TD 09 – Algo avancé: Programmation Dynamique

Exercice 1. *la suite de Fibonacci*

1. Dans votre langage de programmation préféré [1], programmez la fonction de Fibonacci f de manière récursive naïve. Pour rappel, $f(0) = 0$, $f(1) = 1$ et $f(n) = f(n-2) + f(n-1)$ pour tout $n > 1$.
☞

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

print(fibo(600))
```

2. En utilisant votre programme, essayez de calculer $f(600)$. Donnez une borne inférieure sur le nombre $A(n)$ d'appels récursifs de votre fonction sur l'entrée n.
☞ $A(0) = A(1) = 1$. Pour tout $n > 1$, $A(n) = A(n-2) + A(n-1) + 1 \geq A(n-2) + A(n-2) = 2A(n-2)$. Et donc $A(n) \geq 2^{\lfloor n/2 \rfloor} \geq 2^{n/2-1}$. C'est une borne inférieure très grossière, mais elle montre déjà que $A(n)$ est exponentiel en n. Donc $A(600) \geq 2^{299} \geq 10^{90}$ est supérieur au nombre estimé d'atomes dans l'univers observable (environ 10^{80}). C'est donc normal que le programme ne s'arrête pas.

3. Le problème du programme naïf est qu'il calcule la même valeur plusieurs fois. Par exemple, pour calculer $f(5)$ il calcule $f(3)$ et $f(4)$, mais pour calculer $f(4)$, il calcule de nouveau $f(3)$.
Proposez un programme plus intelligent qui rempli un tableau t de façon à ce que $t[i] = f(i)$. Calculez $f(600)$.
☞

```
def fibo2(n):
        t = [0]*(n+1)
        t[0] = 0
        t[1] = 1
        for i in range(2,n+1):
                t[i] = t[i-2] + t[i-1]
        return t[n]

print(fibo2(600))
```

On constate que ce programme a un temps d'exécution (et utilise un espace) linéaire en n.
Le résultat de cette fonction est un entier à 126 chiffres :

110433070572952242346432246767718285942590237357555606380008891875277701705731473925618404421867819924194229142447517901959200

Exercice 2. *Les coefficients binomiaux*

Un coefficient binomial $\binom{n}{k}$ représente le nombre de sous-ensembles S de taille k d'un ensemble E de taille n. [2]
Pour calculer $\binom{n}{k}$, on peut utiliser les formules bien connues :

1. Donc, probablement en Python ?
2. Intuitivement : c'est le nombre de manières différentes que j'ai de choisir k objets parmi n objets disponibles (sans se préoccuper de l'ordre de mes choix).

— $\binom{n}{0} = 1$, car il n'y a qu'un sous-ensemble de taille 0 ;
— $\binom{n}{k} = 0$ si $n < k$, car un sous-ensemble de E est plus petit que E ;
— $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ sinon , car, en prenant $e \in E$, il y a deux cas :
  — $e \in S$ et dans ce cas-là il reste à choisir $k - 1$ éléments parmi $E \setminus \{e\}$;
  — $e \notin S$ et dans ce cas-là il reste à choisir k éléments parmi $E \setminus \{e\}$.

1. programmez la fonction $binome(n, k)$ qui calcule $\binom{n}{k}$ de manière récursive naïve.
☞

```
def binome(n,k):
        if n < k:
                return 0
        if k==0:
                return 1
        return binome(n-1,k-1)+binome(n-1,k)

print(binome(600,300))
```

2. En utilisant votre programme, essayez de calculer $\binom{600}{300}$. Donnez une borne inférieure sur le nombre $A(n)$ d'appels récursifs de $binome(n, n/2)$ quand n est pair.
☞ $binome(n, n/2)$ va appeler
— $binome(n-1, n/2-1)$ qui va appeler $binome(n-2, n/2-2)$ et $binome(n-2, n/2-1)$;
— $binome(n-1, n/2)$ qui va appeler $binome(n-2, n/2-1)$ et $binome(n-2, n/2)$.

Donc, $binome(n-2, n/2-1)$ est appelé 2 fois plus de fois (au moins) que $binome(n, n/2)$. Donc, par récurrence, $binome(0, 0)$ va être appelé au moins $2^{n/2}$ fois. Donc, encore une fois, en essayant de calculer $\binom{600}{300}$, le nombre d'appels récursifs est supérieur au nombre estimé d'atomes dans l'univers observable. C'est donc normal que le programme ne s'arrête pas.

3. Proposez un programme plus intelligent pour calculer $\binom{n}{k}$ et calculez $\binom{600}{300}$.
☞
On fait un tableau à deux dimensions tel que $t[i][j] = \binom{i}{j}$ de taille $(n+1) * (k+1)$. On sait déjà que pour tout i, $t[i][0] = 1$. De plus pour $j > i$, $t[i][j] = 0$, et pour tout $j \leq i$, $t[i][j] = t[i-1][j-1] + t[i-1][j]$.

```
def binome2(n,k):
        t = [ [0]*(k+1) for _ in range(n+1) ]

        for i in range(n+1):
                t[i][0] = 1

        for i in range(n+1):
                for j in range(1,k+1):
                        if i < j:
                                t[i][j] = 0
                        else:
                                t[i][j] = t[i-1][j-1]+t[i-1][j]
        return t[n][k]

print(binome2(600,300))
```

On constate que cet programme a un temps d'exécution (et utilise un espace) quadratique (en $O(n * k)$).
Le résultat de cette fonction est un entier à 180 chiffres :

135107941996194268514474877978504530397233945449193479925965721786474150408005716961950480198274469818673334131365837249043900490761151591695308427048
536947621976068789875968372656

Exercice 3. *Nombre mots acceptés par un AFD*

Dans cet exercice on s'intéresse au problème suivant. Étant donné un AFD A (automate fini déterministe) et un entier k, combien vaut $nombre\_mots(A, k)$ avec $nombre\_mots(A, k)$ le nombre de mots de taille k qui sont acceptés par A?

Pour votre programme :

— A est donné sous la forme d'un tuple :

$$(alphabet, etats, transitions, etat\_initial, etats\_acceptants);$$

— $alphabet$ est un ensemble de lettres ($\{'a', 'b'\}$ par exemple);

— $etats$ et $etats\_acceptants$ sont des ensembles d'entiers ($\{0, 1, 2\}$ par exemple);

— $etat\_initial$ est un entier (0 par exemple);

— $transitions$ est un dictionnaire qui à un couple (état,lettre) associe un nouvel état. Par exemple :

$$\{(0, 'a') : 1, (0, 'b') : 0, (1, 'a') : 1, (1, 'b') : 0\}.$$

1. Écrire un programme naïf qui calcule $nombre\_mots(A, k)$.

☞ Notons $A = (\Sigma, Q, \delta, q_0, F)$.

Une première méthode consiste à écrire une fonction qui étant donné l'automate A et un mot $w \in \Sigma^*$, décide si $w \in L(A)$. Il suffit ensuite d'itérer sur chaque mot de Σ^k et de compter le nombre de mots qui sont acceptés.

```
import itertools

def accepte(A,w):
        alphabet,etats,transitions,etat_initial,etats_acceptants = A
        etat = etat_initial
        for lettre in w:
                etat = transitions[(etat,lettre)]
        return etat in etats_acceptants

def nombre_mots1(A,k):
        alphabet,etats,transitions,etat_initial,etats_acceptants = A
        nb = 0
        for w in itertools.product( alphabet ,repeat=k):
                if accepte(A,w):
                        nb+=1
        return nb
```

L'un des problèmes de cette méthode est qu'on calcule plusieurs fois la même chose. Par exemple, pour savoir si a^k est accepté on calcule :

— en $O(k-1)$ étapes de calcul, dans quel état q_i on se retrouve après avoir lu a^{k-1};

— en $O(1)$ étapes de calcul, $q_j = \delta((q_j), a)$ et si $q_j \in F$.

À côté de ça, pour savoir si $a^{k-1}b$ est accepté on calcule :

— en $O(k-1)$ étapes de calcul, dans quel état q_i on se retrouve après avoir lu a^{k-1};

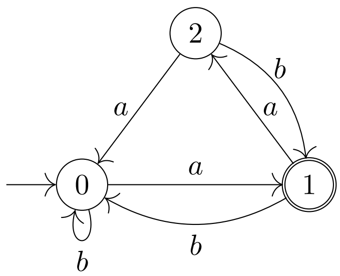
— en $O(1)$ étapes de calcul, $q_\ell = \delta((q_i), b)$ et si $q_\ell \in F$.

Le fait qu'il calcule 2 fois q_i rend notre algorithme plus lent.

On peut être plus malin et s'éviter de calculer plusieurs fois la même chose avec une fonction récursive.

```
def nombre_mots2(A,k):
        alphabet,etats,transitions,etat_initial,etats_acceptants = A
        if k == 0:
                if etat_initial in etats_acceptants:
                        return 1
                return 0
        nb = 0
        for lettre in alphabet:
                etat = transitions[(etat_initial,lettre)]
                nouveau_A = alphabet,etats,transitions,etat,etats_acceptants
                nb+= nombre_mots2(nouveau_A,k-1)
        return nb
```

2. Regardez jusqu'à quelle valeur de k vous arrivez à calculer $nombre\_mots(A, k)$ avec A l'automate suivant :



Ça nous donne :

$$A = (\{'a', 'b'\}, \{0, 1, 2\}, \{(0, 'a') : 1, (0, 'b') : 0, (1, 'a') : 2, (1, 'b') : 0, (2, 'a') : 0, (2, 'b') : 1\}, 0, \{1\})$$

☞ La deuxième version de l'algorithme est un peu plus rapide, mais en gros l'algorithme devient lent vers $k = 20$.

0	0
1	1
2	1
3	3
4	5
5	11
6	21
7	43
8	85
9	171
10	341
11	683
12	1365
13	2731
14	5461
15	10923
16	21845
17	43691
18	87381
19	174763
20	349525

3. Estimer le temps de calcul de votre algorithme.

☞ Dans le premier algorithme, c'est le nombre de mots dans Σ^k fois environ k étapes. Donc en $O(k|\Sigma|^k)$. Le second est un peu plus rapide, car on s'économise environ k étapes de calcul par mot, mais ça reste exponentiel en $O(|\Sigma|^k)$.

4. Proposer un algorithme de programmation dynamique pour résoudre ce problème en temps polynomial. Calculez $nombre\_mots(A, 2000)$.

☞ Clairement, notre algorithme ne peut pas se permettre d'énumérer tous les mots acceptants : il y en a trop. Il faut donc être plus astucieux et trouver une manière de compter les mots acceptants sans les énumérer.

Regardons dans quel état on finit quand on lit les mots de taille 2 :

— $aa : q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2$.
— $ab : q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0$.
— $ba : q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1$.
— $bb : q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0$.

Après avoir lu ab, on se retrouve dans le même état qu'après avoir lu bb. Du coup, pour tout $w \in \Sigma^{k-2}$, $abw \in L(A) \Leftrightarrow bbw \in L(A)$. Donc, on peut gagner du temps en ne calculant que $|\{abw \in L(A)\}|$, car

$$2|\{abw \in L(A)\}| = |\{abw \in L(A)\}| + |\{bbw \in L(A)\}|.$$

D'une manière plus générale : Pour tout $\ell \in [0, k-1]$,

$$|\{w \in \Sigma^k \mid q_0 \xrightarrow{w} q_j| = \sum_{q_i \in Q} |\{u \in \Sigma^\ell \mid q_0 \xrightarrow{u} q_i\}| * |\{v \in \Sigma^{k-\ell} \mid q_i \xrightarrow{v} q_j\}|.$$

Pour résoudre le problème, on va remplir un dictionnaire d de façon à ce que :

$$d[(i, \ell)] = |\{u \in \Sigma^\ell \mid q_0 \xrightarrow{u} q_i\}.$$

On peut l'initialiser avec le fait qu'avant de lire la moindre lettre on est dans l'état initial. Donc, $d[(i,0)] = \begin{cases} 1 \text{ si } i = 0; \\ 0 \text{ sinon.} \end{cases}$.

De plus, pour tout $\ell > 0$ et $q_i \in Q$,

$$d[(i,\ell)] = \sum_{\substack{q_j \in Q \\ u \in \Sigma \\ \delta(q_j,u)=q_i}} d[(j,\ell-1)].$$

```python
def nombre_mots3(A,k):
        alphabet,etats,transitions,etat_initial,etats_acceptants = A
        d = { (etat,l):0 for etat in etats for l in range(k+1) }
        d[0,0] = 1
        for l in range(k):
                for etat in etats:
                        for lettre in alphabet:
                                nouvel_etat = transitions[(etat,lettre)]
                                d[(nouvel_etat,l+1)] += d[(etat,l)]
        return sum( [d[(etat,k)] for etat in etats_acceptants ] )
```

On obtiens : $nombre\_mots(A, 2000) = 382710231758084841410944400392560 6...$