Chapitre 4 : Récursivité

27 septembre 2023

Définition

On considère la fonction `fact` suivante, qui est une implémentation de la fonction mathématique factorielle :

```
int fact(int n)
    {
    int r = 1;
    for (int i = 1; i <= n; i++){
        r = r*i;
    }
    return r;
    }
```

Définition

Définition de la factorielle :

$$n! = \prod_{i=1}^{n} i$$

Autre définition possible :

$$\begin{cases} n! = 1 & \text{si } n = 0 \\ n! = n(n-1)! & \text{sinon} \end{cases}$$

Définition

$n!$ est définie en fonction de celle de $(n-1)!$ quand n est différent de 0 :

On a besoin de la fonction factorielle à l'intérieur de sa propre définition.

Définition

Une fonction **récursive** est une fonction qui fait appel à elle-même dans sa propre définition.

Syntaxe en C

En C : possible de définir des fonctions de façon récursive en les appelant à l'intérieur du corps de la fonction

Exemple : Définition récursive de la fonction factorielle :

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

Syntaxe en C

Exercice : programmer en C une version récursive de l'exponentiation naïve.

Définition récursive du calcul de la puissance :

Syntaxe en C

Exercice : programmer en C une version récursive de l'exponentiation naïve.

Définition récursive du calcul de la puissance :

$$\begin{cases} a^n = 1 & \text{si } n = 0 \\ a^n = a \times a^{n-1} & \text{sinon} \end{cases}$$

Syntaxe en C

Solution :

```
int exp_naive(int a, int n){
    if (n == 0){
        return 1;
    }
    else{
        return a*exp_naive(a,n-1);
    }
}
```

Syntaxe en C

Exercice : programmer en C une version récursive de l'exponentiation rapide.

Rappel : $a^n = a^{2q+r} = a^r(a^2)^q$, où $r = 0$ ou 1 est le reste dans la division euclidienne de n par 2, et $q = \lfloor \frac{n}{2} \rfloor$ est le quotient dans la division euclidienne de n par 2

Définition récursive du calcul de la puissance avec l'exponentiation rapide :

Syntaxe en C

Exercice : programmer en C une version récursive de l'exponentiation rapide.

Rappel : $a^n = a^{2q+r} = a^r(a^2)^q$, où $r = 0$ ou 1 est le reste dans la division euclidienne de n par 2, et $q = \lfloor \frac{n}{2} \rfloor$ est le quotient dans la division euclidienne de n par 2

Définition récursive du calcul de la puissance avec l'exponentiation rapide :

$$\begin{cases} a^n = 1 & \text{si } n = 0 \\ a^n = (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{si } n \neq 0 \text{ et } n \text{ est pair} \\ a^n = a \times (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{si } n \neq 0 \text{ et } n \text{ est impair} \end{cases}$$

Syntaxe en C

```
int exp_rapide(int a, int n){
    if (n == 0){
        return 1;
    }
    else{
        int b = exp_rapide(a,n/2);
        if (n%2 == 0){
            return b*b;
        }
        else {
            return a*b*b;
        }
    }
}
```

Arbre d'Appels

Dans les différentes fonctions ci-dessus, la définition est telle que l'on a deux situations possibles lorsqu'on appelle la fonction sur une valeur donnée :

- La valeur d'entrée produit un **appel récursif**
- La valeur d'entrée est telle qu'aucun appel récursif n'est fait dans le calcul : c'est un **cas terminal** (exemple : 0 pour la factorielle et les deux exponentiations)

Arbre d'Appels

Lorsque l'on appelle une fonction récursive sur une valeur qui n'est pas un cas terminal : plusieurs appels récursifs jusqu'à arriver sur les cas terminaux

Exemple : lors de l'appel de `fact(3)`, on a un appel à `fact(2)`, qui lui-même entraîne un appel récursif à `fact(1)`, qui entraîne un appel à `fact(0)`.

`fact(0) = 1` correspond à un cas terminal : `fact(1)` peut alors être calculé en fonction de `fact(0)`, `fact(2)` en fonction de `fact(1)` et `fact(3)` en fonction de `fact(2)`.

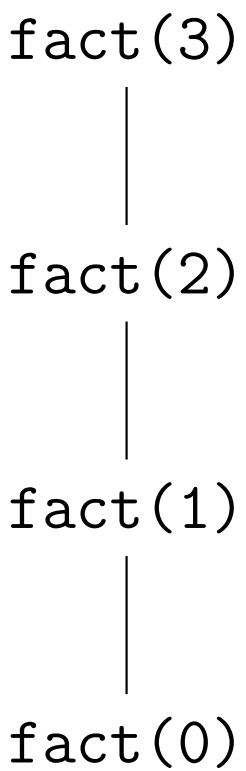
Arbre d'Appels

la liste des appels qui sont produits à partir de l'appel d'une fonction sur une valeur donnée peut être représentée par un arbre : un noeud au sommet de l'arbre correspond à l'appel initial, et chacun des appels récursifs à l'intérieur de l'appel initial va constituer une branche qui donnera elle-même lieu à un sous-arbre.

Une telle structure arborescente est nommée **arbre d'appel**.

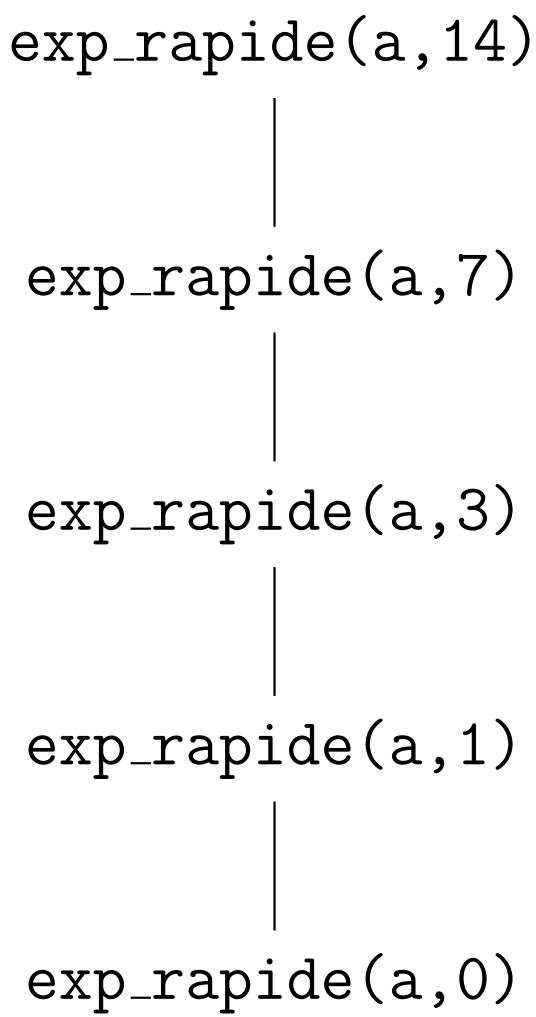
Arbre d'Appels

Exemple : arbre d'appel pour `fact(3)` :



Arbre d'Appels

Exemple : Arbre d'appel pour `exp_rapide(a,14)` :



Arbre d'Appels

Un seul appel récursif dans le corps de la fonction = une unique branche à chaque noeud

Plusieurs appels récursifs = plusieurs branches

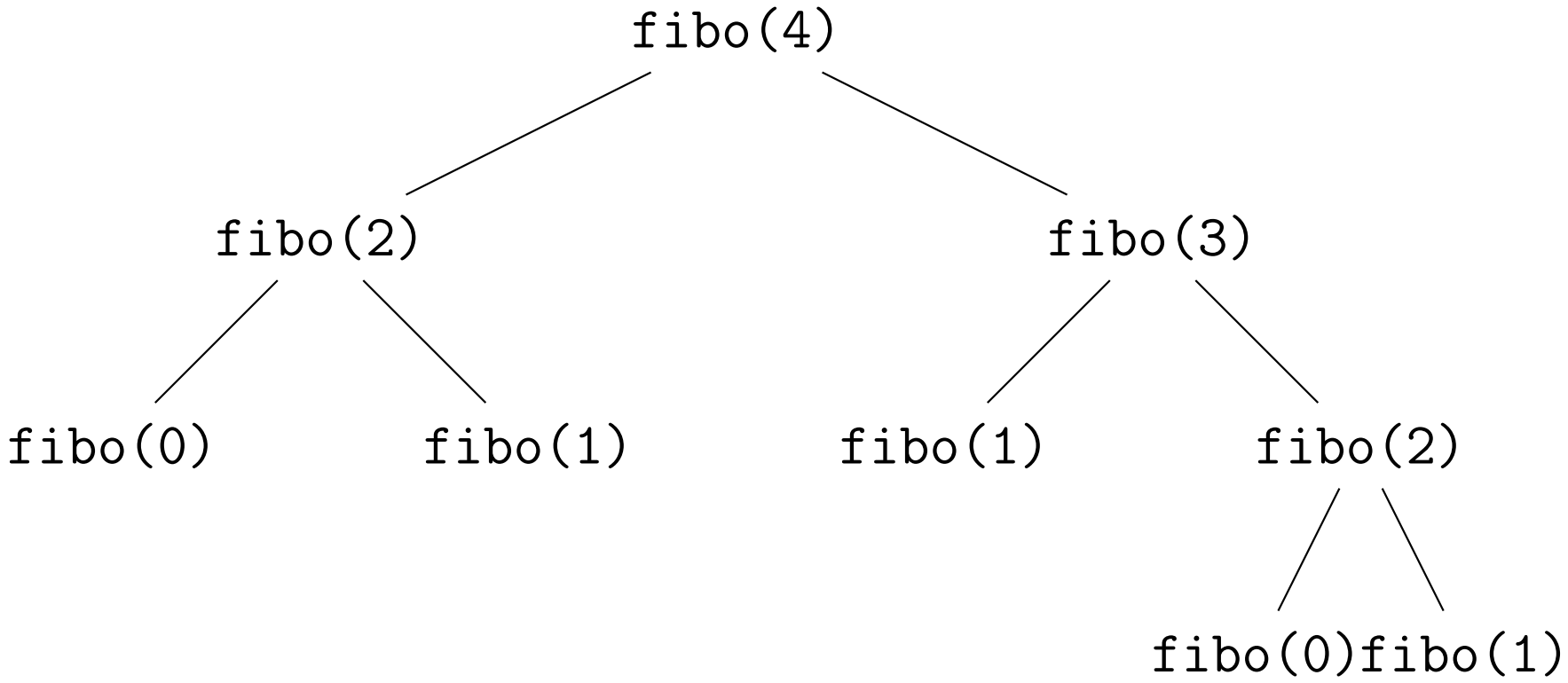
Arbre d'Appels

Exemple : version récursive naïve du calcul du n-ième terme de la suite de Fibonacci :

```
int fibo(int n){
    if (n == 0){
        return 0;
    }
    else if (n==1){
        return 1;
        }
        else {
            return fibo(n-1) + fibo(n-2);
        }
    }
}
```

Arbre d'Appels

Arbre d'appels pour `fibo(4)` :



Arbre d'Appels

Problèmes de cette version :

- Le nombre d'appels de `fibo` croît exponentiellement en fonction de n, et par conséquent la complexité également (complexité de la version itérative : $\Theta(n)$)
- Problème de complexité spatiale : chacun des noeuds de l'arbre existe en mémoire tant que le calcul des valeurs qu'il appelle n'est pas fait

Ce phénomène se produit car des appels récursifs sont effectués plusieurs fois. On verra qu'il existe des façons d'éviter ces appels récursifs superflus (mémoïzation)

Récursif vs Itératif

- Pour toute version récursive d'un algorithme, il existe une version itérative, et vice-versa
- Pas de méthode qui soit clairement meilleure que l'une autre
- Choix de la méthode : peut être influencé par le choix du langage

Récursif vs Itératif

Inconvénient de la récursivité : il peut être dificile de trouver un schéma de définition par récurrence

Avantage : quand on en a un, celui-ci peut permettre de programmer de façon simlpe certains problèmes

Récursif vs Itératif

Exemple : déplacements des Tours de Hanoi

Problème :

- **Entrée :** un entier naturel non-nul n
- **Sortie :** la liste des déplacements de disques à effectuer pour déplacer une tour de Hanoi de taille n de la tige gauche à la tige droite.

Récursif vs Itératif

On peut définir récursivement, de façon très simple, cette liste de mouvement en fonction de n :

- Si $n = 1$: on déplace le seul disque de la tige gauche à la tige droite.
- Si $n > 1$:
  - On déplace la sous-tour de taille $n - 1$ de la gauche au milieu, avec la tige droite comme intermédiaire ;
  - On déplace le disque de base de gauche à droite ;
  - On déplace la sous-tour de taille $n - 1$ du milieu vers la droite, avec la tige gauche comme intermédiaire.