11X001 – Introduction à la programmation des algorithmes

2.3b Fonctions Récursives

François Fleuret

https://fleuret.org/francois

29 Septembre, 2020

Le contenu de ce document a été en grande partie repris du cours de Jean-Luc Falcone.

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Sémantique d'évaluation

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
3 > 13
```

Substitutions: évaluations d'expressions

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
3 > 13
false
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
z * 2
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
z * 2
( x * x - y ) * 2
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
( 5 * 5 - (5 - 2) ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
( 5 * 5 - (5 - 2) ) * 2
( 5 * 5 - 3 ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
( 5 * 5 - (5 - 2) ) * 2
( 5 * 5 - 3 ) * 2
( 25 - 3 ) * 2
```

Substitutions: constantes

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
( 5 * 5 - (5 - 2) ) * 2
( 5 * 5 - 3 ) * 2
( 25 - 3 ) * 2
22 * 2
```

Une référence à une constante peut également être évaluée par substitution.

Par exemple:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
z * 2
( x * x - y ) * 2
( (1+4) * x - y ) * 2
( (1+4) * (1+4) - y ) * 2
( (1+4) * (1+4) - (x - 2) ) * 2
( (1+4) * (1+4) - ((1+4) - 2) ) * 2
( (1+4) * (1+4) - (5 - 2) ) * 2
( 5 * (1+4) - (5 - 2) ) * 2
( 5 * 5 - (5 - 2) ) * 2
( 5 * 5 - 3 ) * 2
( 25 - 3 ) * 2
22 * 2
44
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 5
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 5
let y = 5 - 2
```

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
let z = 25 - 3
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
let z = 25 - 3
let z = 22
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
let z = 25 - 3
let z = 22
z * 2
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
let z = 25 - 3
let z = 22
z * 2
22 * 2
```

Substitutions: constantes (ordre)

L'ordre de ces substitions n'a pas d'importance.

Par exemple, les définitions peuvent être évaluées à l'avance:

```
let x = 1 + 4
let y = x - 2
let z = x * x - y
```

```
let x = 5
let y = 5 - 2
let y = 3
let z = 5 * 5 - 3
let z = 25 - 3
let z = 22
z * 2
22 * 2
44
```

Substitution expression `if...else`

Pour évaluer un `if...else...` il faut:

1. évaluer la condition,
2. remplacer la structure par la branche correspondant à la condition.
3. continuer en évaluant cette branche

Substitution expression `if...else`: exemple

```
let x = 14
if x-1 >= 13 {
  return -1
} else {
  return 2
}
```

Substitution expression `if...else`: exemple

```
let x = 14
if x-1 >= 13 {
  return -1
} else {
  return 2
}
```

```
if 13 >= 13 {
  return -1
} else {
  return 2
}
```

Substitution expression `if...else`: exemple

```
let x = 14
if x-1 >= 13 {
  return -1
} else {
  return 2
}
```

```
if 13 >= 13 {
  return -1
} else {
  return 2
}
```

```
if true {
  return -1
} else {
  return 2
}
```

Substitution expression `if...else`: exemple

```
let x = 14
if x-1 >= 13 {
  return -1
} else {
  return 2
}
```

```
if 13 >= 13 {
  return -1
} else {
  return 2
}
```

```
if true {
  return -1
} else {
  return 2
}
```

```
return -1
```

Modèle d'évaluation: fonction

Pour évaluer une fonction par substitutions:

1. On substitue l'appel de la fonction par son corps.
2. On remplace chaque occurence de chaque argument par l'expression passée.
3. On poursuit les substitutions jusqu'à parvenir à un `return`.
4. On remplace `{ return x }` par `x`.

Remarque: l'ordre d'évaluation n'a encore pas d'importance !

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
(4 * 1 - (2+2)) + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
(4 * 1 - (2+2)) + 1
(4 - (2+2)) + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
(4 * 1 - (2+2)) + 1
(4 - (2+2)) + 1
(4 - 4) + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
(4 * 1 - (2+2)) + 1
(4 - (2+2)) + 1
(4 - 4) + 1
0 + 1
```

Modèle d'évaluation: exemple 1

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
{ return (2+2) * 1 - (2+2) } + 1
((2+2) * 1 - (2+2)) + 1
(4 * 1 - (2+2)) + 1
(4 - (2+2)) + 1
(4 - 4) + 1
0 + 1
1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
0 + 1
```

Modèle d'évaluation: exemple 2

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
0 + 1
1
```

Récursivité

Somme d'un tableau de nombres

On veut sommer un tableau de nombres:

```
func sum( xs: [Int] ) -> Int {
  if xs.count == 0 {
    return 0
  } else if xs.count == 1 {
    return xs[0]
  } else if xs.count == 2 {
    return xs[0] + xs[1]
  } else if xs.count == 3 {
    return xs[0] + xs[1] + xs[2]
  } else if xs.count == 4 {
  // etc
}
```

Somme d'un tableau de nombres

En programmation procédurale on utiliserait une boucle.

En programmation fonctionnelle on utilise une fonction récursive

C'est à dire une fonction qui s'appelle elle même !

Exemple: fonction factorielle

Fonction $\mathbb{N} \to \mathbb{N}$, définie par:

$$f(n) = \begin{cases} 1 & \text{si } n <= 1, \\ n \cdot f(n-1) & \text{sinon.} \end{cases}$$

Exemple: fonction factorielle

Fonction $\mathbb{N} \to \mathbb{N}$, définie par:

$$f(n) = \begin{cases} 1 & \text{si } n <= 1, \\ n \cdot f(n-1) & \text{sinon.} \end{cases}$$

```
func fact( _ n: UInt64 ) -> UInt64 {
  if n <= 1 {
    return 1
  } else {
    return n * fact( n - 1 )
  }
}
```

Condition de terminaison

En général, une fonction récursive contient une condition de terminaison.

Lorsque celle-ci est vraie, la récursion se termine en retournant une valeur.

```
func fact( _ n: UInt64 ) -> UInt64 {
  if n <= 1 { // Condition de terminaison
    return 1
  } else {
    return n * fact( n - 1 )
  }
}
```

Récursion infinie

Si la condition de terminaison n'est pas atteinte, l'évaluation de la fonction devient infinie.

Est-ce que la fonction `sumAll` ci-dessous se terminera toujours ?

```
// Somme les nombres entiers de 1 à n
func sumAll( _ n: Int ) -> Int {
  if n == 1 {
    return 1
  } else {
    return n + sumAll( n - 1 )
  }
}
```

Combien de chiffres dans un nombre ?

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}

print( digits( 1_000_000 ) )
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}

digits( 987 )
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
```

```
if 987<10 {return 1} else {return 1+digits(987/10)}
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
```

```
if 987<10 {return 1} else {return 1+digits(987/10)}
```

```
{return 1+digits(987/10)}
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
```

```
if 987<10 {return 1} else {return 1+digits(987/10)}
```

```
{return 1+digits(987/10)}
```

```
1 + digits(98)
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
1 + ( 1 + (if 9<10 { return 1} else {return 1+digits(9/10)}))
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
1 + ( 1 + (if 9<10 { return 1} else {return 1+digits(9/10)}))
1 + ( 1 + { return 1 })
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
1 + ( 1 + (if 9<10 { return 1} else {return 1+digits(9/10)}))
1 + ( 1 + { return 1 })
1 + ( 1 + 1 )
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
1 + ( 1 + (if 9<10 { return 1} else {return 1+digits(9/10)}))
1 + ( 1 + { return 1 })
1 + ( 1 + 1 )
1 + 2
```

Combien de chiffres dans un nombre ? Exécution

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

```
digits( 987 )
if 987<10 {return 1} else {return 1+digits(987/10)}
{return 1+digits(987/10)}
1 + digits(98)
1 + (if 98<10 {return 1 } else {return 1+digits(98/10)})
1 + {return 1+digits(98/10)}
1 + ( 1 + digits(98/10) )
1 + ( 1 + digits(9) )
1 + ( 1 + (if 9<10 { return 1} else {return 1+digits(9/10)}))
1 + ( 1 + { return 1 })
1 + ( 1 + 1 )
1 + 2
3
```

Méthode de Héron

On a vu que l'on peut calculer une valeur approximative de la racine carrée de x avec la méthode de Héron d'Alexandrie:

- prendre une approximation initiale arbitraire G,
- améliorer cette approximation avec la moyenne arithmétique entre G et $\frac{x}{G}$,
- continuer jusqu'à atteindre la précision souhaitée.

Par exemple $\sqrt{2}$

$$
\begin{aligned}
x &= 2 & G &= 1 \\
\frac{x}{G} &= 2 & G &= \frac{1+2}{2} = \frac{3}{2} = 1.5 \\
\frac{x}{G} &= \frac{4}{3} & G &= \frac{1}{2}\left(\frac{3}{2} + \frac{4}{3}\right) = \frac{17}{12} \approx 1.416667 \\
\frac{x}{G} &= \frac{24}{17} & G &= \frac{1}{2}\left(\frac{17}{12} + \frac{24}{17}\right) = \frac{577}{408} \approx 1.4142156
\end{aligned}
$$

1. Choisir la précision souhaitée ϵ
2. Tester la qualité de la solution courante, G:

$$|x - G^2| \leq \epsilon$$

Par exemple la solution précédente a une erreur de:

$$\left|2 - \left(\frac{577}{408}\right)^2\right| \approx 6.007 \cdot 10^{-6}$$

Méthode de Héron: Implémentation (1)

```
let eps = 1e-9

func solutionOk( _ G: Double, _ x: Double ) -> Bool {
  return abs(x - G * G) < eps
}

func improve( _ G: Double, _ x: Double ) -> Double {
    return (G + x/G) / 2
}
```

Méthode de Héron: Implémentation (2)

```
func attempt( _ G: Double, _ x: Double ) -> Double {
  if solutionOk( G, x ) {
    return G
  } else {
    let nextG = improve( G, x )
    return attempt( nextG, x )
  }
}

func sqrt( _ x: Double ) -> Double {
  return attempt( 1, x )
}
```

Méthode de Héron: Implémentation alternative

```
func sqrt( _ x: Double ) -> Double {
  let eps = 1e-9

  func improve( _ G: Double ) -> Double {
    return (G + x/G) / 2
  }

  func attempt( _ G: Double ) -> Double {
    if abs(x - G * G) < eps {
      return G
    } else {
      return attempt( improve(G) )
    }
  }

  return attempt( 1 )
}
```

Récursion mutuelle: Pair/Impair

- 0 est pair
- le successeur d'un nombre pair est impair
- le successeur d'un nombre impar est pair

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
```

```
if 2 == 0 { return true } else { return odd(2-1) }
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
if 2 == 0 { return true } else { return odd(2-1) }
odd(1)
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
if 2 == 0 { return true } else { return odd(2-1) }
odd(1)
if 1 == 0 { return false } else { return even(1-1) }
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
if 2 == 0 { return true } else { return odd(2-1) }
odd(1)
if 1 == 0 { return false } else { return even(1-1) }
even(0)
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
if 2 == 0 { return true } else { return odd(2-1) }
odd(1)
if 1 == 0 { return false } else { return even(1-1) }
even(0)
if 0 == 0 { return true } else { return odd(0-1) }
```

Récursion mutuelle: Pair/Impair

```
func even( _ n: UInt ) -> Bool {
  if n == 0 {
    return true
  } else {
    return odd(n - 1)
  }
}
```

```
func odd( _ n: UInt ) -> Bool {
  if n == 0 {
    return false
  } else {
    return even(n - 1)
  }
}
```

```
even(2)
if 2 == 0 { return true } else { return odd(2-1) }
odd(1)
if 1 == 0 { return false } else { return even(1-1) }
even(0)
if 0 == 0 { return true } else { return odd(0-1) }
true
```

Pair/Impair: vraie implémentation

```
func even( _ n: UInt ) -> Bool {
  return n % 2 == 0
}

func odd( _ n: UInt ) -> Bool {
  return ! even(n)
}
```

Somme d'un tableau

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}

print(sum([ 3, 4, 2, -7 ]))
```

Récursion terminale

Fonction factorielle: Évaluation

```
func fact( _ n: UInt64 ) -> UInt64 {
  if n <= 1 {
    return 1
  } else {
    return n * fact( n - 1 )
  }
}
```

Fonction factorielle: Évaluation

```
fact(4)
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * (if 1 <= 1 {return 1} else {return 1 *
fact(1-1)})))
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * (if 1 <= 1 {return 1} else {return 1 *
fact(1-1)})))
4 * (3 * (2 * 1))
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * (if 1 <= 1 {return 1} else {return 1 *
fact(1-1)})))
4 * (3 * (2 * 1))
4 * (3 * 2)
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * (if 1 <= 1 {return 1} else {return 1 *
fact(1-1)})))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
```

Fonction factorielle: Évaluation

```
fact(4)
if 4 <= 1 {return 1} else {return 4 * fact(4-1)}
4 * fact(3)
4 * (if 3 <= 1 {return 1} else {return 3 * fact(3-1)})
4 * (3 * fact(2))
4 * (3 * (if 2 <= 1 {return 1} else {return 2 * fact(2-1)}))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * (if 1 <= 1 {return 1} else {return 1 *
fact(1-1)})))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Fonction factorielle: Évaluation

```
fact(4)
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
(4 * (3 * (2 * fact(1))))
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
(4 * (3 * (2 * fact(1))))
(4 * (3 * (2 * 1)))
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
(4 * (3 * (2 * fact(1))))
(4 * (3 * (2 * 1)))
(4 * (3 * 2))
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
(4 * (3 * (2 * fact(1))))
(4 * (3 * (2 * 1)))
(4 * (3 * 2))
(4 * 6)
```

Fonction factorielle: Évaluation

```
fact(4)
(4 * fact(3))
(4 * (3 * fact(2)))
(4 * (3 * (2 * fact(1))))
(4 * (3 * (2 * 1)))
(4 * (3 * 2))
(4 * 6)
24
```

Fonction factorielle: Évaluation

```
fact(6)
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
```

Fonction factorielle: Évaluation

```
fact(6)
6 * fact(5)
6 * (5 * fact(4))
6 * (5 * (4 * fact(3)))
6 * (5 * (4 * (3 * fact(2))))
6 * (5 * (4 * (3 * (2 * fact(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

Fonction factorielle: Espace d'exécution

L'espace d'exécution augmente linéairement durant l'exécution

Fonction factorielle: Espace d'exécution

L'espace d'exécution augmente linéairement durant l'exécution

Chaque programme possède un espace d'exécution limité: la pile (*stack*)

Fonction factorielle: Espace d'exécution

L'espace d'exécution augmente linéairement durant l'exécution

Chaque programme possède un espace d'exécution limité: la pile (*stack*)

Les débordement de pile provoque des erreurs (*stack overflow*)

Fonction fact2

Une fonction recursive terminale ne fait aucune opération après l'appel récursif.

```
func fact2( _ n: UInt64, _ m: UInt64 ) -> UInt64 {
  if n <= 1 {
    return m
  } else {
    return fact2( n - 1, m * n )
  }
}
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
fact2( 2, 12 )
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
fact2( 2, 12 )
if 2 <= 1 {return 12} else {return fact2( 2-1, 2 * 12 )}
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
fact2( 2, 12 )
if 2 <= 1 {return 12} else {return fact2( 2-1, 2 * 12 )}
fact2( 1, 24 )
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
fact2( 2, 12 )
if 2 <= 1 {return 12} else {return fact2( 2-1, 2 * 12 )}
fact2( 1, 24 )
if 1 <= 1 {return 24} else {return fact2( 1-1, 1 * 24 )}
```

Fonction fact2: Évaluation

```
fact2( 4, 1 )
if 4 <= 1 {return 1} else {return fact2( 4-1, 4 * 1 )}
fact2( 3, 4 )
if 3 <= 1 {return 4} else {return fact2( 3-1, 3 * 4 )}
fact2( 2, 12 )
if 2 <= 1 {return 12} else {return fact2( 2-1, 2 * 12 )}
fact2( 1, 24 )
if 1 <= 1 {return 24} else {return fact2( 1-1, 1 * 24 )}
24
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
fact2( 4, 30 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
fact2( 4, 30 )
fact2( 3, 120 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
fact2( 4, 30 )
fact2( 3, 120 )
fact2( 2, 360 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
fact2( 4, 30 )
fact2( 3, 120 )
fact2( 2, 360 )
fact2( 1, 720 )
```

Fonction fact2: Évaluation

```
fact2( 6, 1 )
fact2( 5, 6 )
fact2( 4, 30 )
fact2( 3, 120 )
fact2( 2, 360 )
fact2( 1, 720 )
720
```

Espace d'exécution

L'espace d'exécution est constant

Les résultats sont les mêmes !

```
func fact2( _ n: UInt64, _ m: UInt64 ) -> UInt64 {
  if n <= 1 {
    return m
  } else {
    return fact2( n - 1, m * n )
  }
}
```

Paramètre supplémentaire

On peut cacher le paramètre supplémentaire:

```
func fact( _ n: UInt64 ) -> UInt64 {
  func fact2( _ n: UInt64, _ m: UInt64 ) -> UInt64 {
    if n <= 1 {
      return m
    } else {
      return fact2( n - 1, m * n )
    }
  }
  return fact2( n, 1 )
}
```

Récursion terminale

Certains compilateurs et interpréteurs (comme Swift, Scheme, Scala, etc.) peuvent optimiser ces appels (*tail call elimination*)

Somme de tableau

Récursive:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}
```

Somme de tableau

Récursive:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}
```

Récursive terminale:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt, _ partialSum: Int) -> Int {
    if i >= x.count {
      return partialSum
    } else {
      return sumFrom(i + 1, x[i] + partialSum)
    }
  }
  return sumFrom(0, 0)
}
```

Récursion terminale ?

```
func truc( x: [Int] ) -> Boolean {
  func f( _ i: Int ) -> Boolean {
    if i >= x.count  {
      return true
    } else {
      if x[i] < 0 {
        return false
      } else {
        return f( i + 1 )
      }
    }
  }
  return f(0)
}
```

Récursion terminale ?

```
func chose( x: [Int] ) -> UInt {
  func g( _ i: Int ) -> UInt {
    if i >= x.count  {
      return 0
    } else {
      if x[i] < 0 {
        return g( i + 1 )
      } else {
        return g( i + 1 ) + 1
      }
    }
  }
  return g(0)
}
```

Récursion terminale ?

```
func chose( x: [Int] ) -> UInt {
  func g( _ i: Int, _ n: UInt ) -> UInt {
    if i >= x.count  {
      return n
    } else {
      if x[i] < 0 {
        return g( i + 1, n )
      } else {
        return g( i + 1, n + 1 )
      }
    }
  }
  return g(0, 0)
}
```

Nombre de chiffres dans un nombre

Récursive:

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

Nombre de chiffres dans un nombre

Récursive:

```
func digits( _ n: UInt ) -> UInt {
  if n < 10 {
    return 1
  } else {
    return 1 + digits( n / 10 )
  }
}
```

Récursive terminale:

```
func digits( _ n: UInt ) -> UInt {
  func count( _ n: UInt, _ d: UInt ) -> UInt {
    if n < 10  {
      return d
    } else {
      return count( n / 10, d + 1 )
    }
  }
  return count( n, 1 )
}
```

Récursion terminale ?

```
func sqrt( _ x: Double ) -> Double {
  let eps = 1e-9

  func improve( _ G: Double ) -> Double {
    return (G + x/G) / 2
  }

  func attempt( _ G: Double ) -> Double {
    if abs(x - G * G) < eps {
      return G
    } else {
      return attempt( improve(G) )
    }
  }

  return attempt( 1 )
}
```

FIN