[CI2] Cours 7: Récursion

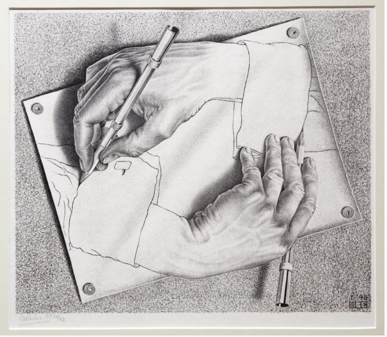
Daniela Petrișan
Université de Paris, IRIF

Récursion

Une définition récursive est une définition dans laquelle on utilise le nom de l'objet défini dans la définition même.

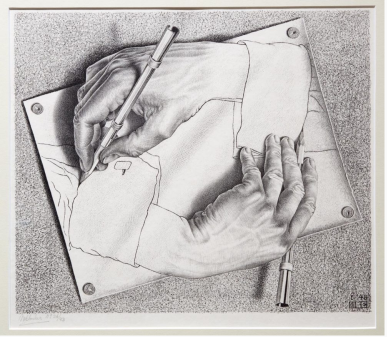
Récursion

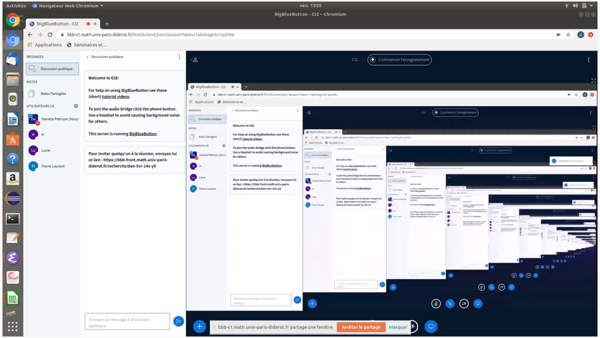
Une définition récursive est une définition dans laquelle on utilise le nom de l'objet défini dans la définition même.



Récursion

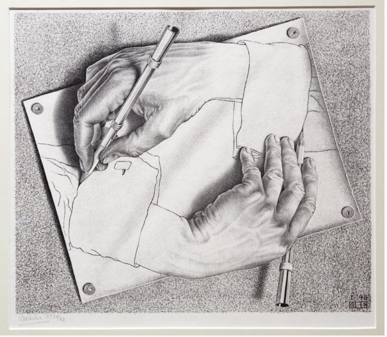
Une définition récursive est une définition dans laquelle on utilise le nom de l'objet défini dans la définition même.

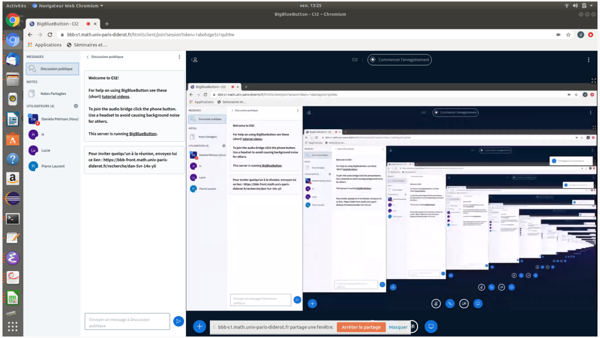




Récursion

Une définition récursive est une définition dans laquelle on utilise le nom de l'objet défini dans la définition même.





GNU – acronyme récursif qui signifie en anglais « GNU's Not UNIX »

see also `https://www.gnu.org/gnu/pronunciation.html`

Fibonacci – implémentation naïve

En informatique, certains algorithmes et certaines structures de données sont intrinsèquement récursifs. Voir par exemple, les listes chaînées que vous avez déjà vues dans IP2.

Fibonacci – implémentation naïve

En informatique, certains algorithmes et certaines structures de données sont intrinsèquement récursifs. Voir par exemple, les listes chaînées que vous avez déjà vues dans IP2.

En mathématiques, on parle de définition par récurrence pour une suite, e.g., la suite de Fibonacci

$$f_0 = 1$$
$$f_1 = 1$$
$$f_{n+2} = f_{n+1} + f_n \quad \forall n \geq 0$$

```java
public static long fiboNaive(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaive(n - 1) + fiboNaive(n - 2);
  }
}
```

Fibonacci – Arbre des appels de l’implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

```
f(5)
.	f(4)
.	.	f(3)
.	.	.	f(2)
.	.	.	.	f(1)
.	.	.	.	f(0)
.	.	.	f(1)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	f(3)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	.	f(1)
```

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

Combien d'appels à la fonction `fiboNaive` sont engendrés par un appel à `fiboNaive(n)` ?

```
f(5)
.	f(4)
.	.	f(3)
.	.	.	f(2)
.	.	.	.	f(1)
.	.	.	.	f(0)
.	.	.	f(1)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	f(3)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	.	f(1)
```

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") + fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

```
f(5)
.       f(4)
.       .       f(3)
.       .       .       f(2)
.       .       .       .       f(1)
.       .       .       .       f(0)
.       .       .       f(1)
.       .       f(2)
.       .       .       f(1)
.       .       .       f(0)
.       f(3)
.       .       f(2)
.       .       .       f(1)
.       .       .       f(0)
.       .       f(1)
```

Combien d'appels à la fonction `fiboNaive` sont engendrés par un appel à `fiboNaive(n)` ? Notons ce nombre par a_n.

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

Combien d'appels à la fonction `fiboNaive` sont engendrés par un appel à `fiboNaive(n)` ? Notons ce nombre par a_n.

$$a_0 = a_1 = 1$$

```
f(5)
.       f(4)
.       .       f(3)
.       .       .       f(2)
.       .       .       .       f(1)
.       .       .       .       f(0)
.       .       .       f(1)
.       .       f(2)
.       .       .       f(1)
.       .       .       f(0)
.       f(3)
.       .       f(2)
.       .       .       f(1)
.       .       .       f(0)
.       .       f(1)
```

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

```
f(5)
.	f(4)
.	.	f(3)
.	.	.	f(2)
.	.	.	.	f(1)
.	.	.	.	f(0)
.	.	.	f(1)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	f(3)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	.	f(1)
```

Combien d'appels à la fonction `fiboNaive` sont engendrés par un appel à `fiboNaive(n)` ? Notons ce nombre par a_n.

$$a_0 = a_1 = 1$$

$$a_{n+2} = 1 + a_{n+1} + a_n \quad \forall n \geq 0$$

Fibonacci – Arbre des appels de l'implémentation naïve

```java
public static long fiboNaiveTracee(int n, String s) {
  System.out.println(s + "f(" + n +")");
  if ( n == 0 || n == 1) {
    return 1;
  } else {
    return fiboNaiveTracee(n - 1, s + ".\t") +
     ↪  fiboNaiveTracee(n - 2, s + ".\t");
  }
}
```

Combien d'appels à la fonction `fiboNaive` sont engendrés par un appel à `fiboNaive(n)` ? Notons ce nombre par a_n. On obtient $a_n = 2f_n - 1$!

$$a_0 = a_1 = 1$$

$$a_{n+2} = 1 + a_{n+1} + a_n \quad \forall n \geq 0$$

```
f(5)
.	f(4)
.	.	f(3)
.	.	.	f(2)
.	.	.	.	f(1)
.	.	.	.	f(0)
.	.	.	f(1)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	f(3)
.	.	f(2)
.	.	.	f(1)
.	.	.	f(0)
.	.	f(1)
```

Mémoïsation : amélioration de la complexité du temps de calcul

L'implémentation naïve du calcul de la suite de Fibonacci est très coûteuse en temps ! On va essayer de calculer f_{200} !

Mémoïsation : amélioration de la complexité du temps de calcul

L'implémentation naïve du calcul de la suite de Fibonacci est très coûteuse en temps ! On va essayer de calculer f_{200} !

Peut-on économiser certains calculs ? Lors du calcul de f_5, nous avons calculé f_3 plusieurs fois, de même pour f_2, etc,... Si nous faisions ces calculs sur papier, nous noterions le résultat et éviterions de refaire le calcul.

Mémoïsation : amélioration de la complexité du temps de calcul

L'implémentation naïve du calcul de la suite de Fibonacci est très coûteuse en temps ! On va essayer de calculer f_{200} !

Peut-on économiser certains calculs ? Lors du calcul de f_5, nous avons calculé f_3 plusieurs fois, de même pour f_2, etc,... Si nous faisions ces calculs sur papier, nous noterions le résultat et éviterions de refaire le calcul.

Nous pouvons utiliser de la mémoire supplémentaire pour faire la même chose. Nous stockerons au fur et à mesure les résultats intermédiaires, et nous les réutiliserons si nécessaire. C'est le mécanisme de mémoïsation.

Mémoïsation : amélioration de la complexité du temps de calcul

L'implémentation naïve du calcul de la suite de Fibonacci est très coûteuse en temps ! On va essayer de calculer f_{200} !

Peut-on économiser certains calculs ? Lors du calcul de f_5, nous avons calculé f_3 plusieurs fois, de même pour f_2, etc,... Si nous faisions ces calculs sur papier, nous noterions le résultat et éviterions de refaire le calcul.

Nous pouvons utiliser de la mémoire supplémentaire pour faire la même chose. Nous stockerons au fur et à mesure les résultats intermédiaires, et nous les réutiliserons si nécessaire. C'est le mécanisme de mémoïsation.

```java
static long[] mem;

public static long fiboMemoisee(int n) {
  mem = new long[n+1];
  return fiboAux(n);
}

public static long fiboAux(int n) {
  if (mem[n] == 0) {
    if (n == 0 || n == 1) {
      mem[n] = 1;
    } else {
      mem[n] = fiboAux(n-1) + fiboAux(n-2);
    }
  }
  return mem[n];
}
```

Arbre d'appel pour le calcul de Fibonacci utilisant la mémoïsation

```java
static long[] mem;

public static long fiboMemoisee(int n) {
  mem = new long[n+1];
  return fiboAux(n);
}

public static long fiboAux(int n) {
  if (mem[n] == 0) {
    if (n == 0 || n == 1) {
      mem[n] = 1;
    } else {
      mem[n] = fiboAux(n-1) + fiboAux(n-2);
    }
  }
  return mem[n];
}
```