Informatique PCSI
Prérequis TP 4 : algorithmes dichotomiques

Exponentiation rapide

Il s'agit d'écrire une fonction `puissance` qui prend en paramètres un flottant x non nul et un entier naturel n et qui renvoie x^n.

Dans une version naïve, on calcule successivement x^2, x^3, etc, en se basant sur la définition mathématique : pour $n \geq 0$, $x^n = 1 \times x \times \ldots \times x$ où n est le nombre de facteur égaux à x.

```
def puissance1(x, n):
    p = 1
    for i in range(n):
        p = p * x
    return p
```

La fonction `puissance1` effectue n produits et $n+1$ affectations. La complexité est donc linéaire en fonction de n.

Remarque

Dans le calcul du coût, un autre facteur ne doit pas être négligé, la taille de `x`. En Python, il n'y a pas de limite de taille pour les entiers. Donc si `x` est de type int, le temps de calcul peut être nettement allongé selon la taille de `x`.

Méthode dichotomique

Pour tout n, si n est pair, $x^n = (x^2)^{n//2}$, sinon $x^n = x(x^2)^{n//2}$, où $n//2$ désigne le quotient entier de n par 2.

Nous en déduisons l'algorithme suivant :

- p vaut 1
- Tant que $n//2$ n'est pas nul
    - si n est impair nous remplaçons p par px
    - nous remplaçons x par x^2 et n par $n//2$.

Le principe d'un algorithme dichotomique est de diviser un problème en deux sous-problèmes.

```
def puissance2(x, n):
    p = 1
    while n != 0:
        if n % 2:
            p = p * x
        x  = x * x
        n = n // 2
    return p
```

Complexité logarithmique.

Considérons l'écriture binaire de n (des précisions sont données au chapitre 11).

$n = n_0 + n_1 \times 2 + n_2 \times 2^2 + \ldots + n_k \times 2^k = \sum_{i=0}^{k} n_i 2^i$, avec pour tout i, n_i égal à 0 ou à 1.

Nous pouvons alors écrire : $x^n = x^{\sum_{i=0}^{k} n_i 2^i} = x^{n_0}(x^2)^{n_1}(x^{2^2})^{n_2}\ldots(x^{2^k})^{n_k}$.

Dans le cas le moins favorable, nous pouvons considérer que nous avons de l'ordre de $4k$ opérations à effectuer, où k est le nombre de chiffres dans l'écriture binaire de n, soit environ $\log_2(n)$ opérations (k multiplications, k élévations au carré, et k calculs de quotient et de reste).

Le logarithme en base b d'un nombre entier n est de l'ordre du nombre de chiffres dans l'écriture en base b de n.

Si $c_{10}(n)$ est le nombre de chiffres dans l'écriture décimale de n, alors nous pouvons en déduire que $c_{10}(n) - 1 \leq \log_{10}(n) < c_{10}(n)$, soit $\log_{10}(n) < c_{10}(n) \leq \log_{10}(n) + 1$.

Si $c_2(n)$ est le nombre de chiffres dans l'écriture binaire de n, alors nous pouvons en déduire que $c_2(n) - 1 \leq \log_2(n) < c_2(n)$, soit $\log_2(n) < c_2(n) \leq \log_2(n) + 1$.

Concaténation de chaînes

On donne une liste de n chaines de caractères, chacune de longueur p. L'objectif est d'écrire une fonction qui renvoie la chaine obtenue par concaténation des n chaines et d'étudier sa complexité en fonction de n et de p. Pour cette étude nous considérons qu'une chaîne est lue et écrite en mémoire caractère par caractère, qu'il n'y a pas plus de lectures que d'écritures et donc que les opérations *élémentaires* à compter sont uniquement les écritures de caractères. On suppose enfin que l'affectation `ch1 = ch1 + ch2`, nécessite de réécrire en mémoire les caractères de la chaîne `ch1` suivis des caractères de la chaîne `ch2`.

- 1er cas : les n chaines sont identiques de longueur p.

En Python, on peut utiliser l'opérateur `*`.

```
def concat1(liste):
    return n * liste[0]
```

La longueur de la chaîne renvoyée est prévisible, c'est $n \times p$.

La fonction écrit les caractères de la chaîne `liste[0]` l'un après l'autre, n fois à la suite. Le nombre d'écritures est donc $p \times n$.

Il est clair qu'on peut difficilement faire mieux.

- 2e cas : les n chaines sont distinctes.

En Python, on peut utiliser l'opérateur `+`.

```
def concat2(liste):
    ch = ''
    for chaine in liste:
        ch = ch + chaine
    return ch
```

À chaque passage dans la boucle, pour concaténer `ch` et `chaine`, tous les caractères de la valeur courante de `ch` sont écrits. La première chaine de la liste est donc écrite n fois, la deuxième $n-1$ fois, etc, jusqu'à la dernière écrite une fois. Le nombre total d'écritures est donc $pn + p(n-1) + \ldots + p = pn(n+1)/2$. Le coût est donc de l'ordre de $p \times n^2/2$. Ceci signifie une complexité quadratique.

Peut-on être plus efficace ? Il apparaît que pour certains algorithmes, on peut passer d'un coût linéaire à un coût logarithmique en utilisant une méthode par dichotomie.

Cela signifie que l'on coupe la liste en deux parties de même longueur à une unité près. On concatène les chaînes de la première moitié, les chaînes de la deuxième moitié, puis on concatène les deux résultats. Pour concaténer les chaînes de chaque partie, on recommencent la procédure avec chaque partie, et ainsi de suite.

Cette procédure décrit une approche dite *descendante* qui peut être programmée de manière récursive (voir TP 5). L'algorithme programmé ci-dessous suit une approche dite *ascendante*. Les chaînes consécutives de la liste sont concaténées deux par deux. Nous avons donc $n//2$ concaténations de deux chaînes de longueur p. La dernière chaîne si elle est seule est ajoutée à cette liste. On recommence alors l'opération avec $n//4$ concaténations de deux chaînes de longueur $2p$, et on continue ce processus jusqu'à obtenir une seule chaîne de longueur np.

Les concaténations au niveau 1 coûtent $2p \times n/2 = p \times n$ opérations.

Les concaténations au niveau 2 coûtent $4p \times n/4 = p \times n$ opérations.

Et ainsi de suite. Le nombre de niveaux étant environ $\log_2(n)$, le nombre total d'opérations est de l'ordre de $p \times n \log_2(n)$.

Nous pouvons donc envisager une complexité de l'ordre de $p \times n \log_2(n)$.

Pour simplifier l'écriture du programme proposé ci-dessous, on suppose que la taille de la liste est une puissance de 2. La variable `taille` représente la longueur des chaînes à concaténer divisée par p.

```
def concat3(liste):
    n = len(liste)
    taille = 1
    while taille < n:
        for g in range(0, n-taille, 2*taille):
            liste[g] = liste[g] + liste[g+taille]
        taille = 2 * taille
    return liste[0]
```

Remarques

— Les chaînes créées sont nombreuses et il apparaît ici une autre forme de coût, que l'on appelle un coût en espace. En effet ces chaînes occupent de l'espace en mémoire et cet espace doit être évalué dans certains cas.

— On obtiendrait le même type de résultat en remplaçant les chaînes par des listes. Mais Python propose des méthodes plus efficaces, liées au caractère *mutable* des listes. Par exemple, il faut éviter d'écrire la concaténation `lst = lst + [elt]` et utiliser plutôt `lst.append(elt)`.