Aix*Marseille université
Initiative d'excellence
Année 2019-20

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Bloc 5 : TP 5

Programmation dynamique 1

1 Suite de Fibonacci

La suite de Fibonacci est définie par $F_0 = F_1 = 1$ et, pour tout $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. On peut la calculer par le programme récursif suivant :

```
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Question 1 Combien d'appels récursifs sont nécessaires au total pour calculer la valeur de F_n, pour $n \in \mathbf{N}$?

Question 2 Écrire une fonction non récursive réalisant le calcul de manière ascendante. A-t-on besoin de stocker tous les résultats des appels précédents ?

Point culturel Une autre technique que le calcul ascendant existe. Il s'agit de la *mémoïsation* (terme introduit par l'informaticien Donald Michie en 1968, provenant du latin *memorandum*, littéralement *qui doit être rappelé*) qui consiste à conserver la fonction récursive, mais à stocker les résultats des appels récursifs au fur et à mesure de leur retour. La façon la plus simple de réaliser la mémoïsation dans un programme récursif est d'utiliser un dictionnaire global qu'on utilise au sein de la fonction elle-même :

```
fibonacci_resultat = {}
def fibonacciMemo(n):
    if n <= 1:
        return 1
    elif n not in fibonacci_resultat:
        fibonacci_resultat[n] = fibonacciMemo(n - 1) + fibonacciMemo(n - 2)
    return fibonacci_resultat[n]
```

C'est bien, mais cela demande de modifier la fonction récursive qu'on a écrit au début… Il y a mieux en Python, à l'aide de décorateurs. On écrit une fonction `memoize` qui prend en argument une fonction (récursive) et qui doit la modifier pour appliquer la mémoïsation. Ensuite, on utilise un décorateur `@memoize` juste avant la définition de la fonction `fibonacci` et le tour est joué !

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return 1
```

```
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Finalement, on se rend compte qu'on a réinventé une technique déjà disponible dans la bibliothèque `functools`, à base de *cache LRU* (pour *least recently used*) :

```
from functools import lru_cache
@lru_cache(maxsize=None) # ou maxsize=n pour une taille de cache de n éléments
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Il est alors intéressant de pouvoir avoir des statistiques sur l'utilisation du cache. Essayez cela par exemple...

```
[fibonacci(n) for n in range(16)]
fibonacci.cache_info()
```

Autre exemple pour s'entraîner Dessiner le graphe d'appels, puis appliquer des méthodes de calcul ascendant et de memoïsation pour le calcul des coefficients binomiaux de manière récursive :

$$\binom{n}{p} = \begin{cases} 1 & \text{si } n = p \\ n & \text{si } p = 1 \\ \binom{n-1}{p-1} + \binom{n-1}{p} & \text{si } 1 < p < n \end{cases}$$

2 Formatage équilibré d'un paragraphe

Intéressons-nous au problème du formatage équilibré d'un paragraphe. On se donne ainsi une suite de n mots $m_1, \ldots, m_n$ de longueurs $\ell_1, \ell_2, \ldots, \ell_n$ et on souhaite imprimer ce paragraphe de manière équilibrée sur des lignes ne pouvant contenir qu'un maximum de M caractères chacune : on supposera donc que $\ell_i \leq M$ pour tout i. Le critère d'équilibre qu'on choisit est le suivant : si une ligne contient les mots de m_i à m_j inclus et qu'on laisse un caractère d'espacement entre chaque mot, le nombre de caractères d'espacements supplémentaires à la fin de la ligne est

$$f(i,j) = M - (j-i) - \sum_{k=i}^{j} \ell_k$$

L'objectif est de minimiser la somme des cubes des nombres de caractères d'espacement présents à la fin de chaque ligne, hormis la dernière ligne (qui peut donc être presque vide si nécessaire). On appelle *déséquilibre de la ligne* la quantité $f(i,j)^3$.

Par exemple, on a trois possibilités pour formater la première ligne de la Déclaration universelle des droits de l'homme, en lignes de 20 caractères (puisque le mot suivant est trop long pour pouvoir tenir sur celle-ci) :

```
Tous␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
les.................
....................
....................
```

```
Tous les␣␣␣␣␣␣␣␣␣␣␣
êtres...............
....................
....................
```

```
Tous les êtres␣␣␣␣␣␣
humains.............
....................
....................
```

Dans le premier cas, le déséquilibre de la première ligne est $f(1,1)^3 = (M-\ell_1)^3 = 16^3$; dans le second cas, il est égal à $f(1,2)^3 = (M-1-\ell_1-\ell_2)^3 = 12^3$; dans le troisième cas, il est égal à $f(1,3)^3 = (M-2-\ell_1-\ell_2-\ell_3)^3 = 6^3$.

Un premier algorithme de formatage équilibré de paragraphe est la stratégie gloutonne consistant à remplir les lignes les unes après les autres en mettant à chaque fois le plus de mots possibles sur la ligne en cours.

Question 3 Montrer sur un exemple simple que la stratégie gloutonne ne produit pas nécessairement le formatage optimal.

On applique désormais un algorithme de programmation dynamique. On note donc $d(i)$ la valeur minimale du déséquilibre total occasionné par le formatage du paragraphe des mots $m_i, m_{i+1}, \ldots, m_n$.

Question 4 Trouver une formule récursive pour calculer $d(i)$, en n'oubliant pas le cas où tous les mots peuvent tenir sur la dernière ligne.

Question 5 En déduire un algorithme de programmation dynamique calculant le déséquilibre minimal qu'on puisse obtenir pour une valeur de M et des longueurs $\ell_1, \ldots, \ell_n$ données (qu'on suppose toutes inférieures à M). On supposera écrite une fonction calculant $f(i, j)$ pour un tableau de longueurs de mots et des indices i et j donnés.

Question 6 Quelle est la complexité de l'algorithme précédent ?

Question 7 Considérons désormais que le déséquilibre d'une ligne abritant les mots d'indice i à j inclus est donné par $f(i, j)$ et non plus $f(i, j)^3$. Montrer alors que l'algorithme glouton renvoie une solution optimale. *Indication : on pourra montrer et utiliser le fait que la fonction $j \mapsto f(i, j) + d(j + 1)$ est décroissante.*

3 Ordonnancement avec une ressource

On considère n demandes d'utilisation d'une ressource rare (une machine coûteuse, un expert, un espace publicitaire...). Chaque demande d'utilisation $i = 0, \ldots, n-1$ est caractérisée par une date de début $d[i]$, une date de fin $f[i]$ et un poids $p[i]$ qui représente le gain obtenu si la demande i est satisfaite. Comme il y a une seule ressource, si les intervalles de deux demandes s'intersectent (c'est-à-dire que pendant un intervalle de temps non nul, les deux demandes ont besoin de la ressource), l'une des deux demandes ne pourra pas être satisfaite. Pour maximiser le gain, on cherche à trouver un sous-ensemble de demandes de poids maximum dont les intervalles sont deux-à-deux disjoints. Le poids d'un ensemble de demandes est la somme des poids des demandes de l'ensemble.

On suppose que les intervalles sont numérotés par ordre croissant des dates de fin. Pour $i = 1, \ldots, n$, on définit $M[i]$ le poids maximum d'un sous-ensemble de l'ensemble de i premières demandes $\{0, 1, \ldots, i-1\}$. On pose donc $M[0] = 0$ voulant dire que le poids maximal associé à l'ensemble vide de demandes est nul.

Considérons comme exemple l'ensemble de 7 demandes définies par

i	0	1	2	3	4	5	6
$d[i]$	1	1	4	3	6	8	6
$f[i]$	2	3	5	7	7	10	10
$p[i]$	100	200	200	400	100	200	400

Question 8 Calculer les valeurs $M[i]$ pour $i = 0, 1, \ldots, 7$ dans l'exemple ci-dessus.

Question 9 Pour $i = 0, 1, ..., n-1$, soit $r[i]$ le nombre d'intervalles qui se terminent avant le début de l'intervalle i. Calculer les valeurs de $r[i]$ pour l'exemple ci-dessus, pour $i = 0, \ldots, n-1$.

Question 10 Décrire un algorithme qui calcule les valeurs de $r[i]$ pour $i = 0, \ldots, n-1$ de façon efficace.

Question 11 Donner une formule récursive qui permet de calculer $M[i+1]$ à partir des valeurs de $M[r[i]]$ et de $M[i]$ pour $i = 0, \ldots, n-1$. Justifier.

Question 12 Décrire un algorithme de programmation dynamique qui calcule les valeurs $M[i]$ pour $i = 0, 1, \ldots, n$. Quelle est la complexité de cet algorithme ?

Question 13 Quelles informations faut-il sauvegarder pour pouvoir calculer a posteriori un ensemble d'intervalles de poids maximum, c'est-à-dire de poids $M[n]$? Compléter l'algorithme de la question précédente pour qu'il sauvegarde cette information et décrire un algorithme qui l'utilise pour calculer l'ensemble d'intervalles de poids maximum.