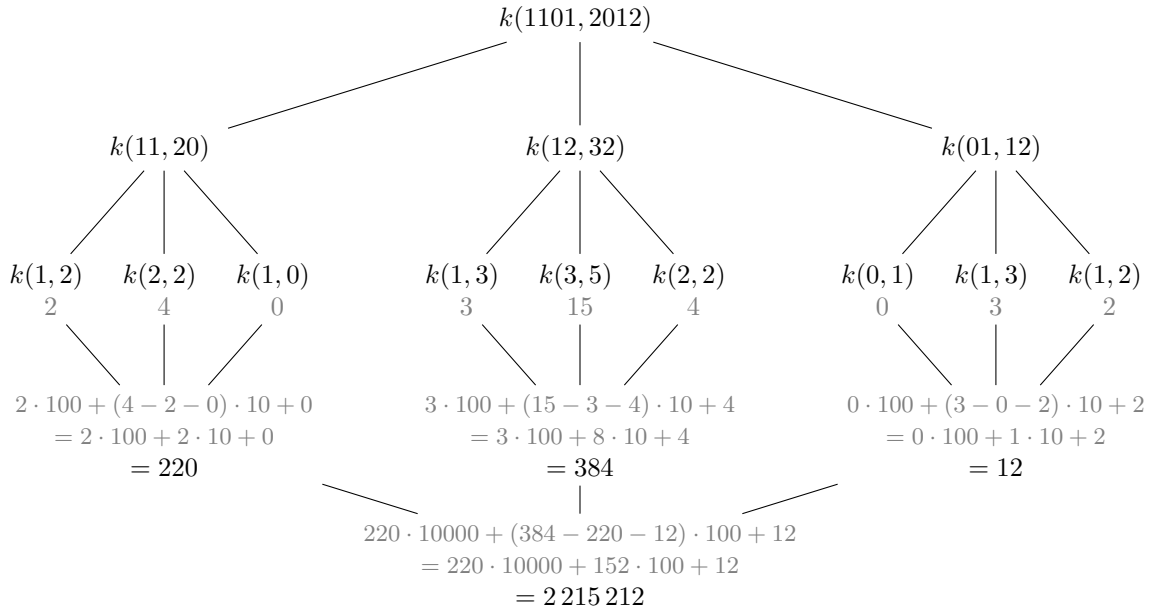
EA4 – Éléments d'algorithmique
Contrôle du 4 mars 2020 – Sujet B

Correction

Exercice 1 :

Décrire le déroulement de l'algorithme de Karatsuba pour calculer le produit 1101×2012, en représentant en particulier l'arbre des appels récursifs. Prendre le produit de chiffres décimaux comme cas de base.

La manière la plus parlante de décrire le déroulement d'un algorithme récursif consiste à représenter l'arbre des appels récursifs, ainsi que l'arbre (inversé) de leurs retours.
Dans le cas de l'algorithme de Karatsuba, placer à gauche l'appel fournissant les chiffres de poids fort et à droite celui fournissant les chiffres de poids faible améliore grandement la lisibilité !



Exercice 2 :

Pour chaque couple de fonctions, indiquer si les assertions sont vraies ou fausses.

				$f \in \Omega(g)$		$f \in O(g)$		$f \in \Theta(g)$	
$f(n) = 3(n^2-1)^2$	$\in \Theta(n^4)$	$g(n) = 4n^3 + 5n$	$\in \Theta(n^3)$	**oui**	non	oui	**non**	oui	**non**
$f(n) = \log(3(n^2-1)^2)$	$\in \Theta(\log n)$	$g(n) = \log(4n^3 + 5n)$	$\in \Theta(\log n)$	**oui**	non	**oui**	non	**oui**	non
$f(n) = \log(4n^3)$	$\in \Theta(\log n)$	$g(n) = 4(\log n)^3$	$\in \Theta((\log n)^3)$	oui	**non**	**oui**	non	oui	**non**
$f(n) = 4n^3$	polynôme	$g(n) = 3^{(4n)}$	exponentielle	oui	**non**	**oui**	non	oui	**non**
$f(n) = 3^{(n+4)}$	$= 81 \cdot 3^n$	$g(n) = 3^{(4n)}$	$= 81^n$	oui	**non**	**oui**	non	oui	**non**
$f(n) = 3^{(n+4)}$	$= 81 \cdot 3^n$	$g(n) = 3^n$		**oui**	non	**oui**	non	**oui**	non
$f(n) = \log(n^4)$	$= 4 \log n$	$g(n) = \sqrt[3]{n}$	$= n^{1/3}$	oui	**non**	**oui**	non	oui	**non**
$f(n) = \log(4^n)$	$= 2n$	$g(n) = \sqrt[3]{n}$	$= n^{1/3}$	**oui**	non	oui	**non**	oui	**non**

(note : log *désigne le logarithme en base 2)*

Exercice 3 :

Pour chacun des algorithmes `foo_`i`_` suivants, donner une relation de récurrence satisfaite par le nombre $A_i(n)$ d'additions effectuées pour une entrée de taille n, et en déduire (sans démonstration) l'ordre de grandeur de $A_i(n)$.

```
def foo_1_(T) :
  return 0 if len(T) == 0 else foo_1_(T[3:]) + 3
```

Remarque : si $n \leqslant 2$, `T[3:]=[]`. Pour $n > 2$, `foo_1_` effectue une addition en plus de (celles cachées dans) l'appel récursif sur un tableau de taille $n-3$.

$$A_1(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \text{ ou } 2 \\ A_1(n-3)+1 & \text{si } n > 2 \end{cases} \quad \Longrightarrow \quad A_1(n) = \left\lceil \frac{n}{3} \right\rceil \in \Theta(n).$$

```
def foo_2_(T) :
  return 0 if len(T) == 0 else sum(e for e in T) + foo_2_(T[2:])
```

Pour $n > 1$, `foo_2_` effectue n additions (une plus $n-1$ dans l'appel à `sum`) en plus de (celles cachées dans) l'appel récursif sur un tableau de taille $n-2$.

$$A_2(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ A_2(n-2)+n & \text{si } n > 0 \end{cases} \quad \Longrightarrow \quad A_2(n) \in \Theta(n^2).$$

```
def foo_3_(T) :
  return 0 if len(T) == 0 else foo_3_(T[:len(T)//2]) + 1
```

Pour $n > 1$, `foo_3_` effectue une addition en plus de (celles cachées dans) l'appel récursif sur un tableau de taille $n//2 = \lfloor n/2 \rfloor$.

$$A_3(n) = \begin{cases} 0 & \text{si } n = 0 \\ A_3(n//2)+1 & \text{si } n > 0 \end{cases} \quad \Longrightarrow \quad A_3(n) = 1 + \lfloor \log_2(n) \rfloor \in \Theta(\log n).$$

Exercice 4 :

On considère l'algorithme suivant :

```
def F(n) :
    return 1 if n < 4 else 2 * F(n-1) + F(n-4)
```

Soit $C(n)$ le nombre d'opérations arithmétiques sur des entiers effectuées lors de l'exécution de `F(n)`. Donner une définition de $C(n)$ par récurrence.

Pour $n \geqslant 4$, `F` effectue 2 opérations (`*` et code+) en plus de (celles cachées dans) les appels récursifs sur $n-1$ et $n-4$. Donc :

$$C(n) = \begin{cases} 0 & \text{si } n < 4 \\ C(n-1)+C(n-4)+2 & \text{sinon.} \end{cases}$$

En déduire que $C(n)$ est croissante, puis que $C(n) \in \Omega(2^{n/4})$.

Par définition, $C(k) \geqslant 0$ pour tout k, donc en particulier, pour tout $n \geqslant 4$, $C(n-4)+2 > 0$, et $C(n) > C(n-1)$. La fonction C est donc croissante.

En particulier, pour tout $n > 4$, on a $C(n-1) \geqslant C(n-4)$, d'où $C(n) \geqslant 2C(n-4)$.
On en conclut que $C(n) \in \Omega(2^{n/4})$.

Proposer un algorithme `Fbis(n)` calculant la même valeur que `F(n)` de manière plus efficace.

Les méthodes utilisées pour améliorer le calcul de la suite de Fibonacci peuvent être adaptées pour n'importe quelle suite récurrente linéaire. Mais il faut tenir compte de l'ordre de cette récurrence – c'est-à-dire la taille de la « fenêtre » utilisée. Ici, la récurrence est d'ordre 4, il faut donc 4 variables temporaires pour adapter la méthode itérative, ou une matrice d'ordre 4 pour adapter la méthode matricielle.

Par programmation dynamique *(version avec complexité en espace optimale) :*

```
def Fbis(n) :
  if n < 4 : return 1
  ppprev, pprev, prev, last = 1, 1, 1, 1
  for i in range(4, n) :
    ppprev, pprev, prev, last = pprev, prev, last, 2 * last + ppprev
  return last
```

Par exponentiation de matrice *(produit 4×4 et exponentiation binaire non écrits ici) :*

```
def Fter(n) :
  if n < 4 : return 1
  M = [ [2, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0] ]
  return puissance(M, n-1)[0][0] # calcule M**(n-1) par exponentiation binaire
```

Quel est l'ordre de grandeur du nombre d'opérations arithmétiques sur des entiers effectuées par `Fbis(n)` ?

Méthode itérative (`Fbis`) : $\Theta(n)$ opérations arithmétiques (additions)
Méthode matricielle (`Fter`) : $\Theta(\log n)$ opérations arithmétiques (additions et multiplications)

Est-ce une mesure pertinente de sa complexité en temps ?

Oui et non... oui au sens où l'ordre de grandeur des autres opérations (tests, incrémentations de compteur, affectations) n'est pas plus grand.
Mais non, au sens où ces opérations arithmétiques ne peuvent pas être considérées comme des opérations élémentaires du fait de la croissance (linéaire) des entiers manipulés. Comme on l'a vu pour Fibonacci, la complexité en temps de `Fbis` est quadratique et non linéaire, et celle de `Fter` est super-linéaire et non logarithmique.

Exercice 5 :

On s'intéresse au problème suivant : étant donné un tableau `T` de n entiers, et un entier `x`, déterminer un élément de `T` le plus proche de `x` (c'est-à-dire un elément `e` de `T` minimisant `abs(x-e)`).

Décrire un algorithme `plusProche(x, T)` permettant de résoudre ce problème.

Un bête parcours séquentiel convient ; penser à sauvegarder à la fois l'élément le plus proche (ou son indice) et la distance correspondante pour éviter de la recalculer inutilement.

```
def plusProche(x, T) :
  if T==[] : return None
  near, dist = T[0], abs(x-T[0])
  for y in T[1:] :
    d = abs(x-y)   # pour éviter les recalculs
    if d < dist :
      near, dist = y, d
  return near
```

Quel est l'ordre de grandeur de la complexité (en temps) de cet algorithme ?

La complexité en temps est linéaire : $\Theta(n)$.

On suppose maintenant que `T` est strictement croissant. Décrire un algorithme strictement plus efficace `plusProcheBis(x, T, deb=0, fin=None)` pour résoudre ce problème dans ce cas.

Une recherche dichotomique modifiée permet de trouver l'élément le plus proche. Attention, il peut s'agir soit de l'élément immédiatement inférieur, soit de l'élément immédiatement supérieur (sauf cas particuliers où `x` *est présent dans* `T`*, ou au contraire où* `x` *n'est pas dans l'intervalle* `[T[0],T[-1]]`*). Il faut donc soit adapter un peu la condition permettant de décider si l'appel récursif est fait à gauche ou à droite, soit relancer sur un peu plus de la moitié du tableau pour conserver l'élément médian.*

Solution 1 : *tester si l'élément médian est le plus proche en comparant avec ses voisins, et sinon relancer sur le demi-tableau approprié (élément médian non compris)*

```
def plusProcheBis(x, T, deb=0, fin=None) :
  # retourne l'élément le plus proche de x dans le sous-tableau T[deb:fin] supposé trié
  if fin == None : fin = len(T)     # appel initial
  if fin == deb : return None       # longueur 0 (appel initial nécessairement)
  mil = (fin+deb)//2                # longueur > 0
  if mil > deb and abs(x-T[mil-1]) < abs(x-T[mil]) :
    return plusProcheBis(x, T, deb, mil)
  elif mil+1 < fin and abs(x-T[mil+1]) < abs(x-T[mil]) :
    return plusProcheBis(x, T, mil+1, fin)
  else : return T[mil]              # notamment si mil = deb = fin-1
```

Solution 2 : *tester seulement si l'élément médian est plus petit ou plus grand que* `x`*, puis relancer sur le demi-tableau approprié (mais attention : élément médian compris, cette fois). Dans ce cas, la récursion est toujours menée à son terme, jusqu'à un tableau de longueur 2.*

```
def plusProcheBis(x, T, deb=0, fin=None) :
  # retourne l'élément le plus proche de x dans le sous-tableau T[deb:fin] supposé trié
  if fin == None : fin = len(T)       # appel initial
  if fin == deb : return None         # longueur 0
  if fin == deb+1 : return T[deb]     # longueur 1
  if fin == deb+2 :                   # longueur 2
    if abs(x-T[deb]) < abs(x-T[deb+1]) : return T[deb]
    return T[deb+1]
  mil = (fin+deb)//2                  # longueur > 2
  if x < T[mil] : return plusProcheBis(x, T, deb, mil+1) # indice mil compris
  else : return plusProcheBis(x, T, mil, fin)            # indice mil compris
```

Rappel : *il ne faut pas faire de recopie de tableau lors du passage de paramètre à l'appel récursif, faute de quoi la complexité en temps est linéaire. D'où la nécessité de toujours travailler sur le même tableau* `T`*, en utilisant les indices* `deb` *et* `fin` *pour définir la portion concernée.*

Quel est l'ordre de grandeur de la complexité (en temps) de cet algorithme ?

Le principe de dichotomie permet d'obtenir ici une complexité en temps logarithmique : $\Theta(\log n)$.