TP 11 : Fonctions récursives - I

1 Notion de récursivité

• Une fonction en `python` peut appeler toute fonction définie.

• En particulier une fonction (en `python` ou tout autre langage de programmation) peut s'appeler elle-même. **On parle alors de fonction récursive :** une fonction est récursive lorsqu'elle s'appelle elle même.

• Notamment lorsque le calcul de `fonction(n)` appelle le calcul de `fonction(n-1)`, etc... jusqu'à résoudre celui de `fonction(0)`.

• Un exemple classique est le calcul de factorielle n : $n!$.

```
def fact(n):
    if (n == 0):                # Initialisation
        return 1                # 0! = 1
    else:
        return n * fact(n-1)    # Relation de récurrence
```

• Par exemple l'appel de `fact(3)` appelle `fact(2)` qui appelle `fact(1)` qui finalement appelle `fact(0)` qui retourne 1.

Ainsi `fact(1)` retourne `1*fact(0)`=1, `fact(2)` retourne `2*fact(1)`=2, et finalement `fact(3)` retourne `3*fact(2)`=6.

2 Récursivité : Pile d'exécution

• Les appels successifs sont stockés dans une pile d'exécution.

PILE
`fact(0) = 1`
`fact(1) = 1*fact(0)`
`fact(2) = 2*fact(1)`
`fact(3) = 3*fact(2)`

– Appel de `fact(3)` : Retourne `3*fact(2)`

– Appel de `fact(2)` : Retourne `2*fact(1)`

– Appel de `fact(1)` : Retourne `1*fact(0)`

– Appel de `fact(0)` : Retourne 1

• Les appels successifs sont stockés dans une pile d'exécution.

PILE
`fact(0) = 1`
`fact(1) = 1*fact(0) = 1 * 1 = 1`
`fact(2) = 2*fact(1)` $= 3 * 2 = 6$
`fact(3) = 3*fact(2) = 3 * 2 = 6`

– Appel de `fact(0)` : Retourne 1

– Appel de `fact(1)` : Retourne `1*fact(0)` = 1

– Appel de `fact(2)` : Retourne `2*fact(1) = 2`

– Appel de `fact(3)` : Retourne `3*fact(2) = 6`. C'est le résultat retourné.

• La complexité (ici !) est linéaire en temps et en espace.

• Avantages : élégant, concis, se prête bien à la récurrence, permet de résoudre facilement des problèmes compliqués.

• Inconvénients : complexité en espace (du fait de la pile d'exécution). En `python` le nombre d'appels récursifs est limité à 1000 : pile de capacité limitée.

• Autre désavantage : la récursivité peut facilement donner lieu à des boucles infinis.

Par exemple dans le programme précédent l'appel de `fact(-1)` appelle `fact(-2)`, qui appelle `fact(-3)`, etc..., donnant lieu à une boucle infinie, et dans la pratique à une erreur de dépassement de capacité de la pile d'exécution.

• Amélioration du programme pour éviter ce phénomène : ou encore,

```
def fact(n):
    assert isinstance(n,int) and n >= 0
    if (n == 0):                # Initialisation
        return 1                # 0! = 1
    else:
        return n * fact(n-1)    # Relation de récurrence
```

• Terminaison et correction du programme :

1. si l'argument `n` n'est pas un entier naturel : terminaison avec erreur.

2. si `n` est un entier naturel : terminaison et correction de `fact(n)` se démontrent par un récurrence immédiate sur n : vrai si $n = 0$, et si `fact(n-1)` se termine et retourne $(n-1)!$, alors `fact(n)` se termine en retournant $n.(n-1)! = n!$.

3 Exemples

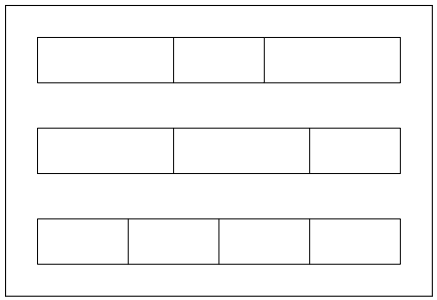
3.1 Rangées de briques

• Etudions un autre exemple où le principe de récurrence est plutôt analogue au principe de récurrence forte.

• Soit $N \geqslant 2$ un entier naturel. Combien y a t-il de façon de constituer une rangée

de briques de longueur N en utilisant seulement des briques de longueur 2 et 3 ?

• Exemple : Voilà 3 façons de faire une rangée de longueur 8 (il y en a d'autres).



• Pour le résoudre facilement : on va procéder par récursivité en se ramenant :

1. Au cas $N-2$: lorsqu'on ajoute à la fin de la rangée une brique de longueur 2.
2. Au cas $N-3$: lorsqu'on ajoute à la fin de la rangée une brique de longueur 3.

L'ensemble {rangées de briques} est partitionné en :

{rangées de briques finissant par longueur 2}
{rangées de briques finissant par longueur 3}.

• On a la relation de récurrence :

Nombre de rangées(N) = Nombre de rangées(N-2) + Nombre de rangées(N-3)

• On en déduit la résolution par la fonction récursive :

```
def briques(N):
    if N < 2:
        return 0
    elif (N == 2) or (N == 3):
        return 1
    else:
        return briques(N-2) + briques(N-3)
```

3.2 L'algorithme d'exponentiation rapide

• Regardons un autre exemple, lui très utile en pratique :

• Calcul d'une puissance x^n par multiplications successives

Supposons que l'on s'interdise l'emploi de l'opération de mise en puissance (comme c'est le cas pour un microprocesseur), et que l'on veuille écrire une fonction prenant en paramètre un nombre x et un entier n et qui retourne la puissance x^n.

• Solution naïve non récursive, à l'aide d'une boucle `for` :

```
def puissance(x,n):
    result = 1
    for i in range(n):
        result = result * x
    return result
```

Complexité : linéaire (en $O(n)$) en temps dans le pire et le meilleur des cas. Bornée (en $O(1)$) en espace.

• Regardons un autre exemple, lui très utile en pratique :

• Calcul d'une puissance x^n par multiplications successives

Supposons que l'on s'interdise l'emploi de l'opération de mise en puissance (comme c'est le cas pour un microprocesseur), et que l'on veuille écrire une fonction prenant en paramètre un nombre x et un entier n et qui retourne la puissance x^n.

• Solution naïve, récursive :

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

Complexité : linéaire (en $O(n)$) en temps dans le pire et le meilleur des cas. Linéaire (en $O(n)$) en espace.

• On peut aller beaucoup plus vite, en mettant à profit les propriétés de la mise en puissance :

$$\boxed{x^0 = 1 \quad ; \quad x^{2p} = (x^2)^p \quad ; \quad x^{2p+1} = x.x^{2p}}$$

• Ainsi :

$$puissance(x,n) = \begin{cases} 1 & \text{si n=0} \\ puissance(x^2, n/2) & \text{si } n \text{ est pair} \\ x \times puissance(x^2, (n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

• Code `python` :

```
def puissance(x,n):
    if n == 0:
        return 1
    elif n%2==0:
        return puissance(x**2,n//2)
    else:
        return x * puissance(x**2,n//2) # ici : n//2 = (n-1)/2
```

Vérifions sur un exemple que l'algorithme est beaucoup plus rapide :

Exemple calcul de x^{100} :

• 100 est pair. On obtient x^{100} en élevant x au carré, $x_1 = x^2$ (1ère mult.) ; on devra élever x_1 à l'exposant 50.

• 50 est pair. On obtient $(x_1)^{50}$ en élevant x_1 au carré, $x_2 = (x_1)^2$ (2ème mult.) ; on devra élever x_2 à l'exposant 25.

• 25 est impair. On obtient $(x_2)^{25}$ en élevant x_2 au carré, $x_3 = (x_2)^2$ (3ème mult.) ; on devra élever x_3 à l'exposant 12 puis multiplier par x_2 (4ème mult.).

• 12 est pair. On obtient $(x_3)^{12}$ en élevant x_3 au carré, $x_4 = (x_3)^2$ (5ème mult.) ; on devra élever x_4 à l'exposant 6.

• 6 est pair. On obtient $(x_4)^6$ en élevant x_4 au carré, $x_5 = (x_4)^2$ (6ème mult.) ; on devra élever x_5 à l'exposant 3.

• 3 est impair. On obtient $(x_5)^3$ en élevant x_5 au carré, (7ème mult.) puis en multipliant par x_5 (8ème mult.).

Au total : 8 multiplications, contre 100 pour l'algorithme naïf !

• Une variante de l'algorithme d'exponentiation rapide, utilisant plutôt :

$$\boxed{x^0 = 1 \quad ; \quad x^{2p} = (x^p)^2 \quad ; \quad x^{2p+1} = x.x^{2p}}$$

• Ainsi :

$$puissance(x,n) = \begin{cases} 1 & \text{si n=0} \\ (puissance(x,n/2))^2 & \text{si } n \text{ est pair} \\ x \times (puissance(x,(n-1)/2))^2 & \text{si } n \text{ est impair} \end{cases}$$

• Code `python` :

```
def puissance(x,n):
    if n == 0:
        return 1
    elif n%2==0:
        return puissance(x,n//2) ** 2
    else:
        return x * puissance(x,n//2) ** 2 # ici : n//2 = (n-1)/2
```

Exemple calcul de x^{100} :

• 100 est pair. On obtient x^{100} en élevant au carré x^{50}, soit une multiplication.

• 50 est pair. On obtient x^{50} en élevant au carré x^{25}, soit une multiplication.

• 25 est impair. On obtient x^{25} en élevant au carré x^{12} puis en multipliant par x, soit 2 multiplications.

• 12 est pair. On obtient x^{12} en élevant au carré x^6, soit une multiplication.

• 6 est pair. On obtient x^6 en élevant au carré x^3, soit une multiplication.

• 3 est impair. On obtient x^3 en élevant x au carré puis en multipliant par x.

Au total : 8 multiplications, contre 100 pour l'algorithme naïf !

3.3 Complexité de l'algorithme d'exponentiation rapide

• On démontre que l'algorithme d'exponentiation rapide a une complexité logarithmique (d'ordre $\Theta(\log(n))$).

• Exercice* :

1. Soit $n \in \mathbb{N}$ et $p = \lfloor \log_2(n) \rfloor$ (où $\lfloor x \rfloor$ désigne la partie entière de x et $\log_2$ le logarithme base 2, i.e. $y = \log_2(x) \iff 2^y = x$.)
   Justifier que :
   $$2^p \leqslant n < 2^{p+1}$$
2. Montrons par récurrence sur p que lorsque $2^p \leqslant n < 2^{p+1}$, alors il faut entre p et $2.(p+1)$ multiplications pour le calcul de de x^n par l'algorithme d'exponentiation rapide.
3. En déduire que l'algorithme d'exponentiation rapide a pour complexité $\Theta(\log(n))$.

• Montrons que sa complexité est logarithmique :

1. Soit $n \in \mathbb{N}$ et $p = \lfloor \log_2(n) \rfloor$.
   Puisque : $2^{\log_2(n)} = n$, $\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$, et $x \mapsto 2^x$ est strictement croissante, alors : $\boxed{2^p \leqslant n < 2^{p+1}}$.

2. Montrons par récurrence sur p qu'il faut entre p et $2.(p+1)$ multiplications pour le calcul de de x^n par l'algorithme d'exponentiation rapide :
   Initialisation. Si $p = 0$: $1 \leqslant n < 2$, et le calcul de $x^n = x$ par l'algorithme d'exponentiation rapide nécessite une multiplication. Vrai au rang 0.
   Hérédité. Supposons la propriété vraie au rang p et supposons que $2^{p+1} \leqslant n < 2^{p+2}$. L'algorithme obtient x^n en : élevant x au carré x^2 (1 multiplication), puis en élevant x^2 à la puissance $\lfloor n/2 \rfloor$, puis éventuellement en multipliant par x ($\leqslant 1$ multiplication).
   Or $2^p \leqslant \lfloor n/2 \rfloor < 2^{p+1}$. Par hypothèse de récurrence l'élévation à la puissance $\lfloor n/2 \rfloor$ par exponentiation rapide nécessite entre p et $2(p+1)$ multiplications, donc celui de x^n nécessite entre $p+1$ et $2(p+1)+2 = 2(p+2)$ multiplications. cqfd.

3. Le nombre d'opérations effectuées par l'algorithme a même ordre que le nombre de multiplications effectuées (au plus 2 tests par multiplication).

D'après la question précédente, le calcul de x^n nécessite :

<u>au moins :</u> $p = (\lfloor \log_2(n) \rfloor \geqslant \log_2(n) - 1 = \dfrac{\log(n)}{\log(2)} - 1$ multiplications,

<u>au plus :</u> $2(p+1) = 2(\lfloor \log_2(n) \rfloor + 1) \leqslant 2\log_2(n) + 2 = 2\dfrac{\log(n)}{\log(2)} + 2$ multiplications.

Donc la la complexité est en $\Theta(\log(n))$.

Exemple : l'exponentiation rapide de x^{2^p} :

$$x^{2^p} = \underbrace{(\cdots((x^2)^2)^2\cdots)^2}_{p \text{ élévations au carré}}$$

nécessite p multiplications (une pour chaque élévation au carré).

4 Exercices

Exercice 1. On rappelle l'algorithme d'Euclide permettant de retourner le *pgcd* de 2 entiers a et b :

```
Tant que b ≠ 0
    r = reste de la division de a par b
    a = b
    b = r
retourner a
```

Ecrire une version récursive de l'algorithme d'Euclide.

Exercice 2. Le format de papier A0 correspond à un rectangle de largeur de 84,1cm et une longueur de 118,9cm. le format A1 est obtenu en coupant en deux parties égales le format A0, il a donc pour longueur la largeur de A0 et pour largeur la moitié de la longueur de A0. Sur le même principe une feuille A1 contient deux feuilles A2, une feuille A2 deux feuilles A3, etc...
Ecrire une fonction récursive prenant en paramètre un entier naturel n et qui retourne longueur et largeur, dans cet ordre, d'une feuille de format An.

Exercice 3. Soit f une application continue sur une intervalle $[a, b]$ vérifiant $f(a).f(b) \leqslant 0$. D'après le théorème des valeurs intermédiaires f admet une racine sur $[a, b]$. On peut déterminer une valeur approchée de cette racine, par une recherche par dichotomie.

Ecrire une version récursive de la recherche d'une racine par dichotomie. L'appel de `dichotomie(f,a,b,e)` retournera une valeur approchée à e près d'une racine de f sur $[a, b]$.

Exercice 4. Ecrire une fonction récursive retournant le nombre de façons différentes de constituer une rangée de longueur n de briques uniquement avec des briques de longueur 1 et 2 ?