TP (S2) 2 Preuves & Complexité

Objectifs

- Savoir prouver la terminaison d'une boucle.
- Savoir prouver un programme simple en utilisant un invariant de boucle.
- Savoir calculer la complexité temporelle et/ou spatiale d'un programme.

1. PREUVE D'UN PROGRAMME

Exercice 1 **Terminaison de la recherche dichotomique** [Sol 1] On rappelle la fonction de recherche dichotomique d'un élément dans une liste triée :

```
def dicho(L:list, v:int)->bool:
    """ Renvoie True si v appartient à L où L est une liste \
    ↪ d'entiers triée dans l'ordre croissant """
    i_deb = 0
    i_fin = len(l) - 1
    trouve = False
    while not trouve and i_deb <= i_fin:
        i_m = (i_deb + i_fin) // 2
        if L[i_m] == v:
            trouve = True
        elif L[i_m] < v:
            i_deb = i_m + 1
        else:
            i_fin = i_m - 1
    return trouve
```

Notons ℓ_k (resp. d_k, resp. f_k) la longueur (resp. l'indice de début, resp. l'indice de fin) de la liste à la fin de l'itération k ; on a donc de la relation $f_k = d_k + \ell_k - 1$. On suppose par l'absurde que la boucle ne termine jamais.

1. Montrer que ℓ_k est au moins divisé par deux pour chaque tour de boucle.
2. En déduire que l'algorithme termine. *Rappelons qu'un invariant a été donné dans le TP1 :* $P(k)$ *: «* $(v \notin L) \vee (v \in L[d_k : f_k + 1]$ *».*

Exercice 2 **Fonction d'Ackermann** [Sol 2] Cette fonction est définie sur $\mathbb{N}^2$ récursivement comme ceci :

$$\forall (m,n) \in \mathbb{N}^2, \quad A(m,n) = \begin{cases} n+1 & \text{si } m = 0, \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0, \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$

1. Écrire la fonction `A(m: int, n:int)->int` qui renvoie la valeur de $A(m,n)$, et la tester.
2. Montrer que pour tout $(m,n) \in \mathbb{N}^2$, l'appel à `A(m,n)` se termine et renvoie un naturel.

Exercice 3 **Le tri par sélection** [Sol 3] Pour mettre en oeuvre ce tri nous aurons besoin de la fonction suivante :

```
def pos_max(L: list)->int:
    """ Renvoie l'indice de la dernière occurrence du maximum de \
    ↪ L
        où L est une liste de nombres supposée non vide """
    m, p, n = L[0], 0, len(L) #initialisation avec le premier \
    ↪ élément
    for i in range(1, n):  # on parcourt les éléments suivants
        if L[i] >= m:  # si on trouve un élément supérieur ou \
        ↪ égal
            m, p = L[i], i  # on met à jour nos variables
    return p
```

1. Proposer un invariant pour la boucle `for` et faire la preuve de cette fonction.

2. Le principe de ce tri est le suivant :

   - Première étape : on trouve le plus grand élément de L et on l'échange avec le dernier élément de L.
   - Deuxième étape : on trouve le plus grand élément parmi les $n-1$ premiers éléments de L et on l'échange avec l'avant-dernier élément de L.
   - etc.

   2.1) Écrire la fonction `tri_select(L: list)->none` qui réalise à l'aide de la fonction précédente le tri par sélection en place de la liste L.

   2.2) Faire la preuve.

Exercice 4 **Méthode dichotomique continue** [Sol 4] On considère la fonction suivante qui met en œuvre la méthode dichotomique pour la résolution à ε près d'une équation $f(x) = 0$ possédant une solution unique dans un intervalle $[a; b]$:

```
def dicho(f: Callable, a: float, b: float, epsilon: \
↪ float)->float:
    """ Calcule une valeur approchée à epsilon près de l'unique
        solution de f(x)=0 dans l'intervalle [a;b],
        pré-condition : ... """
    fa = f(a)
    while b-a > 2*epsilon:
        milieu = (a+b)/2
        fm = f(milieu)
        if fa*fm <= 0:
            b = milieu
        else:
            a = milieu
            fa = fm
    return (a+b)/2
```

1. Compléter la pré-condition dans la docstring et commenter le code.

2. Établir la terminaison de la boucle `while`.

3. Proposer un invariant de boucle et faire la preuve.

4. Proposer une version récursive de cette fonction.

2. COMPLEXITÉ D'UN ALGORITHME

Exercice 5 **Recherche d'un maximum** [Sol 5] Écrire une fonction `IndiceMaxListe` qui renvoie le plus petit indice de l'élément maximal d'une la liste `L` de flottants. Quelle est sa complexité temporelle ?

Exercice 6 **Fonction mystère** [Sol 6] On considère la fonction suivante :

```
def Mystere(L) :
    lg = len(L)
    doublon = False
    for i in range(lg-1) :
        for j in range(i+1,lg):
            if L[j] == L[i] :
                doublon = True
    return doublon
```

1. Compléter cette fonction en écrivant sa signature et sa docstring.

2. Calculer sa complexité.

Exercice 7 **Calculs de complexités** [Sol 7] Évaluer la complexité exacte de chacune des fonctions suivantes puis donner sa complexité asymptotique.

1.
```
for i in range(5, n-5):
    for j in range(i-5, i+5):
        x += 1
```

2.
```
for i in range(n):
    for j in range(i):
        for k in range(j):
            x += 1
```

3.
```
i = n
while i > 1:
    x = x+1
    i = i//2
```

Exercice 8 **Complexité de l'exponentiation rapide** [Sol 8] On reprend l'algorithme d'exponentiation rapide pour calculer x^n qui a été présenté dans le cours :

```python
def expR(x: float, n: int)-> float :
    """Renvoie x^n pour x réel et n entier naturel."""
    X = x
    N = n
    R = 1
    while N != 0 :
        if N%2 == 0 :
            N = N//2
        else :
            R = R*X
            N = (N-1)//2
        X = X*X
    return R
```

1. Compter le nombre de multiplications effectuées ($X * X$ et $R * X$) dans l'algorithme d'exponentiation rapide lorsque $n = 2^p$ (on néglige le temps d'exécution des autres opérations élémentaires devant celui de la multiplication).
2. Lorsque $2^p \leqslant n < 2^{p+1}$, donner un encadrement du nombre de multiplications.
3. **[Complexité exacte]** De manière générale, on peut décomposer n en binaire. Plus précisément, on l'écrit sous la forme :

   $$n = 2^{p_1} + \cdots + 2^{p_k}, \quad \text{avec :} \quad 0 \leqslant p_1 < \cdots < p_k \text{ entiers tels que } \quad 2^{p_k} \leqslant n < 2^{p_k+1}.$$

   3.1) Montrer que le nombre de multiplications est $p_k + k + 1$.

   3.2) En déduire que le nombre $C(n)$ de multiplications vérifie :

   $$\log_2(n) < C(n) \leqslant 2\log_2(n) + 2.$$ *(On pourra remarquer en justifiant que $k \leqslant p_k + 1$)*

Exercice 9 **Suite de FIBONACCI** [Sol 9] On s'intéresse dans cet exercice à la suite de FIBONACCI dont on rappelle la définition :

$$F_0 = 0, \quad F_1 = 1, \quad \text{et :} \quad \forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n.$$

1. Écrire une fonction itérative `Fibo1` à qui on fournit un entier naturel n et qui renvoie F_n. Calculer la complexité temporelle de cette fonction.

2. Proposer une fonction récursive « naturelle » `Fibo2` qui renvoie le même résultat que `Fibo1` et calculer sa complexité temporelle.

3. On propose la fonction récursive suivante :

```python
def Fibo3(a, b, n):
    if n == 0:
        return a
    else:
        return Fibo3(b, a+b, n-1)
```

Déterminer la signature et la docstring de cette fonction, et expliquer comment on pourrait l'utiliser pour calculer F_n. Calculer sa complexité.

SOLUTIONS DES EXERCICES

Solution 1

1. Discutons suivant la parité de ℓ_k :

- Si ℓ_k est pair, notons $\ell_k = 2p$ avec p entier,ce qui donne $f_k = d_k + 2p - 1$.
  L'étape $k+1$ construit l'indice milieu $\left\lfloor \frac{d_k + f_k}{2} \right\rfloor = d_k + p - 1$.
  En cas de décalage à gauche, on pose $d_{k+1} = d_k$ et $f_{k+1} = d_k + p - 2$ donc $\ell_{k+1} = f_{k+1} - d_{k+1} + 1 = p - 1$.
  En cas de décalage à droite, on pose $d_{k+1} = d_k + p$ et $f_{k+1} = f_k$ donc $\ell_{k+1} = f_{k+1} - d_{k+1} + 1 = p$.
  Dans ces deux cas, on a bien $\ell_{k+1} \leqslant \frac{\ell_k}{2}$.
- Si ℓ_k est impair, notons $\ell_k = 2p + 1$ avec p entier, donc $f_k = d_k + 2p$.
  L'indice milieu devient $\left\lfloor \frac{d_k + f_k}{2} \right\rfloor = d_k + p$.
  Les décalages à gauche et à droite donnent tous deux $\ell_{k+1} = p \leqslant \frac{\ell_k}{2}$.

2. La suite ℓ_k est donc une suite d'entiers strictement décroissante. Il existe donc un rang k_0 tel que $\ell_{k_0} < 1$. Ceci contredit l'hypothèse initiale : la boucle s'arrête.

Solution 2

1. La fonction :

```
def A(m: int, n:int)->int :
    """ Renvoie la valeur de A(m,n) (fonction d'Ackermann)
        m et n sont des naturels (pré-condition) """
    if m == 0:
        return n+1
    elif n == 0:
        return A(m-1,1)
    else:
        return A(m-1,A(m,n-1))
```

```
>>> A(3, 1)
13
```

Essayez de faire grandir les paramètres; vous verrez que l'on attend très vite la taille limite de la pile de récursivité.

2. On montre pour tout m, P(m) : « $\forall n \in \mathbb{N}$, l'appel à A(m,n) se termine et renvoie un naturel » .

Initialisation. C'est vrai au rang $m = 0$, d'après le cas terminal.

Hérédité. Supposons P(m) pour un entier m, on montre alors P($m+1$) en faisant une récurrence sur n.

- Pour $n = 0$, A($m+1,0$) se termine et renvoie un naturel.
- Supposons pour un entier n que A($m+1,n$) se termine et renvoie un naturel, alors pour calculer A($m+1,n+1$), il faut calculer A(m,A($m+1,n$)), or le nombre p = A($m+1,n$) est calculable et c'est un naturel (hypothèse de récurrence sur n), et donc A(m,p) se termine et renvoie un naturel (hypothèse de récurrence sur m). Donc P($m+1$) est vrai.

Ainsi, par principe de récurrence (sur m), P(m) est vraie pour tout m, d'où le résultat.

Solution 3

1. On peut proposer l'invariant suivant P(k) : « m_k = L[p_k] = max(L[0 : $k+1$]) » .

Initialisation. P(0) est vraie, puisque m_0 = L[0] = L[0 : 1] d'après l'initialisation.

Hérédité. Supposons P(k) vraie pour un entier k avec $k < n-1$, il y a donc une itération $k+1$, la valeur de i à l'itération $k+1$ est $i_{k+1} = k+1$, on effectue le test.

- si L[k+1] >= m_k alors m_{k+1} = L[k+1] et $p_{k+1} = k+1$, compte-tenu de l'hypothèse de récurrence on a bien m_{k+1} = max(L[0 : $k+2$]) = L[p_{k+1}],
- et si L[k+1] < m_k, alors $m_{k+1} = m_k$ et $p_{k+1} = p_k$, et il est clair, compte-tenu de l'hypothèse de récurrence, que m_{k+1} = max(L[0 : $k+2$]) = L[p_{k+1}].

Après la dernière itération (numéro $n-1$), on a P($n-1$) c'est à dire m_{n-1} = L[p_{n-1}] = max(L[0 : n]), mais L[0:n]=L, donc la valeur p_{n-1} renvoyée est bien un indice où se trouve le maximum de L (la dernière occurrence du maximum à cause de l'inégalité large dans le test).

2. **2.1)** La fonction :

```python
def tri_select(L: list)->None:
    """ Réalise le tri par sélection en place de la liste L,
        où L est une liste de nombres """
    n = len(L)
    #la première étape se fait sur la liste en entier: L[:n]
    #la deuxième sur ses n-1 premiers éléments: L[:n-1]
    #la dernière sur ses deux premiers éléments: L[:n-2]
    for i in range(n-1):
        p = pos_max(L[:n-i])
        L[p], L[n-i-1] = L[n-i-1], L[p]
```

2.2) On remarque que si la liste L est de longueur au plus 1, la fonction ne fait rien, ce qui est attendu. On suppose maintenant que l'on a une liste L de longueur $n \geqslant 2$.

La boucle `for` est exécutée $n-1$ fois et se termine forcément. On peut proposer l'invariant suivant pour la boucle P(k) : « la liste `L[n-k : n]` est triée dans l'ordre croissant et tous ses éléments sont supérieurs ou égaux à ceux de la liste `L[:n-k]` ».

Initialisation. P(0) est vraie car `L[n:n]` est vide (une liste vide est triée).

Hérédité. Supposons P(k) vraie avec $k < n-1$, il y a donc une itération $k+1$, la valeur de i à l'itération $k+1$ est $i_{k+1} = k$, l'élément maximal de `L[0:n-k]` est échangé et se retrouve alors à l'indice $n-i-1 = n-k-1$, par hypothèse de récurrence cet élément est inférieur ou égal à tous ceux de `L[n-k:n]` (qui eux sont dans l'ordre croissant), donc la liste `L[n-k-1:n]` est triée dans l'ordre croissant, de plus, le plus petit élément de cette liste est `L[n-k-1]` qui était l'élément maximal de `L[0:n-k]`, par conséquent les éléments de `L[0:n-k-1]` sont tous inférieurs ou égaux à `L[n-k-1]` et donc inférieurs ou égaux à tous ceux de `L[n-k-1:n]`, donc P($k+1$) est vraie.

Correction. À l'issue de l'itération $n-1$ (dernière itération) l'invariant dit que la liste `L[n-(n-1):n]`, c'est à dire `L[1:n]`, est triée dans l'ordre croissant, et ses éléments sont supérieurs ou égaux à ceux de `L[:n-(n-1)]` c'est à dire `L[0:1]` (qui ne contient que `L[0]`), par conséquent la liste L est bien triée dans l'ordre croissant.

Solution 4

1. Pré-condition : « `epsilon` >0, `a < b`, f est continue, `f(a)*f(b) <= 0` et il y a une unique solution sur $[a,b]$ ». L'unicité de la solution est garantie lorsque f est strictement monotone sur $[a;b]$.

2. On a $a_0 = a$, $b_0 = b$, on montre ensuite par récurrence que $b_k - a_k = \dfrac{b_0 - a_0}{2^k}$, on a donc une suite réelle de limite nulle, il existe un indice k tel que $b_k - a_k \leqslant \varepsilon$, la boucle s'arrête donc.

3. Lorsque la précondition est remplie, on sait qu'il existe un unique $c \in [a;b]$ tel que $f(c) = 0$. On propose alors l'invariant P(k) : « $fa_k = f(a_k)$, $a \leqslant a_k \leqslant c \leqslant b_k \leqslant b$ et $f(a_k) \times f(b_k) \leqslant 0$ ».

   Initialisation. P(0) est vraie (on suppose la précondition vérifiée).

   Hérédité. Si P(k) est vraie et s'il y une itération $k+1$: la variable `milieu` contient $\dfrac{a_k + b_k}{2}$ qui est le milieu de l'intervalle $[a_k; b_k]$, le test a deux issues possibles :

   - Si $fa * fm <= 0$ alors $f(a_k)$ et fm sont de signes contraires donc f s'annule entre a_k et le milieu (c'est forcément en c par unicité), dans ce cas on $b_{k+1} =$ milieu et $a_{k+1} = a_k$, donc $fa_{k+1} = fa_k = f(a_k) = f(a_{k+1})$, $a \leqslant a_{k+1} \leqslant c \leqslant b_{k+1} \leqslant b$ et $f(a_{k+1}) \times f(b_{k+1}) \leqslant 0$.
   - Si $fa * fm > 0$ alors $f(a_k)$ et fm sont de même signe, donc par hypothèse de récurrence fm et $f(b_k)$ sont de signes contraires, f s'annule entre le milieu et b_k (c'est forcément en c par unicité), dans ce cas on $b_{k+1} = b_k$, $a_{k+1} =$ milieu, et $fa_{k+1} = f(a_{k+1})$ donc $fa_{k+1} = f(a_{k+1})$, $a \leqslant a_{k+1} \leqslant c \leqslant b_{k+1} \leqslant b$ et $f(a_{k+1}) \times f(b_{k+1}) \leqslant 0$.

   Dans tous les cas P($k+1$) est vérifiée, ce qui achève la récurrence.

   Correction. Soit n le numéro de la dernière itération, P(n) est vraie, donc $a_n \leqslant c \leqslant b_n$ et puisqu'il n'y a pas d'itération $n+1$, on a $b_n - a_n \leqslant 2\varepsilon$, ce qui entraîne que la distance entre c et $\dfrac{a_n + b_n}{2}$ (qui est le résultat renvoyé) est inférieure ou égale à ε.

4. Version récursive :

```python
def dichoRec(f: Callable, a: float, b: float, epsilon: \
↪ float)->float:
    """ Calcule une valeur approchée à epsilon près de l'unique
        solution de f(x)=0 dans l'intervalle [a;b],
        pré-condition : epsilon>0, a<b, f continue, \
        ↪ f(a)*f(b)<=0,
                        unique solution """
```

```
    if b-a <= 2*epsilon:
        return (a+b)/2
    else:
        milieu = (a+b)/2
        if f(a)*f(milieu) <= 0:
            return dichoRec(f,a,milieu,epsilon)
        else:
            return dichoRec(f,milieu,b,epsilon)
```

Solution 5 Fonction `IndiceMaxListe(L)`.

```
def IndiceMaxListe(L:list)->int :
    """ Calcule le plus petit indice de l'élément maximal
        d'une la liste L de flottants """
    lg = len(L)
    maxi = L[0]
    idx = 0
    for i in range(lg) :
        if L[i] > maxi:
            maxi = L[i]
            idx = i
    return idx
```

L'algorithme est de complexité asymptotique $\mathrm{O}(n)$.

Solution 6

1.

```
def TestDoublon(L:list)->bool :
    """
    Détermine si une liste contient un élément en au moins \
    ↪ deux exemplaires

    Parameters
    ----------
    L : list
        liste dont on veut savoir si elle admet des doublons.

    Returns
    -------
    bool
        True si la liste contient au moins un doublon et False \
        ↪ sinon.

    """
    lg = len(L)
    doublon = False
    for i in range(lg-1) :
        for j in range(i+1,lg) :
            if L[j] == L[i] :
                doublon = True
    return doublon
```

2. Notons n la longueur de la liste. Il y a $n-1$ passages dans la boucle en i et pour chaque $i \in [\![0, n-2]\!]$, $n-i-1$ passages dans la boucle en j. À chaque passage dans cette boucle en j, il y a au plus deux opérations élémentaires (un test et une affectation) et au mieux une seule opération élémentaire (seulement un test). On obtient donc :

$$\mathrm{C}_{\text{pire}}(n) = 2 + \sum_{i=0}^{n-2} 2(n-i-1) = 2 + n(n-1)$$

$$\mathrm{C}_{\text{meilleur}}(n) = 2 + \sum_{i=0}^{n-2} (n-i-1) = 2 + \frac{n(n-1)}{2}$$

L'algorithme est de complexité asymptotique $\mathrm{O}\left(n^2\right)$.

Solution 7

1.

$$\mathrm{C}(n) = \sum_{i=5}^{n-6} \sum_{j=i-5}^{i+4} 2 = 2 \sum_{i=5}^{n-6} (i+4-(i-5)+1) = 18(n-10)$$

d'où une complexité en $\mathrm{O}(n)$.

2.

$$\mathrm{C}(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \sum_{k=0}^{j-1} 2 = 2 \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} j = 2 \sum_{i=0}^{n-1} \frac{i(i-1)}{2} = \frac{n(n-1)(n-2)}{3}$$

d'où une complexité en $\mathrm{O}\left(n^3\right)$.

3. Pour tout entier positif n non nul, il existe un entier positif p tel que :

$$2^p \leqslant n < 2^{p+1}$$

Un tel entier est donné par $p = \lfloor \log_2(n) \rfloor$ où $\log_2(n)$ désigne le logarithme en base 2 de n. On a $i_{k+1} = i_k // 2$ pour tout $k \in [\![0\,,\,\mathrm{N}]\!]$ où N désigne le numéro de la

dernière itération (on montera après que N = p). Montrons par récurrence sur k que :

$$2^{p-k} \leqslant i_k < 2^{p+1-k}.$$

Initialisation. La propriété a déjà été initialisée.

Hérédité. Soit $k \in [\![0\,,\,N-1]\!]$, tel que $2^{p-k} \leqslant i_k < 2^{p+1-k}$. Alors on a la relation $i_k = 2i_{k+1} + r$ avec $r = 0$ ou 1. Ainsi :

$$2^{p-k} \leqslant 2i_{k+1} + r < 2^{p+1-k} \Longrightarrow 2^{p-k-1} \leqslant i_{k+1} + \frac{r}{2} < 2^{p-k}. \quad \text{Alors :}$$

- $i_{k+1} \leqslant i_{k+1} + \dfrac{r}{2} < 2^{p-k}$ donc $i_{k+1} < 2^{p-k}$. On a bien l'encadrement de droite.
- On a : $2^{p-k-1} \leqslant i_{k+1} + \dfrac{r}{2}$, or 2^{p-k-1} et i_{k+1} sont des entiers, on obtient alors $2^{p-k-1} \leqslant i_{k+1}$ en passant par exemple à la partie entière.

La propriété est donc héréditaire, par principe de récurrence, on a bien :

$$\forall k \in [\![0\,,\,N-1]\!], \quad 2^{p-k} \leqslant i_k < 2^{p+1-k}.$$

On déduit alors que l'entier p est aussi le nombre d'itérations du `while` ; en effet, lorsque $k < p$ la minoration entraîne que $i_k > 1$, et pour $k = p$, on a $1 \leqslant i_p < 2$ donc $i_p = 1$. Il y a donc pour la complexité :

- une première affectation,
- p passages dans la boucle avec, à chaque passage 5 opérations élémentaires (un test, deux affectations, une addition et une division),
- puis un dernier test où la condition `i > 1` n'est pas vraie.

On déduit :

$$C(n) = 1 + \left(\sum_{k=1}^{p} 5\right) + 1 = 5p + 2 = O(p) = \boxed{O\left(\log_2(n)\right)}.$$

Solution 8

1. La variable N est divisée par 2 à chaque passage dans la boucle, donc, $p+1$ fois et donc X est multiplié avec lui même $p+1$ fois, mais il y a aussi une autre multiplication quand N = 1 (R=R*X), ce qui fait $p+2$ multiplications en tout. La méthode naïve impose quant à elle $2^p - 1$ multiplications !

2. On a $N_{i+1} = \left\lfloor \dfrac{N_i}{2} \right\rfloor$ et $N_0 = n$, donc $2^p \leqslant N_0 < 2^{p+1}$, on en déduit par récurrence sur i que $2^{p-i} \leqslant N_i < 2^{p-i+1}$ (faite dans l'exercice **??**), on en déduit que $N_{p+1} = 0$ et donc comme dans la question précédente, le nombre d'itérations est $p+1$, à chaque itération on a 1 ou 2 multiplications, donc $p+1 \leqslant C(n) \leqslant 2p+2$ où $p = \lfloor \log_2(n) \rfloor$, et donc $\boxed{\log_2(n) + 1 \leqslant C(n) \leqslant 2\log_2(n) + 2}$. Malheureusement cet encadrement n'est pas assez fin pour pouvoir obtenir un grand o.

3. **3.1)** Faisons une récurrence sur k.

Initialisation. La formule est vraie au rang $k = 1$.

Hérédité. Supposons la vraie au rang k et soit $n = 2^{p_1} + \cdots + 2^{p_{k+1}}$ avec $0 \leqslant p_1 < \cdots < p_{k+1}$ des entiers.

- N va d'abord être divisé par 2, p_1 fois, ce qui va donner p_1 fois le produit X * X.
- On a alors $N = 1 + 2^{p_2 - p_1} + \cdots + 2^{p_{k+1} - p_1}$ qui est impair, et on va alors avoir deux multiplications supplémentaires (X = X * X et R = R * X).
- On obtient alors $N = 2^{p_2 - p_1 - 1} + \cdots + 2^{p_{k+1} - p_1 - 1}$. Par hypothèse de récurrence, il y restera $p_{k+1} - p_1 - 1 + k + 1$ produits, ce qui fait en tout : $p_1 + 2 + p_{k+1} - p_1 - 1 + k + 1 = p_{k+1} + k + 2$: c'est la formule au rang $k+1$.

3.2) D'une part, on remarque qu'il y a k entiers distincts $p_1, \ldots, p_{k-1}$, on ne compte pas 0 puisqu'on peut avoir $p_1 = 0$) dans l'intervalle $[0, p_k]$ donc $\boxed{p_k \geq k-1}$.

D'autre part, on a $p_k = \lfloor \log_2(n) \rfloor$, d'où l'encadrement $p_k \leq \log_2(n) < p_k + 1$. Il en résulte :

$$\log_2(n) < p_k + 1 \leq \underbrace{p_k + k + 1}_{=C(n)} \leq 2p_k + 2 \leq 2\log_2(n) + 2$$

La borne de droite peut être atteinte, lorsque $n = 1 + 2 + 2^2 + \cdots + 2^k$.

Solution 9

1.

```
def Fibo1(n:int)->int :
    U, V = 0, 1
    for _ in range(n):
        U, V = V, U + V
    return U
```

On trouve $C(n) = 2 + (n-1) \times 3 = 3n - 1 = \boxed{O(n)}$.

2.

```
def Fibo2(n:int)->int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
```

```
        return Fibo2(n-1) + Fibo2(n-2)
```

On trouve $C(0)=1$, $C(1)=2$ et $C(n)=2+C(n-1)+C(n-2)+1=C(n-1)+C(n-2)+3$, en posant $u_n=C(n)+3$, la suite u est une suite de FIBONACCI avec $u_0=4$ et $u_1=5$, on en déduit l'expression de u_n, puis :

$$C(n)=\left(2+3\frac{\sqrt{5}}{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n+\left(2-3\frac{\sqrt{5}}{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n-3=\boxed{O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)}$$

On obtient donc une complexité exponentielle !

3.

```
def Fibo3(a:float,b:float,n:int)->float:
    """
    Détermine le terme F_{n} d'une suite de Fibonacci ayant \
    ↪ pour a pour terme d'indice 0 et b pour terme d'indice 1

    Parameters
    ----------
    a : float
    b : float
    n : int

    Returns
    -------
    float
        la valeur du réel F_{n}

    """
    if n == 0:
        return a
    else:
        return Fibo3(b, a+b, n-1)
```

En appelant `Fibo3(0,1,n)`, on obtient le terme F_n de la suite initiale de FIBONACCI. En effet, on montrerait par récurrence double, que pour tout $n\geqslant 1$, $F_n=G_{n-1}$ où (G_n) est la suite définie par :

$$G_0=b,\quad G_1=a+b,\quad \forall n\in\mathbb{N},\quad G_{n+2}=G_n+G_{n+1}.$$

On a alors une nouvelle fonction, qui est de complexité bien meilleure. Puisqu'on trouve $C(0)=1$ (1 test), et $C(n)=C(n-1)+2$ (un test et une addition) d'où $C(n)=2n+1=\boxed{O(n)}$.