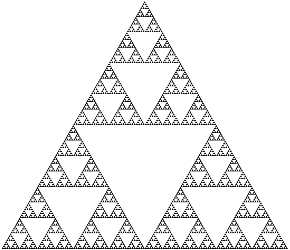
Récursivité

Informatique pour tous



Fonctions récursives

Définition

On dit qu'une fonction est **récursive** si elle s'appelle elle-même.

Fonctions récursives

Définition

On dit qu'une fonction est **récursive** si elle s'appelle elle-même.

On utilise souvent une fonction récursive quand **un problème peut se ramener à l'étude de sous-problèmes**.

Fonctions récursives

Par exemple, pour calculer $n!$, on peut utiliser le fait que :

$$n! = n \times (n-1)!, \text{ si } n > 0$$

$$0! = 1$$

Fonctions récursives

Par exemple, pour calculer $n!$, on peut utiliser le fait que :

$$n! = n \times (n-1)!, \text{ si } n > 0$$

$$0! = 1$$

Le calcul de $n!$ se ramène à celui de $(n-1)!$

Fonction récursive factorielle

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

Fonction récursive factorielle

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

À comparer avec la version itérative :

```
def fact(n):
    res = 1
    for i in range(1, n+1):
        res *= i
    return res
```

Fonction factorielle

⚠ On peut facilement faire des fonctions qui ne terminent jamais si on oublie le cas de base...

```
def fact(n):
    return n * fact(n-1)
```

Fonction factorielle

⚠ On peut facilement faire des fonctions qui ne terminent jamais si on oublie le cas de base...

```
def fact(n):
    return n * fact(n-1)
```

```
RecursionError: maximum recursion depth exceeded
```

Python limite le nombre d'appels récursifs à 1000.

Fonction récursive

En général, une fonction récursive possède :

1. Un/des cas de base, où la fonction retourne directement une valeur.
2. Sinon, un/des appels récursifs sur un sous-problème «plus petit».

```
def f(...):
    if ...: # cas de base
        return ...
    return ... f(...) ... # appel(s) récursif(s)
```

Fonction récursive

En général, une fonction récursive possède :

1. Un/des cas de base, où la fonction retourne directement une valeur.
2. Sinon, un/des appels récursifs sur un sous-problème «plus petit».

```
def f(...):
    if ...: # cas de base
        return ...
    return ... f(...) ... # appel(s) récursif(s)
```

Autres exemples : calcul de somme, de suite récurrente...

Autres exemples

```python
def f(n):
    if n != 0:
        print(n)
        f(n - 1)
```

Qu'affiche `f(5)` ?

Autres exemples

```python
def f(n):
    if n != 0:
        print(n)
        f(n - 1)
```

Qu'affiche `f(5)` ? 5 4 3 2 1

```python
def f(n):
    if n != 0:
        f(n - 1)
        print(n)
```

Qu'affiche `f(5)` ?

Autres exemples

```
def f(n):
    if n != 0:
        print(n)
        f(n - 1)
```

Qu'affiche `f(5)` ? 5 4 3 2 1

```
def f(n):
    if n != 0:
        f(n - 1)
        print(n)
```

Qu'affiche `f(5)` ? 1 2 3 4 5

Terminaison

Question

Comment prouver qu'une fonction récursive **termine** ?

Terminaison

Question

Comment prouver qu'une fonction récursive **termine** ?

Généralement, on trouve un argument qui diminue à chaque fois qu'un appel est effectué.

Terminaison

Si $n \in \mathbb{N}^*$, `fact(n)` termine car l'argument diminue de 1 à chaque appel récursif jusqu'à valoir 0.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

Terminaison

Si $n \in \mathbb{N}^*$, `fact(n)` termine car l'argument diminue de 1 à chaque appel récursif jusqu'à valoir 0.

```python
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

… par contre `fact(n)` ne termine pas si $n < 0$!

Terminaison

Question

Est-ce que la fonction `f` suivante termine ? Pour quelles valeurs de a et b ?

```python
def f(a, b):
    if a <= 0 or b <= 0:
        return 1
    if a % 2 == 0:
        return f(b + 1, a - 3)
    else:
        return f(b - 2, a + 1)
```

Terminaison

```python
def f(a, b):
    if a <= 0 or b <= 0:
        return 1
    if a % 2 == 0:
        return f(b + 1, a - 3)
    else:
        return f(b - 2, a + 1)
```

`f(a, b)` termine si $a \geq 0$ et $b \geq 0$ car la **somme** des deux arguments diminue strictement à chaque appel récursif, donc forcément au bout d'un moment l'un des deux sera ≤ 0.

Correction

Question

Comment prouver qu'une fonction récursive est **correct** ?

Correction

Question

Comment prouver qu'une fonction récursive est **correct** ?

Par récurrence !

Correction de fact

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

On peut montrer :

$$H_n : \texttt{fact}(n) \text{ termine et renvoie la valeur } n!$$

Correction de fact

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

- H_0 est vraie car `fact`(0) renvoie 1 et $0\,! = 1$.

Correction de fact

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

- H_0 est vraie car `fact`(0) renvoie 1 et $0\,! = 1$.
- Si H_{n-1} est vraie alors l'appel `fact`$(n-1)$ renvoie bien $(n-1)!$, par hypothèse de récurrence.
  Donc `fact`(n) renvoie bien $n\times$ `fact`$(n-1) = n!$

Complexité

Question

Comment trouver la complexité (nombre d'opérations) d'une fonction récursive ?

Complexité

Question

Comment trouver la complexité (nombre d'opérations) d'une fonction récursive ?

Souvent, cela revient à résoudre une équation de récurrence.

Complexité

Soit $C(n)$ le nombre d'opérations réalisées par `fact`(n). Il faut trouver une formule de récurrence pour $C(n)$.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

Complexité

Soit $C(n)$ le nombre d'opérations réalisées par `fact`(n). Il faut trouver une formule de récurrence pour $C(n)$.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

$$C(n) = \begin{cases} 1, & \text{si n} = 0. \\ K + C(n-1), & \text{si } n > 0. \end{cases}$$

(K est une constante)

Complexité

$$C(n) = \begin{cases} 1, & \text{si n} = 0. \\ K + C(n-1), & \text{si } n > 0. \end{cases}$$

On a vu comment résoudre ce type de suite en mathématiques :

$$C(n) = K + C(n-1) = K + K + C(n-2) = ... = \underbrace{K + K + ... + K}_{n} + C(0)$$

Complexité

$$C(n) = \begin{cases} 1, & \text{si n} = 0. \\ K + C(n-1), & \text{si } n > 0. \end{cases}$$

On a vu comment résoudre ce type de suite en mathématiques :

$$C(n) = K + C(n-1) = K + K + C(n-2) = ... = \underbrace{K + K + ... + K}_{n} + C(0)$$

$$C(n) = Kn + 1 = O(n)$$

`fact` a donc une complexité **linéaire**, comme pour la version itérative.

Exercice : calcul de suite récurrente

Soit u_n la suite (récurrente) définie par :

$$u_0 = 11$$

$$u_n = 7u_{n-1}^2 - 2(n-1)$$

Exercice : calcul de suite récurrente

Soit u_n la suite (récurrente) définie par :

$$u_0 = 11$$

$$u_n = 7u_{n-1}^2 - 2(n-1)$$

On peut facilement écrire une fonction récursive qui calcule u_n :

Exercice : calcul de suite récurrente

Soit u_n la suite (récurrente) définie par :

$$u_0 = 11$$

$$u_n = 7u_{n-1}^2 - 2(n-1)$$

On peut facilement écrire une fonction récursive qui calcule u_n :

```
def u(n):
    if n == 0:
        return 11
    return 7*u(n-1)**2 - 2*(n-1)
```

Suite de Fibonacci

Question

Écrire un algorithme récursif pour calculer le $n$ième terme de la suite de Fibonacci F_n.

$$F_n = \begin{cases} 0, & \text{si n} = 0. \\ 1, & \text{si n} = 1. \\ F_{n-1} + F_{n-2}, & \text{sinon.} \end{cases}$$

Fibonacci

Solution récursive « naïve » :

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Question

Quelle est sa complexité $C(n)$?

Fibonacci

Solution récursive « naïve » :

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Question

Quelle est sa complexité $C(n)$?

$$C(n) = k + C(n-1) + C(n-2), \quad \text{si } n \geq 2.$$

En particulier $C(n) \geq F_n$.

Fibonacci

Solution récursive « naïve » :

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Question

Quelle est sa complexité $C(n)$?

$$C(n) = k + C(n-1) + C(n-2), \quad \text{si } n \geq 2.$$

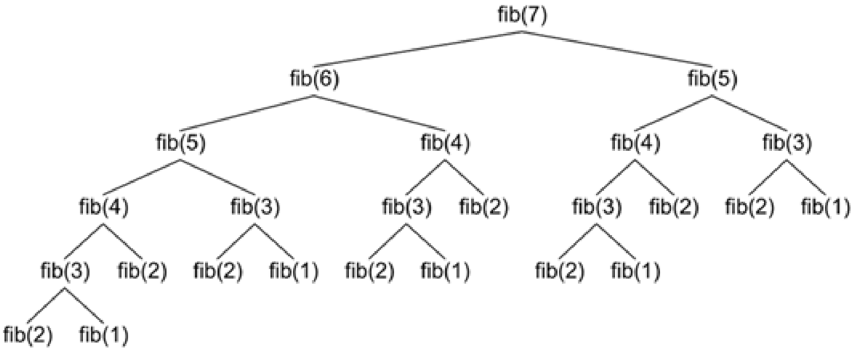
En particulier $C(n) \geq F_n$.

$$F_n \sim \frac{\phi^n}{\sqrt{5}} \gg \frac{1.6^n}{\sqrt{5}}$$

Ce qui est une très mauvaise complexité (exponentielle) !

Fibonacci

`fib(7)` effectue plusieurs fois les mêmes calculs, d'où son inefficacité :



Fibonacci

Pour calculer un terme de la suite de Fibonacci, il est préférable d'utiliser un algorithme itératif (sans récursivité) :

Fibonacci

Pour calculer un terme de la suite de Fibonacci, il est préférable d'utiliser un algorithme itératif (sans récursivité) :

```
def fib_iter(n):
    f0, f1 = 0, 1
    for i in range(n):
        f0, f1 = f1, f0 + f1
    return f0
```

Sa complexité est :

Fibonacci

Pour calculer un terme de la suite de Fibonacci, il est préférable d'utiliser un algorithme itératif (sans récursivité) :

```python
def fib_iter(n):
    f0, f1 = 0, 1
    for i in range(n):
        f0, f1 = f1, f0 + f1
    return f0
```

Sa complexité est : $O(n)$

Puissance rapide

Comment ** est implémenté en Python ? Quelle est la meilleure méthode pour calcule x^n ?

Puissance rapide

Comment `**` est implémenté en Python ? Quelle est la meilleure méthode pour calcule x^n ?

```
def puiss_naive(x, n):
    if n == 0:
        return 1
    return x * puiss_naive(x, n - 1)
```

Question

Quel est le nombre de multiplications $C(n)$ effectuées par `puiss_naive(x, n)` ?

Puissance rapide

Comment ** est implémenté en Python ? Quelle est la meilleure méthode pour calcule x^n ?

```
def puiss_naive(x, n):
    if n == 0:
        return 1
    return x * puiss_naive(x, n - 1)
```

Question

Quel est le nombre de multiplications $C(n)$ effectuées par `puiss_naive(x, n)` ?

$C(n) = 1 + C(n-1) = ... = n - 1.$

On peut faire mieux.

Puissance rapide

Question

En utilisant les égalités suivantes, écrire une fonction récursive pour calculer x^n :

$$\begin{cases} x^n = (x^{\frac{n}{2}})^2 & \text{si } n \text{ est pair} \\ x^n = x \times (x^{\frac{n-1}{2}})^2 & \text{sinon} \end{cases}$$

Dans les deux cas, le calcul de x^n se ramène à celui de $x^{n//2}$.

Puissance rapide

```python
def puiss_rapide(x, n):
    if n == 0:
        return 1
    res = puiss_rapide(x, n // 2)
    if n % 2 == 0:
        return res * res
    else:
        return res * res * x
```

Puissance rapide

```python
def puiss_rapide(x, n):
    if n == 0:
        return 1
    res = puiss_rapide(x, n // 2)
    if n % 2 == 0:
        return res * res
    else:
        return res * res * x
```

Question

Quelle est le nombre de multiplications $C(n)$ de `puiss_rapide(x, n)` ?

Puissance rapide

```python
def puiss_rapide(x, n):
    if n == 0:
        return 1
    res = puiss_rapide(x, n // 2)
    if n % 2 == 0:
        return res * res
    else:
        return res * res * x
```

$$C(n) \leq 2 + C(n//2) \leq 2 + 2 + C(n//4) \leq \ldots \leq \underbrace{2 + 2 + \ldots + 2}_{k} + C(n//2^k)$$

Puissance rapide

```python
def puiss_rapide(x, n):
    if n == 0:
        return 1
    res = puiss_rapide(x, n // 2)
    if n % 2 == 0:
        return res * res
    else:
        return res * res * x
```

$$C(n) \leq 2+C(n//2) \leq 2+2+C(n//4) \leq ... \leq \underbrace{2+2+...+2}_{k}+C(n//2^k)$$

Quand $k \geq \log_2(n)$, $n//2^k = 0$ donc :

$$C(n) \leq \log_2(n) \times 2 + C(0) = \log_2(n) \times 2 + 1$$

D'où $\boxed{C(n) = O(\log(n))}$.

Puissance rapide

`puiss_rapide(x, n)` demande donc $O(\log(n))$ multiplications au lieu des $n-1$ de `puiss_naive(x, n)`.

L'opérateur `**` de Python utilise cet algorithme de puissance rapide.

Tours de Hanoï

Règles du jeu :

1. On ne peut déplacer qu'un disque à la fois (celui du dessus).
2. Il est interdit de poser un disque sur un disque de taille inférieure.

Tours de Hanoï

<u>But du jeu</u> : déplacer les n disques de gauche sur la tige de droite.

Tours de Hanoï

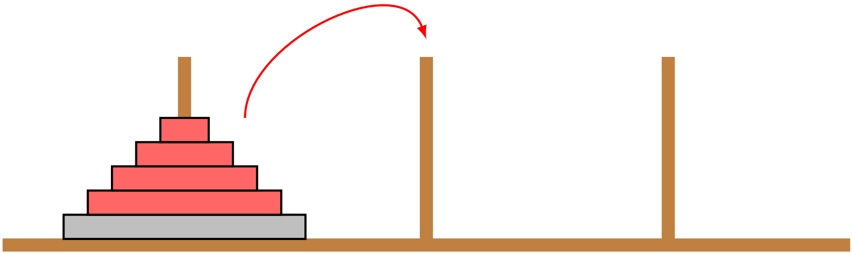
Montrer que c'est possible et écrire un algorithme qui affiche les déplacements à faire pour résoudre ce problème.

Tours de Hanoï

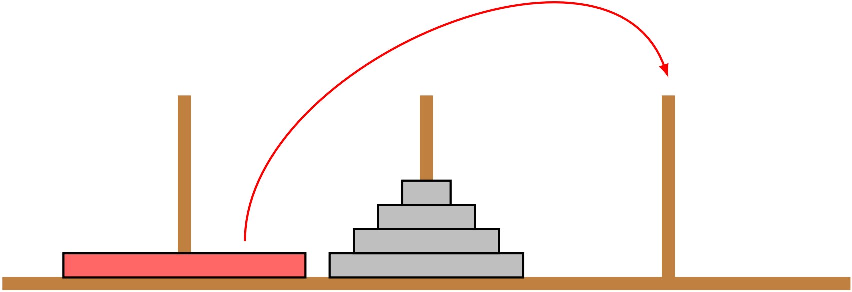
Montrer que c'est possible et écrire un algorithme qui affiche les déplacements à faire pour résoudre ce problème.
Indice : supposons que l'on sache résoudre le problème pour $n-1$ disques. Comment le faire pour n disques ?

Tours de Hanoï

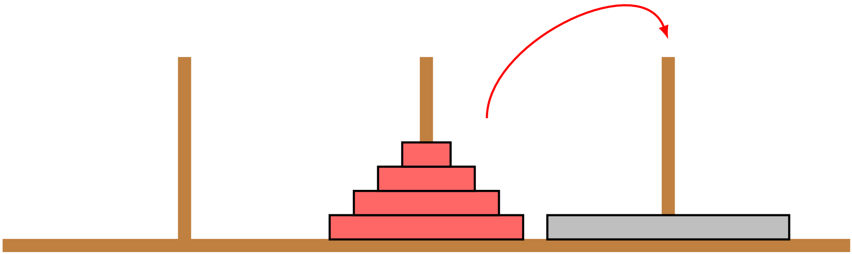
Tours de Hanoï



Tours de Hanoï



Tours de Hanoï



Tours de Hanoï

Tours de Hanoï

```
def hanoi(n, debut, milieu, fin):
    if n != 0:
        hanoi(n - 1, debut, fin, milieu)
        print("deplacer de la tige", debut,
"vers la tige", fin)
        hanoi(n - 1, milieu, fin, debut)
```

Tours de Hanoï

```
def hanoi(n, debut, milieu, fin):
    if n != 0:
        hanoi(n - 1, debut, fin, milieu)
        print("deplacer de la tige", debut,
"vers la tige", fin)
        hanoi(n - 1, milieu, fin, debut)
```

Quel est le nombre de déplacements `D(n)` réalisé par `hanoi` ?

Tours de Hanoï

```
def hanoi(n, debut, milieu, fin):
    if n != 0:
        hanoi(n - 1, debut, fin, milieu)
        print("deplacer de la tige", debut,
"vers la tige", fin)
        hanoi(n - 1, milieu, fin, debut)
```

Quel est le nombre de déplacements `D(n)` réalisé par `hanoi` ?

$$D(n) = 1 + 2D(n-1) = 1 + 2 + 4D(n-2)$$

$$D(n) = 1 + 2 + 2^2 + \ldots + 2^{n-1} + D(0) = 2^n - 1$$

Palindrome

Définition

Un palindrome est un mot qui est le même si on lit le mot de droite à gauche.

Exemple : ABBA, bob, eluparcettecrapule...

Palindrome

Question

Écrire une fonction récursive ayant une chaine de caractères en argument et qui renvoie `True` si cette chaine est un palindrome, `False` sinon.

Palindrome

Question

Écrire une fonction récursive ayant une chaine de caractères en argument et qui renvoie `True` si cette chaine est un palindrome, `False` sinon.

Il faut se poser la question : comment réduire ce problème à un sous-problème ?

Palindrome

Un mot est un palindrome si et seulement si les deux extrémités sont égales et ce qu'il y a entre est aussi un palindrome :

$$a\,\underbrace{bcpndabbadcb}_{\text{palindrome}}\,a$$

Cas de base :

Palindrome

Un mot est un palindrome si et seulement si les deux extrémités sont égales et ce qu'il y a entre est aussi un palindrome :

$$a\,\underbrace{bcpndabbadcb}_{\text{palindrome}}\,a$$

Cas de base : chaîne de caractère de taille ≤ 1.

Palindrome

```
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Pourquoi `palindrome` termine ?

Palindrome

```
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Pourquoi `palindrome` termine ?

Réponse : la taille de la chaine `s` en argument diminue à chaque appel récursif.

Palindrome

```
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Quelle hypothèse de récurrence utiliser pour montrer que `palindrome` est correcte ?

Palindrome

```python
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Quelle hypothèse de récurrence utiliser pour montrer que `palindrome` est correcte ?

H_n : Si s est une chaine de caractères de taille n, `palindrome(s)` renvoie `True` ssi s est un palindrome.

Palindrome

```
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Quel est le nombre d'opérations réalisées par `palindrome(s)` ?

Palindrome

```python
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])
```

Question

Quel est le nombre d'opérations réalisées par `palindrome(s)` ?

La complexité ne dépend que de la taille n de s. On la note $C(n)$.

$$C(n) = k + C(n-2) = k + k + \ldots + k + C(0) = k \times \lfloor \frac{n}{2} \rfloor + C(0) = O(n)$$