Exercice 1. Suite de Fibonacci

On souhaite calculer les nombres de Fibonacci grâce à un algorithme de type diviser pour régner. On rappelle que la suite de Fibonacci $(f_n)_{n\in\mathbb{N}}$ est définie par :

$$f_0 = f_1 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N} : f_{n+2} = f_{n+1} + f_n$$

Cette suite vérifie $f_n = \Theta(\phi^n)$ où $\phi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or. On admet que :

$$\forall (p,q) \in \mathbb{N}^2 : f_{p+q+2} = f_{p+1}f_{q+1} + f_p f_q \qquad \text{(se montre par récurrence sur } p\text{)}$$

1. Soit $k \in \mathbb{N}^*$. Exprimer f_{2k+2}, f_{2k+1} et f_{2k} à partir de f_k et f_{k+1}.
2. Déterminer les complexités des fonctions :

```
let rec fibo1 n = match n with
  | 0 | 1 -> 1
  | n -> fibo1 (n-2) + fibo1 (n-1);;
```

```
(* 'aux k' renvoie (f_k, f_{k+1}) *)
let fibo2 n =
  let rec aux k = match k with
    | 0 -> 1,1
    | n -> let fkm1, fk = aux (k-1) in
           fk, fkm1 + fk in
  fst (aux n);;
```

```
let rec fibo3 n = match n with
  | 0 | 1 -> 1
  | 2 -> 2
  | n -> let fk = fibo3 (n/2) in
         let fkp1 = fibo3 (n/2+1) in
         if n mod 2 = 0
         then fk*fk + (fkp1 - fk)*(fkp1 - fk)
         else 2*fk*fkp1 - fk*fk;;
```

3. Écrire une fonction « `fibo: int -> int` » qui prend en entrée un entier n et renvoie f_n. Quelle est la complexité de votre fonction ? Vous devez obtenir une meilleure complexité que celles trouvées à la question 2.

Exercice 2. Tri rapide

Le tri rapide est un algorithme qui utilise le principe du diviser pour régner afin de trier une liste d'entiers. La procédure est la suivante :

→ Si la liste est vide, elle est triée

→ Sinon (par exemple, `[5; 4; 9; 3; 5; 0]`) :

(i) On récupère le premier élément de la liste (appelé le pivot) et on crée deux nouvelles listes `li1` et `li2`. La liste `li1` contient tous les éléments inférieurs ou égaux au pivot et la liste `li2` contient tous les éléments strictement supérieurs au pivot. Avec l'exemple ci-dessus, le pivot est `5`, la liste `li1` est `[4; 3; 5; 0]` et la liste `li2` est `[9]`. Notez que le pivot lui-même n'appartient ni à `li1`, ni à `li2`. En revanche, le deuxième `5` de la liste initiale apparaît dans la liste `li1`.

(ii) On trie `li1` et `li2` récursivement. Avec l'exemple ci-dessus, on obtient `[0; 3; 4; 5]` et `[9]`.

(iii) On obtient le résultat final grâce à des concaténations. Avec l'exemple ci-dessus, on renvoie :

```
[0; 3; 4; 5] @ (5::[9])
```

1. Écrire une fonction « `tri_rapide: 'a list -> 'a list` » qui applique l'algorithme de tri rapide à une liste.
2. Quelle est la complexité pire cas du tri rapide ?
3. Quelle est la complexité meilleur cas du tri rapide ?
4. Rappeler les complexités pire cas et meilleur cas du tri fusion. Comparer.

Pour information, la complexité moyenne du tri rapide est en $\Theta(n \log n)$ où n est la taille de la liste en entrée. En pratique, le tri rapide est en général plus rapide que le tri fusion.

Exercice 3. Nombre d'inversions dans une liste

Soit `n` $\in \mathbb{N}$ et `li` une liste de taille `n` contenant une et une seule fois chaque élément de $[\![0, n-1]\!]$. Dans cet exercice, pour `i` $\in [\![0, n-1]\!]$, on note `li[i]` l'élément d'indice `i` de `li`. Une *inversion* pour la liste `li` est un couple d'entiers `(i, j)` tel que $0 \leqslant i < j \leqslant n-1$ et `li[i] > li[j]`.

1. Sur feuille, trouver les 7 inversions pour la liste `[3; 4; 0; 1; 5; 2]`.
2. (a) Décrire un algorithme simple pour calculer le nombre d'inversions dans une liste.
   (b) Quelle est la complexité de votre procédure ?

Dans la suite, on s'intéresse à une méthode de type "diviser pour régner" pour calculer le nombre d'inversions. Soit `li` une liste et `m` le nombre d'inversions dans cette liste. Afin de calculer `m`, on note `li1` et `li2` les deux listes obtenues lorsqu'on coupe `li` au milieu. Par exemple :

Si `li = [3; 4; 0; 1; 5; 2]`	alors `li1 = [3; 4; 0]`	et	`li2 = [1; 5; 2]`
Si `li = [3; 4; 0; 1; 5]`	alors `li1 = [3; 4; 0]`	et	`li2 = [1; 5]`

3. Soit n_1 la taille de `li1` et n_2 la taille de `li2`. Exprimer n_1 et n_2 en fonction de `n`.

Soit m_1 le nombre d'inversions dans `li1` et m_2 le nombre d'inversions dans `li2`. On note également m' le nombre de couples `(i, j)` avec `i` $\in [\![0; n_1 - 1]\!]$, `j` $\in [\![0; n_2 - 1]\!]$ et `li1[i] > li2[j]`.

4. Exprimer `m` en fonction de m_1, m_2 et m'.

Afin de déterminer m', on va supposer que les deux listes `li1` et `li2` ont été triées. Ainsi, on dispose de deux listes triées `t1` et `t2` contenant les mêmes éléments que `li1` et `li2`.

5. Écrire une fonction « `fusion: int list -> int list -> int -> int * (int list)` » qui prend en entrée `t1`, `t2`, et n_1, et renvoie m' ainsi qu'une liste triée `t` contenant les mêmes éléments que `li`. Votre fonction devra s'exécuter en un temps Θ`(n)`.
6. En déduire une fonction « `nb_inv: int list -> int` » qui renvoie le nombre d'inversions dans la liste donnée en entrée. Étant donné que la fonction `List.length` est coûteuse, vous devez l'utiliser au plus une fois.
7. Quelle est la complexité de la fonction `nb_inv` ?

Exercice 4. Tri rapide en place

Un algorithme de tri est dit "en place" s'il n'effectue pas de copie de la structure de données à trier. Par exemple, le tri rapide vu dans l'exercice 2, n'est pas en place puisque les listes `li1` et `li2` contiennent des copies des éléments de la liste initiale. De même, le tri fusion vu dans le cours n'est pas un tri en place. Dans cet exercice, on essaye d'écrire une implémentation du tri rapide qui soit en place. Pour cela, on va trier des tableaux au lieu de trier des listes. Étant donnés un tableau `tab` et deux entiers `i` et `j` avec `i, j` $\in [\![0; n-1]\!]$, on notera `tab.(i,j)` le tableau contenant les éléments d'indices `k` $\in [\![i; j]\!]$:

```
tab.(i,j) = [|tab.(i), tab.(i+1), ..., tab.(j-1), tab.(j)|]
```

La principale difficulté est d'implémenter la division du tableau en utilisant le pivot (c'est à dire le point (i) de l'exercice 2). Pour cela, on considère un tableau `tab` de taille `n`, et deux entiers `i` et `j` tels que $0 \leqslant i \leqslant j \leqslant n-1$. On souhaite appliquer le point (i) de l'exercice 2 au tableau `tab.(i,j)` en choisissant pour pivot le dernier élément `tab.(j)`. L'idée est de diviser le tableau `tab.(i,j)` en quatre régions :

→ Une région contenant les éléments inférieurs ou égaux au pivot.

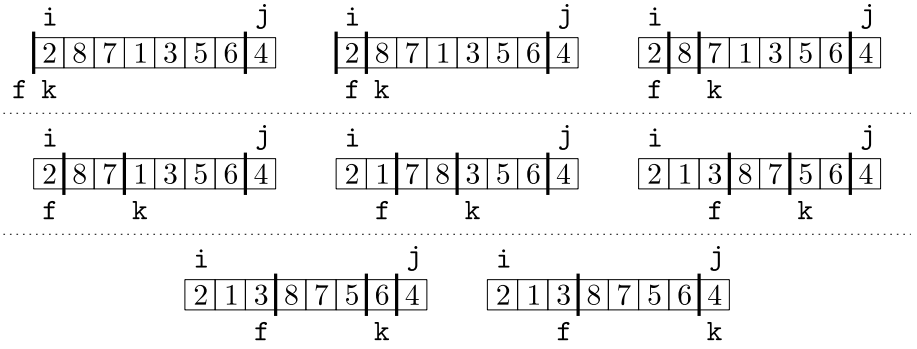
→ Une région contenant les éléments strictement supérieurs au pivot.

→ Une région contenant les éléments qui n'ont pas encore été comparés au pivot.

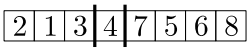
→ Une région contenant le pivot (cette région contient un seul élément).

Initialement, les deux premières régions sont vides et la troisième région contient `n-1` éléments. À chaque étape de la fonction `partition`, on considère le premier élément de la troisième région et on le place dans la région 1 ou 2 en fonction du résultat de sa comparaison avec le pivot. Par exemple, le schéma ci-dessous montre l'évolution du tableau `tab.(i,j) = [|2; 8; 7; 1; 3; 5; 6; 4|]` lors de l'utilisation de la fonction `partition`. Dans ce schéma :

– Les quatre régions sont séparées par les lignes verticales en gras.
– L'entier `f` est l'indice de la dernière case de la première région (si cette région est vide, `f` vaut `i-1`).
– L'entier `k` est l'indice du premier élément de la troisième région.
– À chaque étape de la fonction `partition`, on effectue zéro ou une transposition sur les éléments de `tab.(i,j)`, puis on met à jour les variables `f` et `k`.



À la fin de la fonction `partition`, il reste à échanger le pivot avec le premier élément de la deuxième région pour obtenir :



Lorsque la fonction `partition` a été exécutée, on trie les deux sous-tableaux récursivement à l'aide du tri rapide.

1. Écrire la fonction « `partition: 'a array -> int -> int -> int` » qui prend en entrée un tableau `tab` ainsi que les entiers `i` et `j`, et qui modifie le tableau `tab.(i,j)` comme décrit ci-dessus. Votre fonction renverra l'indice de la case du tableau dans laquelle se trouve le pivot (`i+3` pour l'exemple ci-dessus)
2. Écrire une fonction « `tri_rapide_en_place: 'a array -> unit` » qui trie le tableau donné en entrée à l'aide d'un tri rapide en place.

Exercice 5. Preuve du Master Theorem

Théorème (Master Theorem)**.** *Soit $A \in \mathbb{N}^*$, $K \in \left]1\,;+\infty\right[$, $f : \mathbb{N}^* \to \mathbb{R}_+$ et $n_0 \in \mathbb{N}^*$.*
Pour tout $i \in [\![1; A]\!]$, on considère $\varphi_i : \mathbb{N}^* \to \mathbb{R}$ une fonction bornée telle que pour tout $n \in \mathbb{N}^*$: $\frac{n}{K} + \varphi_i(n) \in [\![0; n-1]\!]$. On définit $T : \mathbb{N} \to \mathbb{R}_+$ par :

$$\begin{cases} T(n) \text{ est fixé} & \text{pour tout } n \leqslant n_0 \\ T(n) = \left[\sum_{i=1}^{A} T\left(\frac{n}{K} + \varphi_i(n)\right)\right] + f(n) & \text{pour tout } n > n_0 \end{cases} \tag{1}$$

Soit $\alpha = \log_K(A)$ et on suppose que $f(n) = \Theta(n^\beta)$ où $\beta > 0$. On a trois cas :
(i) Si $\alpha > \beta$ alors $T(n) = \Theta(n^\alpha)$
(ii) Si $\alpha = \beta$ alors $T(n) = \Theta(n^\alpha \log(n))$
(iii) Si $\alpha < \beta$ alors $T(n) = \Theta(n^\beta)$

Remarque. La définition de T donnée ci-dessus est assez lourde mais nécessaire pour pouvoir faire une preuve formelle. En pratique, lorsqu'on veut appliquer le Master Theorem, on se contente de définir T par la version condensée :

$$\begin{cases} T(n) \text{ est fixé} & \text{pour tout } n \leqslant n_0 \\ T(n) \approx A \times T\left(\frac{n}{K}\right) + f(n) & \text{pour tout } n > n_0 \end{cases} \tag{2}$$

Notez que dans l'équation (2), la fonction T est mal définie. En effet, $\frac{n}{K} \in \mathbb{N}$ n'est pas garanti et donc $T\left(\frac{n}{K}\right)$ peut ne pas avoir de sens. Pour autant, c'est l'équation (2) qui est utilisée en pratique, n'utilisez pas les notations de l'équation (1) en dehors de cet exercice.

Preuve

Pour montrer le Master Theorem, on note `f_rec` la fonction dont on essaye de déterminer la complexité (cette complexité est notée T dans le théorème) et on fixe $n \in \mathbb{N}$ un entier donné en entrée de `f_rec`. On dira qu'un appel récursif est à profondeur $p \in \mathbb{N}$ s'il est à profondeur p dans l'arbre des appels récursifs. En d'autres termes :

→ Il existe un seul appel à profondeur 0 qui est l'appel principal et qui a pour entrée n.

→ Les entrées des appels à profondeur 1 sont de la forme $\dfrac{n}{K} + \varphi_i(n)$ où $i \in [\![1, A]\!]$.

→ Plus généralement, les entrées des appels récursifs à profondeur $p+1$ sont de la forme $\dfrac{m}{K} + \varphi_i(m)$ où $i \in [\![1, A]\!]$ et m est une entrée d'un appel récursif à profondeur p avec $m > n_0$.

Dans un premier temps, on va montrer qu'il existe une constante $c_0 \geqslant 0$ indépendante de n et de p telle que l'entrée d'un appel récursif à profondeur p appartient à $\left[\dfrac{n}{K^p} - c_0, \dfrac{n}{K^p} + c_0\right]$. Pour cela, notons $M \geqslant 0$ un majorant des φ_i :

$$|\varphi_i(m)| \leqslant M \qquad \text{pour tout } i \in [\![1, A]\!] \text{ et tout } m \in \mathbb{N}^*.$$

1. Soit m l'entrée d'un appel à profondeur p.
   (a) Montrer que $\dfrac{n}{K^p} - \left[\displaystyle\sum_{k=0}^{p-1} \frac{M}{K^k}\right] \leqslant m \leqslant \dfrac{n}{K^p} + \left[\displaystyle\sum_{k=0}^{p-1} \frac{M}{K^k}\right]$.
   (b) En déduire qu'il existe une constante $c_0 \geqslant 0$ indépendante de n et de p telle que $m \in \left[\dfrac{n}{K^p} - c_0, \dfrac{n}{K^p} + c_0\right]$.
2. Pour $n \in \mathbb{N}^*$, on définit :
$$p_n = \left\lfloor \log_K \left(\frac{n}{c_0 + n_0} \right) \right\rfloor.$$
   Puisqu'on s'intéresse au comportement asymptotique de $T(n)$, on peut supposer que n est suffisamment grand pour que $p_n \geqslant 0$. Montrez que :
   (a) Aucun appel récursif à profondeur $p < p_n$ ne se fait sur une entrée $m \leqslant n_0$.
   (b) Tout appel récursif à profondeur p_n a pour entrée un entier $m \leqslant m_0$ où m_0 est une constante indépendante de n.

D'après la question 2a, tous les appels à profondeur $p < p_n$ font A appels récursifs. Ainsi si $p \leqslant p_n$, il y a exactement A^p appels à profondeur p.

Dans la suite, on va montrer le Master Theorem en remplaçant les Θ par des $\mathcal{O}$. Le cas des Ω se traite de la même façon et ne sera pas détaillé ici.

Afin de donner un majorant sur $T(n)$, on remarque que $T(n) = T_1(n) + T_2(n)$ où :

→ $T_1(n)$ est la somme des $f(m)$ où m parcourt tous les arguments des appels récursifs à profondeur $p < p_n$.

→ $T_2(n)$ est la somme des $T(m)$ où m parcourt tous les arguments des appels récursifs à profondeur p_n.

3. Montrer qu'il existe une constante $c_1 \geqslant 0$ indépendante de n telle que :
$$T_1(n) \leqslant c_1 n^\beta \sum_{p=0}^{p_n - 1} K^{p(\alpha - \beta)}$$

D'après la question 2b, chaque appel à profondeur p_n s'exécute en temps constant. En effet, si on note t_0 le maximum des $T(m)$ pour $m \leqslant m_0$ alors un appel à profondeur p_n s'exécute en temps au plus t_0. Notez que cet argument fonctionne car m_0 et donc t_0 sont des constantes indépendantes de n.

4. Montrer qu'il existe une constante $c_2 \geqslant 0$ indépendante de n telle que $T_2(n) \leqslant c_2 n^\alpha$.
5. En déduire le Master Theorem où les Θ sont remplacés par des $\mathcal{O}$.