

# IFT159

## Analyse et programmation

### Chapitre 11 — Récursivité

Gabriel Girard

Département d'informatique



23 novembre 2008



Analyse et programmation

1/59

## Chapitre 11 — Récursivité

- 1 Concept de Récursivité
- 2 Exemple 1 : Factoriel
- 3 Exemple 2 : Multiplication
- 4 Exemple 3 : Tours de Hanoï
- 5 Exemple 4 : Les tris
  - Tri bulle
  - Tri fusion
- 6 Exemple 5 : Blob
- 7 Exemple 6 : Sierpinski Gasket



Analyse et programmation

2/59

## Définition de la Récursivité

### Définition

Une fonction est récursive si elle se rappelle elle-même afin de répéter un certain nombre de fois un traitement avec des données différentes.



Analyse et programmation

4/59

## Caractéristiques d'un problème récursif

- 1 Un ou plusieurs cas du problème (appelé des cas d'arrêt) ont des solutions directes.
- 2 Pour les autres cas, il y a une méthode pour réduire le problème afin qu'il s'approche progressivement des cas d'arrêt.



Analyse et programmation

5/59

## Forme d'un algorithme récursif

- > Si c'est un cas d'arrêt
- alors on résout le problème
  - sinon on le réduit grâce à la récursivité

## Factoriel : algorithme

- $\text{fact}(0) = 1$  (cas d'arrêt)
- $\text{fact}(n) = n * \text{fact}(n-1)$  (décomposition)

## Factoriel : implémentation

```
#include <iostream.h>

main()
{
    int fact(int);
    int nombre, factoriel;

    cout << " Entrez le nombre : " ;
    cin >> nombre;

    factoriel = fact(nombre);

    cout << " le factoriel de " << nombre
         << " est : " << factoriel << endl;

    return 0;
}
```

## Factoriel : implémentation

```
int fact(int nb)
{
    int factoriel;

    if (nb == 1)
        factoriel = 1;
    else factoriel = nb * fact(nb-1);

    return factoriel;
}
```

## Factoriel : trace d'exécution

Entrez le nombre dont on doit calculer le factoriel : 7

```

1 - factoriel(7)
  2 - factoriel(6)
    3 - factoriel(5)
      4 - factoriel(4)
        5 - factoriel(3)
          6 - factoriel(2)
            7 - factoriel(1)
              7 - retour = 1
            6 - retour = 2
          5 - retour = 6
        4 - retour = 24
      3 - retour = 120
    2 - retour = 720
  1 - retour = 5040

```

le factoriel de 7 est : 5040

## Multiplication : algorithme

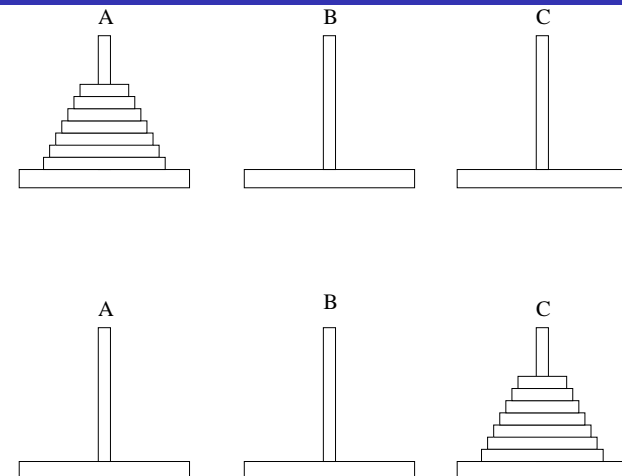
- $\text{mult}(n,1) = n$  (cas d'arrêt)
- $\text{mult}(n,m) = n + \text{mult}(n,m-1)$  (décomposition)

## Tours de Hanoi : description du problème

On doit déplacer un nombre donné d'anneaux de tailles différentes d'une tour (la tour originale) à une autre tour (la tour cible) en respectant les contraintes suivantes :

- seulement un anneau est déplacé à la fois et il doit être au sommet d'une tour ;
- un anneau ne peut être placé sur un autre anneau de plus petite taille.

## Tours de Hanoi : description du problème



## Tours de Hanoi : algorithme

- 1 le cas simple (cas d'arrêt) est de déplacer un anneau.
- 2 il reste à déplacer  $n-1$  disques.

17/59

## Tours de Hanoi : algorithme

- 1 on déplace  $n - 1$  anneaux de A vers B ;
- 2 on déplace le dernier anneau de A vers C ;
- 3 on déplace  $n - 1$  anneaux de B vers C ;

18/59

## Tours de Hanoi : algorithme

On reprend les étapes 1 et 3.

Pour déplacer  $n - 1$  anneaux de A vers B :

- 1 déplace  $n - 2$  anneaux de A vers C ;
- 2 déplace l'anneau restant de A vers B ;
- 3 déplace les  $n - 2$  anneaux de C vers B ;

19/59

## Tours de Hanoi : algorithme

Algorithme pour déplacer  $n$  étages de  $T_o$  vers  $T_c$  en utilisant  $T_a$

- 1 s'il n'y a qu'une tour à déplacer,
  - 1 la déplacer de  $T_o$  vers  $T_c$
- 2 sinon
  - 1 déplacer  $n - 1$  étages de  $T_o$  vers  $T_a$  en utilisant  $T_c$
  - 2 déplacer 1 étage de  $T_o$  vers  $T_c$
  - 3 déplacer  $n - 1$  étages de  $T_a$  vers  $T_c$  en utilisant  $T_o$

20/59

## Tours de Hanoi : implémentation

```

void hanoi(char tr_orig, char tr_cible, char tr_aux, int n)
{
    int i;
    if (n == 1)
    { cout << " - Deplacement de l'anneau " << n
      << " de la tour " << tr_orig << " vers la tour "
      << tr_cible << ".\n";
    }
    else {
        hanoi(tr_orig, tr_aux, tr_cible, n-1);
        cout << " - Deplacement de l'anneau " << n
              << " de la tour " << tr_orig << " vers la tour "
              << tr_cible << ".\n";
        hanoi(tr_aux, tr_cible, tr_orig, n-1);
    }
}

```

## Tours de Hanoi : implémentation

```
/* Programme principal */
```

```

int main()
{
    void hanoi (char, char, char, int);

    int n;
    cout << "Probleme de la tour de Hanoi. Entrez N : ";
    cin >> n;
    cout << n << endl;
    hanoi('A', 'C', 'B', n);

    return 0;
}

```

## Tours de Hanoi : trace de l'exécution

Probleme de la tour de Hanoi. Entrez N : 3

```

1 - hanoi (A, C, B, 3)
  2 - hanoi (A, B, C, 2)
    3 - hanoi (A, C, B, 1)
      3 - Deplacement de l'anneau 1 de la tour A vers la tour C.
    2 - Deplacement de l'anneau 2 de la tour A vers la tour B.
      3 - hanoi (C, B, A, 1)
        3 - Deplacement de l'anneau 1 de la tour C vers la tour B.
    1 - Deplacement de l'anneau 3 de la tour A vers la tour C.
  2 - hanoi (B, C, A, 2)
    3 - hanoi (B, A, C, 1)
      3 - Deplacement de l'anneau 1 de la tour B vers la tour A.
    2 - Deplacement de l'anneau 2 de la tour B vers la tour C.
      3 - hanoi (A, C, B, 1)
        3 - Deplacement de l'anneau 1 de la tour A vers la tour C.

```

## Les tris

1 tri bulle récursif

2 tri fusion

## Tri bulle itératif

```

void tri_bulle(int nb, int tab[]){
    void swap(int&, int&);

    for (int k = nb; k > 1; k--){
        for (int i = 1; i < k; i++){
            if ( tab[i-1] > tab[i] ){
                swap(tab[i-1], tab[i]);
            }
        }
    }

    return;
}

```

26/59

## Tri bulle récursif

```

void tri_bulle(int nb, int tab[])
{
    void swap(int&, int&);

    for (int i = 1; i < nb; i++){
        if ( tab[i-1] > tab[i] ){
            swap(tab[i-1], tab[i]);
        }
    }
    tri_bulle(nb-1, tab);

    return;
}

```

27/59

## Tri fusion : utilisation

```

#define MAX 20
void main()
{
    int tri(int [],int );
    int tab[MAX],i, code;

    cout << "Entrez les chiffres a trier : " << endl;
    cin >> tab[0];
    i = 0;
    while (tab[i] != 0 && i < MAX-1)
        { i++ ; cin >> tab[i];}
    cout << endl << endl;

    code = tri(tab,i);

    if (code ==0)
    {   for ( int j = 0; j < i; j++) cout << " " << tab[j];
        cout << endl;
    }
}

```

28/59

## Tri fusion : implémentation

```

int tri(int liste[],int longueur)
{
    int liste1[MAX], liste2[MAX];
    int long1, long2;
    int index1,index2;

    if (longueur == 0) return 1;
    if (longueur == 1) return 0;

    long1 = longueur / 2;
    long2 = longueur - long1;
    for(int i =0; i<long1; i++) liste1[i] = liste[i];
    for(int j = 0; j < long2; j++) liste2[j] = liste[i+j];

    tri(liste1,long1);
    tri(liste2,long2);
}

```

29/59

## Tri fusion : implémentation

```

index1 = 0;
index2 = 0;
for(int index=0; index1<long1 && index2<long2; index++)
{ if (liste1[index1] >= liste2[index2])
  { liste[index]=liste1[index1];
    index1++;
  }
  else {
    liste[index]=liste2[index2];
    index2++;
  }
}
while (index1<long1) { liste[index]=liste1[index1];
                      index++; index1++;
                    }
while (index2<long2) { liste[index]=liste2[index2];
                      index++; index2++;
                    }

return 0; }

```

## Tri fusion : trace de l'exécution

Entrez les chiffres à trier : [ 2 5 9 7 1 44 8 ]

Tri liste de longueur 7 -> [ 2 5 9 7 1 44 8 ]

1 - tri liste de longueur 3 -> [ 2 5 9 ]

2 - tri liste de longueur 1 -> [ 2 ]

2 - tri liste de longueur 2 -> [ 5 9 ]

3 - tri liste de longueur 1 -> [ 5 ]

3 - tri liste de longueur 1 -> [ 9 ]

3 - regroupe -> liste longueur 2 => [ 9 5 ]

2 - regroupe -> liste longueur 3 => [ 9 5 2 ]

1 - tri liste de longueur 4 -> [ 7 1 44 8 ]

2 - tri liste de longueur 2 -> [ 7 1 ]

3 - tri liste de longueur 1 -> [ 7 ]

3 - tri liste de longueur 1 -> [ 1 ]

3 - regroupe -> liste longueur 2 => [ 7 1 ]

## Tri fusion : trace de l'exécution

2 - tri liste de longueur 2 -> [ 44 8 ]

3 - tri liste de longueur 1 -> [ 44 ]

3 - tri liste de longueur 1 -> [ 8 ]

3 - regroupe -> liste longueur 2 => [ 44 8 ]

2 - regroupe -> liste longueur 4 => [ 44 8 7 1 ]

1 - regroupe -> liste longueur 7 => [ 44 9 8 7 5 2 1 ]

Liste triée

44 9 8 7 5 2 1

## Blob : description

- On vous demande d'écrire un programme qui compte le nombre de points dans une figure dessinée sur un tableau en 2D.
- Les cellules pleines peuvent être connectées pour former une figure.
- Deux cellules sont connectées si elles se touchent horizontalement, verticalement ou diagonalement.

## Blob : description

1	1	0	0	0	1	1	1
1	1	0	0	0	0	0	1
1	0	0	1	0	0	0	1
1	0	0	1	1	0	0	0
1	0	1	1	1	1	0	0
1	0	0	1	1	0	0	0
1	0	0	1	0	0	1	1
1	1	0	0	0	1	0	1

## Blob : algorithme

## ■ Algorithme

## ■ Cas d'arrêt :

- 1 la cellule est vide — on retourne 0
- 2 la cellule n'est pas dans la grille — on retourne 0

## ■ Décomposition :

Si une cellule n'est pas vide, on retourne 1 plus le nombre de cellule associées aux voisins de la cellule courante.

## ■ Contraintes

\*\* Pour s'assurer que l'on ne compte pas deux fois la même cellule, lorsqu'on la compte on la marque comme vide ou avec une valeur égale à -1 (si on veut retrouver la figure originale).

## Blob : algorithme

On reçoit les coordonnées en x et en y

Si (x,y) est vide retourne 0

Si (x,y) est hors de la grille retourne 0

Si (x,y) est remplie

retourne 1 + compte des cellules de ses voisins

## Blob : implémentation

```
#define MAX_Y 8
#define MAX_X 8
int main()
{
    int cpt,x,y;
    int grille[MAX_X][MAX_Y] = { 1,1,0,0,0,1,1,1,
                                  1,1,0,0,0,0,0,1,
                                  1,0,0,1,0,0,0,1,
                                  1,0,0,1,1,0,0,0,
                                  1,0,1,1,1,1,0,0,
                                  1,0,0,1,1,0,0,0,
                                  1,0,0,1,0,0,1,1,
                                  1,1,0,0,0,1,0,1  };
}
```

```
int fig_cpt(int [MAX_X][MAX_Y], int, int);
```

## Blob : implémentation

```

cout << "Voici la grille." << endl << endl;
for(int i=0; i< MAX_X; i++) {
    cout << "          ";
    for(int j=0; j< MAX_Y; j++)
        cout << grille[i][j] << " ";
    cout << endl;
}

cout << endl << endl
    << "Entrer les coordonnees en X et en Y : ";
cin >> x >> y;

cpt = fig_cpt(grille, x, y);

cout << "Le nombre de cellule dans la figure est "
    << cpt << endl<<endl;
}

```

## Blob : implémentation

```

int fig_cpt(int grille[MAX_X][MAX_Y], int x, int y)
{
    int compte;

    if (x<0 || (x > MAX_X -1) || y < 0 || (y > MAX_Y -1))
        compte = 0;
    else if (grille[x][y] == 0)    compte = 0;
    else {
        grille[x][y] = 0;
        compte = 1 + fig_cpt(grille, x-1, y) +
            fig_cpt(grille, x-1, y+1) +
            fig_cpt(grille, x-1, y-1) +
            fig_cpt(grille, x, y+1) +
            fig_cpt(grille, x, y-1) +
            fig_cpt(grille, x+1, y+1) +
            fig_cpt(grille, x+1, y) +
            fig_cpt(grille, x+1, y-1) ;
    }
    return compte; }

```

## Blob : trace de l'exécution

Voici la grille.

```

1 1 0 0 0 1 1 1
1 1 0 0 0 0 0 1
1 0 0 1 0 0 0 1
1 0 0 1 1 0 0 0
1 0 1 1 1 1 0 0
1 0 0 1 1 0 0 0
1 0 0 1 0 0 1 1
1 1 0 0 0 1 0 1

```

## Blob : trace de l'exécution

```

Entrer les coordonnees en X et en Y : 7 7
Premier appel -> fig_cpt(grille, 7, 7)
  1 - fig_cpt(grille, 7, 7)
    2 - fig_cpt(grille, 6, 7)
      3 - fig_cpt(grille, 5, 7) +++++ retourne 0
      3 - fig_cpt(grille, 5, 8) +++++ retourne 0
      3 - fig_cpt(grille, 5, 6) +++++ retourne 0
      3 - fig_cpt(grille, 6, 8) +++++ retourne 0
      3 - fig_cpt(grille, 6, 6)
        4 - fig_cpt(grille, 5, 6) +++++ retourne 0
        4 - fig_cpt(grille, 5, 7) +++++ retourne 0
        4 - fig_cpt(grille, 5, 5) +++++ retourne 0
        4 - fig_cpt(grille, 6, 7) +++++ retourne 0
        4 - fig_cpt(grille, 6, 5) +++++ retourne 0
        4 - fig_cpt(grille, 7, 7) +++++ retourne 0
        4 - fig_cpt(grille, 7, 6) +++++ retourne 0
        4 - fig_cpt(grille, 7, 5)

```

## Blob : trace de l'exécution

```

5 - fig_cpt(grille, 6, 5) +++++ retourne 0
5 - fig_cpt(grille, 6, 6) +++++ retourne 0
5 - fig_cpt(grille, 6, 4) +++++ retourne 0
5 - fig_cpt(grille, 7, 6) +++++ retourne 0
5 - fig_cpt(grille, 7, 4) +++++ retourne 0
5 - fig_cpt(grille, 8, 6) +++++ retourne 0
5 - fig_cpt(grille, 8, 5) +++++ retourne 0
5 - fig_cpt(grille, 8, 4) +++++ retourne 0

```

```

4 retour --- 1

```

```

3 retour --- 2

```

```

3 - fig_cpt(grille, 7, 8) +++++ retourne 0
3 - fig_cpt(grille, 7, 7) +++++ retourne 0
3 - fig_cpt(grille, 7, 6) +++++ retourne 0

```

```

2 retour --- 3

```

## Blob : trace de l'exécution

```

2 - fig_cpt(grille, 6, 8) +++++ retourne 0
2 - fig_cpt(grille, 6, 6) +++++ retourne 0
2 - fig_cpt(grille, 7, 8) +++++ retourne 0
2 - fig_cpt(grille, 7, 6) +++++ retourne 0
2 - fig_cpt(grille, 8, 8) +++++ retourne 0
2 - fig_cpt(grille, 8, 7) +++++ retourne 0
2 - fig_cpt(grille, 8, 6) +++++ retourne 0

```

```

1 retour --- 4

```

Le nombre de cellule dans la figure est 4

## Blob : trace de l'exécution

Voici la grille.

```

1 1 0 0 0 1 1 1
1 1 0 0 0 0 0 1
1 0 0 1 0 0 0 1
1 0 0 1 1 0 0 0
1 0 1 1 1 1 0 0
1 0 0 1 1 0 0 0
1 0 0 1 0 0 1 1
1 1 0 0 0 1 0 1

```

## Blob : trace de l'exécution

Entrer les coordonnees en X et en Y : 0 0

Premier appel -> fig\_cpt(grille, 0, 0)

```

1 - fig_cpt(grille, 0, 0)
2 - fig_cpt(grille, -1, 0)  +---- retour 0
2 - fig_cpt(grille, -1, 1)  +---- retour 0
2 - fig_cpt(grille, -1, -1) +---- retour 0
2 - fig_cpt(grille, 0, 1)
3 - fig_cpt(grille, -1, 1)  +---- retour 0
3 - fig_cpt(grille, -1, 2)  +---- retour 0
3 - fig_cpt(grille, -1, 0)  +---- retour 0
3 - fig_cpt(grille, 0, 2)   +---- retour 0
3 - fig_cpt(grille, 0, 0)   +---- retour 0
3 - fig_cpt(grille, 1, 2)   +---- retour 0
3 - fig_cpt(grille, 1, 1)
4 - fig_cpt(grille, 0, 1)   +---- retour 0
4 - fig_cpt(grille, 0, 2)   +---- retour 0
4 - fig_cpt(grille, 0, 0)   +---- retour 0
4 - fig_cpt(grille, 1, 2)   +---- retour 0
4 - fig_cpt(grille, 1, 0)

```

## Blob : trace de l'exécution

```

5 - fig_cpt(grille, 0, 0)  ++++ retour 0
5 - fig_cpt(grille, 0, 1)  ++++ retour 0
5 - fig_cpt(grille, 0, -1) ++++ retour 0
5 - fig_cpt(grille, 1, 1)  ++++ retour 0
5 - fig_cpt(grille, 1, -1) ++++ retour 0
5 - fig_cpt(grille, 2, 1)  ++++ retour 0
5 - fig_cpt(grille, 2, 0)
6 - fig_cpt(grille, 1, 0)  ++++ retour 0
6 - fig_cpt(grille, 1, 1)  ++++ retour 0
6 - fig_cpt(grille, 1, -1) ++++ retour 0
6 - fig_cpt(grille, 2, 1)  ++++ retour 0
6 - fig_cpt(grille, 2, -1) ++++ retour 0
6 - fig_cpt(grille, 3, 1)  ++++ retour 0
6 - fig_cpt(grille, 3, 0)
7 - fig_cpt(grille, 2, 0)  ++++ retour 0
7 - fig_cpt(grille, 2, 1)  ++++ retour 0
7 - fig_cpt(grille, 2, -1) ++++ retour 0
7 - fig_cpt(grille, 3, 1)  ++++ retour 0
7 - fig_cpt(grille, 3, -1) ++++ retour 0
7 - fig_cpt(grille, 4, 1)  ++++ retour 0
7 - fig_cpt(grille, 4, 0)

```

## Blob : trace de l'exécution

```

8 - fig_cpt(grille, 3, 0)  ++++ retour 0
8 - fig_cpt(grille, 3, 1)  ++++ retour 0
8 - fig_cpt(grille, 3, -1) ++++ retour 0
8 - fig_cpt(grille, 4, 1)  ++++ retour 0
8 - fig_cpt(grille, 4, -1) ++++ retour 0
8 - fig_cpt(grille, 5, 1)  ++++ retour 0
8 - fig_cpt(grille, 5, 0)
9 - fig_cpt(grille, 4, 0)  ++++ retour 0
9 - fig_cpt(grille, 4, 1)  ++++ retour 0
9 - fig_cpt(grille, 4, -1) ++++ retour 0
9 - fig_cpt(grille, 5, 1)  ++++ retour 0
9 - fig_cpt(grille, 5, -1) ++++ retour 0
9 - fig_cpt(grille, 6, 1)  ++++ retour 0
9 - fig_cpt(grille, 6, 0)
10 - fig_cpt(grille, 5, 0) ++++ retour 0
10 - fig_cpt(grille, 5, 1) ++++ retour 0
10 - fig_cpt(grille, 5, -1) ++++ retour 0
10 - fig_cpt(grille, 6, 1) ++++ retour 0
10 - fig_cpt(grille, 6, -1) ++++ retour 0
10 - fig_cpt(grille, 7, 1)

```

## Blob : trace de l'exécution

```

11 - fig_cpt(grille, 6, 1) ++++ retour 0
11 - fig_cpt(grille, 6, 2) ++++ retour 0
11 - fig_cpt(grille, 6, 0) ++++ retour 0
11 - fig_cpt(grille, 7, 2) ++++ retour 0
11 - fig_cpt(grille, 7, 0)
12 - fig_cpt(grille, 6, 0)  +++ retour 0
12 - fig_cpt(grille, 6, 1)  +++ retour 0
12 - fig_cpt(grille, 6, -1) +++ retour 0
12 - fig_cpt(grille, 7, 1)  +++ retour 0
12 - fig_cpt(grille, 7, -1) +++ retour 0
12 - fig_cpt(grille, 8, 1)  +++ retour 0
12 - fig_cpt(grille, 8, 0)  +++ retour 0
12 - fig_cpt(grille, 8, -1) +++ retour 0
11 retour --- 1
11 - fig_cpt(grille, 8, 2) ++++ retour 0
11 - fig_cpt(grille, 8, 1) ++++ retour 0
11 - fig_cpt(grille, 8, 0) ++++ retour 0

```

## Blob : trace de l'exécution

```

10 retour --- 2
10 - fig_cpt(grille, 7, 0)  ++++ retour 0
10 - fig_cpt(grille, 7, -1) ++++ retour 0
9 retour --- 3
9 - fig_cpt(grille, 6, -1) ++++ retour 0
8 retour --- 4
8 - fig_cpt(grille, 5, -1) ++++ retour 0
7 retour --- 5
7 - fig_cpt(grille, 4, -1) ++++ retour 0
6 retour --- 6
6 - fig_cpt(grille, 3, -1) ++++ retour 0
5 retour --- 7
5 - fig_cpt(grille, 2, -1) ++++ retour 0
4 retour --- 8
4 - fig_cpt(grille, 2, 2) ++++ retour 0
4 - fig_cpt(grille, 2, 1) ++++ retour 0
4 - fig_cpt(grille, 2, 0) ++++ retour 0

```

## Blob : trace de l'exécution

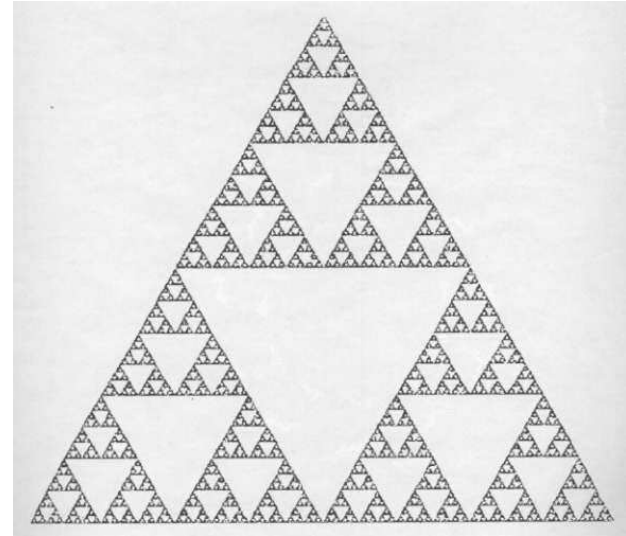
```

3 retour --- 9
3 - fig_cpt(grille, 1, 0) ++++ retour 0
2 retour --- 10
2 - fig_cpt(grille, 0, -1) ++++ retour 0
2 - fig_cpt(grille, 1, 1) ++++ retour 0
2 - fig_cpt(grille, 1, 0) ++++ retour 0
2 - fig_cpt(grille, 1, -1) ++++ retour 0
1 retour --- 11
Le nombre de cellule dans la figure est 11

```

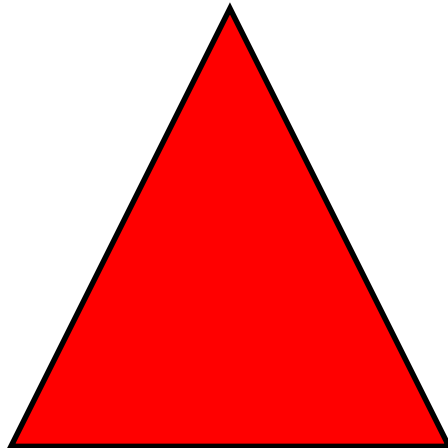
51/59

## Sierpinski Gasket : description



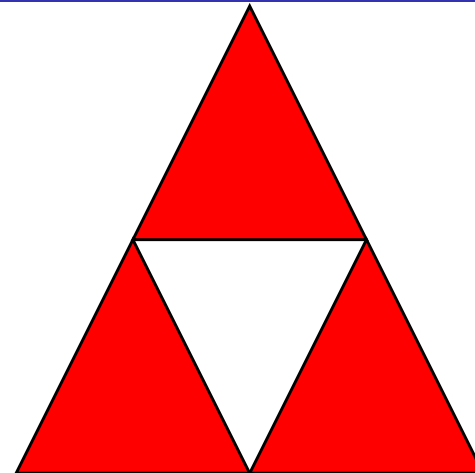
53/59

## Sierpinski Gasket : description



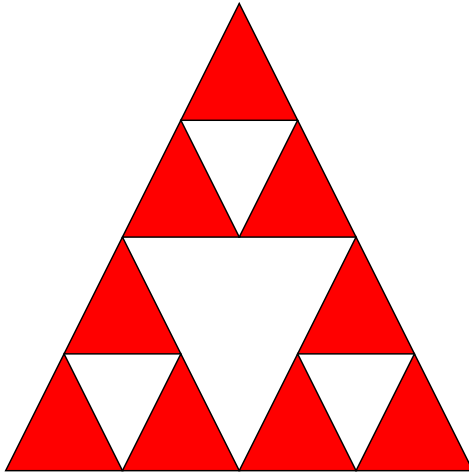
54/59

## Sierpinski Gasket : description

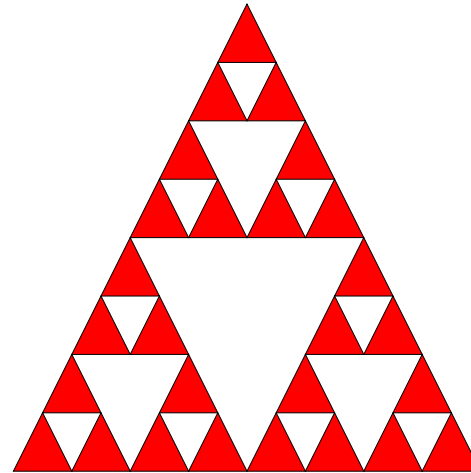


55/59

## Sierpinski Gasket : description



## Sierpinski Gasket : description



## Sierpinski Gasket : implémentation

```
drawGasket(point p1, point p2, point p3, int level)
{
    if ( level == 0 )
    {
        // dessiner un triangle plein
        drawTriangle(p1, p2, p3);
    }
    else
    {
        // faire les trois sous-paniers de Sierpinski
        level = level - 1;
        drawGasket(p1, (p1 + p2)/2, (p1 + p3)/2, level);
        drawGasket((p2 + p1)/2, p2, (p2 + p3)/2, level);
        drawGasket((p3 + p1)/2, (p3 + p2)/2, p3, level);
    }
}
```

## Exercice

- 1 Écrivez un programme récursif qui compte le nombre de chemins pour traverser un labyrinthe. Le labyrinthe est représenté par un grille de 8 X 8. Un chemin permet de traverser le labyrinthe de la case (0,0) jusqu'à la case (7,7).
- 2 Écrivez un programme récursif qui implante la recherche dichotomique.
- 3 Écrivez un programme récursif qui trouve le « plus grand commun diviseur ».
- 4 Écrivez un programme récursif qui inverse une chaîne.
- 5 Écrivez un programme récursif qui détermine si une chaîne est un palindrome.
- 6 Écrivez un programme récursif qui fait la somme des éléments d'une liste.