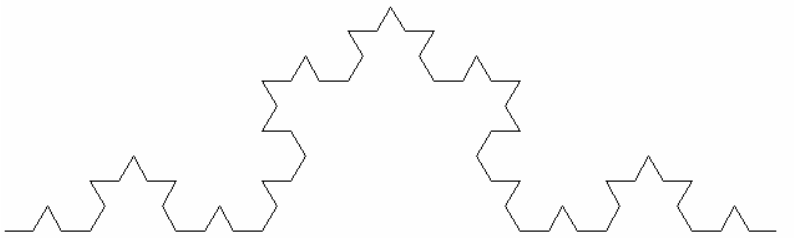
S2:



S3:



S4:



تمرين 6: إذا كان التابع *f* يملك n دخلاً مختلفاً و قيم خرجه مختلفة و عددها أقل مـــن n ، عندئذ يوجد دخلان *a, b* بحيث $a \neq b$ و $f(a)=f(b)$. اكتب دالة إجرائية لإيجاد *a, b* بحيث إن $f(a)=f(b)$. افرض أن دخل التابع هي n ,..... 2, 1 .

تمرين 7: قدم خوارزمية لتوليد كل التباديل لمجموعة ما مكونة من n عنصراً ، ثم احسب التعقيد الزمني للخوارزمية المقدمة .

تمرين 8: لنفترض النمذجة الخوارزمية التالية للعبة Baguenaudier : ليكن لدينا الجدول T[1 … n] من القيم البوليانية ، في الحالة الابتدائية تكون كل القيم في T مساوية لـــ True. الهدف من اللعبة وضع قيمة False في جميع خانات الجدول ، مع العلم أنه لا يمكن تبديل قيمة خانة في الجدول إلا في الحالات التالية :

- يمكن دائما تبديل (قلب) قيمة T(1) .
- يمكن تبديل قيمة T(2) إذا كانت : T(1)=True .
- في حالة j>2 يمكن تبديل قيمة T[j] إذا كان :

T[j-1]=True; T[1]=T[2]= … =T[j-2]=False;

و المطلوب :

1. قدم بنى المعطيات المناسبة
2. اكتب الدالة الإجرائية العودية Bag(m) حيث الحالة الابتدائية :

T[1]=T[2]= … =T[m]=True

و الحالة النهائية المطلوبة هي : T[1]=T[2]= … =T[m]=False

3. اكتب الدالة الإجرائية العودية DeBag(m) حيث الحالة الابتدائية :

T[1]=T[2]= … =T[m]=False

و الحالة النهائية المطلوبة هي : T[1]=T[2]= … =T[m]=True

4. اكتب دالة إجرائية تكرارية ItrBag(m) تقوم بعمل الإجرائية العودية Bag(m) نفسه .