Devoir surveillé n° 1 — corrigé

I Trois exercices

Exercice 1 : Prévoir le résultat

I.1) `a` est une fonctionnelle qui à une fonction, ici appelée `b`, associe sa valeur en 4, le tout divisé par 5. On en déduit que `b` doit être de type `int -> int`, en remarquant que les opérateurs sont des opérateurs d'entiers.

```
let a = function b -> (b 4) / 5;;
  a : (int -> int) -> int = <fun>
```

La liaison `b` précédente était locale et n'existe plus dans l'environnement. Le nom `a` est lié globalement à la fonction précédente, mais utilisé ici comme argument, masquant cette liaison globale. Il vaudrait mieux commencer par renommer la variable muette `a` ici pour ne pas se tromper. L'identifiant `b` est une fonction curryfié prenant `a` et `c` comme arguments. `c` et `d` sont nécessairement des entiers puisqu'ils sont additionnés. Ainsi `a` doit être une fonction. Elle prend en entrée la fonction qui ajoute `c` à son argument, donc de type `int -> int`, et renvoie un entier. Ainsi `a` doit être de type `(int -> int) -> int`. La fonction `b` renvoie la somme entre cet entier et `c`. On en déduit alors le type de `b`.

```
let b a = function c -> a (function d -> c + d) + c;;
  b : ((int -> int) -> int) -> int -> int = <fun>
```

Le `a` fait ici référence à la fonction globale précédente, dont le type correspond bien au premier argument de `b`. Il y a donc application partielle et on en déduit le type de `c`, nom qui n'a rien à voir avec celui utilisé comme nom d'argument précédemment.

```
let c = b a;;
  c : int -> int = <fun>
```

On applique enfin `c` sur l'argument `11`, ce qui revient à évaluer `b a 11`. On applique donc `a` sur la fonction qui ajoute 11 : on ajoute donc 11 à 4 et on divise ce résultat par 5 pour trouver 3. Dans le corps de `b`, ceci est ensuite ajouté à 11 et c'est donc 14 qui est renvoyé.

```
let d = c 11;;
  d : int = 14
```

`a` attend comme argument une fonction de type `int -> int` qui n'est pas celui de `b`. On obtient alors une erreur de type :

```
let e = a b;;
  This expression has type
    ((int -> int) -> int) -> int -> int,
  but is used with type int -> int.
```

I.2) Oui.

Exercice 2 : Tri insertion

Cf. TD.

Exercice 3 : Écriture en base b

I.6) Comme $32 = 3^3 + 3^2 + 2 \times 3$, sa représentation est $[1;1;2;0]$.

I.7) On a $16 = 2^4$ et $16^2 = 2^8 = 128$. L'entier représenté est $2 \times 128 + 16 + 11 = 539$.

I.8) Pour calculer a_0 il suffit de prendre $(n \mod b)$. On remarque ensuite que $\lfloor \frac{n}{b} \rfloor = \sum_{k=1}^{d} a_k b^{k-1} = \sum_{k=0}^{d-1} a_{k+1} b^k$. Ainsi $(a_{k+1})_{k \in [\![0;d-1]\!]}$ est la décomposition de $\lfloor \frac{n}{b} \rfloor$ dans la base b, que l'on peut calculer récursivement. On en déduit donc la fonction auxiliaire `ecriture_base_aux` de type `ecriture_base_aux` :

```
let rec decomposition n base =
  if n < base then
    [n]
  else
    (n mod base) :: (decomposition (n / base) base)
;;
```

C'est une fonction auxiliaire et non pas la fonction recherchée car on calcule la décomposition dans l'ordre croissant des indices et non la « représentation » comme demandé. Il suffit alors de renverser la liste :

```
let ecriture_base n base =
  rev (ecriture_base_aux n base)
;;
```

La complexité est linéaire en fonction du nombre de chiffres de n dans la décomposition en base b, la taille de la liste de sortie, c'est-à-dire aussi un $\Theta(\log_b n) = \Theta(\log n)$. On peut donner également une version récursive terminale qui calcule directement la représentation dans le bon ordre grâce à un accumulateur :

```
let rec ecriture_base_aux n base acc =
  if n = 0 then
    acc
  else
    ecriture_base_aux (n / base) base ((n mod base) ::
      ↪ acc)
;;
```

```
let ecriture_base n base = ecriture_base_aux n base [];;
```

I.9) C'est exactement l'évaluation d'un polynôme en un point, comme dans le DM n° 1. Encore une fois, il faut commencer par renverser la représentation.

```
let rec lecture_base_aux representation base =
  match representation with
  | [] -> 0
  | tete :: queue -> tete + base * (lecture_base_aux queue
    ↪ base)
;;
```

```
let lecture_base representation base =
  lecture_base_aux (rev representation) base
;;
```

On peut également préférer une version récursive terminale :

```
let rec lecture_base_aux representation base acc =
  match representation with
  | [] -> acc
  | tete :: queue -> lecture_base_aux queue base (tete +
    ↪ base * acc)
;;
```

```
let lecture_base representation base =
  lecture_base_aux representation base 0
;;
```

La complexité est la même que pour la question précédente.

II Représentation d'ensembles avec des intervalles

II.1) La fonction `premier` est de type `('a * 'b) -> 'a`.

II.2) Les intervalles $[a,b]$ et $[c,d]$ sont disjoints si et seulement si $b < c$ ou $d < a$, puisque $a \leq b$ et $c \leq d$ par hypothèse dans tout le problème. On traduit immédiatement cela en CAML, avec une complexité en $\Theta(1)$.

```
let disjoint (a, b) (c, d) =
  b < c || d < a
;;
```

II.3) La fusion de deux intervalles en est l'union si et seulement si ces deux intervalles ne sont pas disjoints.

II.4) Par définition, la fusion de $[a,b]$ et $[c,d]$ est l'intervalle $[\min(a,c), \max(b,d)]$, ce qui s'écrit directement en CAML, toujours en $\Theta(1)$.

```
let fusion (a, b) (c, d) =
  (min a c, max b d)
;;
```

II.5) Comme $[a,b]$ et $[c,d]$ sont deux intervalles bien formés et disjoints, ce qui est imposé par (a) et (b), on a $c \notin [a,b]$, $a \leq b$ et $c \leq d$ et donc $b < c \iff (a < c) \vee ((a = c) \wedge (b < d))$.

II.6) Il reste à vérifier le point (b). Montrons ce résultat par récurrence sur la longueur n de la liste triée[1] d'intervalles bien formés. Si $n = 0$ ou $n = 1$ la propriété est immédiate. Supposons la propriété vraie au rang $n \geq 1$ et soit une liste $(I_1, \ldots, I_n, I_{n+1})$ triée d'intervalles bien formés de taille $n+1$. Par hypothèse de récurrence les intervalles $\{I_k\}_{1 \leq k \leq n}$ sont deux à deux disjoints. Reste à vérifier que I_{n+1} est disjoint de chacun d'entre eux. La liste étant strictement croissante on a, pour $0 \leq k \leq n$, $I_k < I_{n+1}$, *i.e.* $\max(I_k) < \min(I_{n+1})$, ce qui implique $I_k \cap I_{n+1} = \emptyset$. La propriété est ainsi vraie au rang $n+1$ et le raisonnement par récurrence permet de conclure.

II.7) Si la liste est vide ou si l'élément recherché est « à gauche » de l'intervalle en tête de liste, qui est croissante, la recherche échoue. Si l'élément appartient à l'intervalle en tête (on a déjà vérifié qu'il était supérieur au minimum de cet intervalle) la recherche est réussie. Si l'élément est « à droite » de l'intervalle de tête, on poursuit la recherche dans la queue de la liste.

```
let rec appartenir valeur liste =
  match liste with
   | [] -> false
   | (a, b) :: _ when valeur < a -> false
   | (a, b) :: _ when valeur <= b -> true
   | _ :: queue -> appartenir valeur queue
;;
```

La complexité dans le meilleur des cas, si l'élément appartient à la tête de liste, est en $\Theta(1)$. Dans le pire des cas l'élément n'appartient à aucun intervalle et la complexité est en $\Theta(n)$ où n est la longueur de la liste, entièrement parcourue et au plus une fois.

II.8) Si la liste est vide, on renvoie la liste formée de cet unique intervalle. Si l'intervalle à ajouter n'est pas disjoint de la tête, on fusionne ces deux intervalles (*i.e.* on prend l'union) et on ajoute récursivement ce résultat à la queue. Si au contraire l'intervalle en tête est inférieur à l'intervalle à ajouter, c'est qu'il faut ajouter l'intervalle dans la queue de la liste. Enfin, si l'intervalle en tête est supérieur (seul autre cas possible), il n'y a plus de fusions nécessaires et on peut directement ajouter l'intervalle à cette place.

1. Au sens ici strictement croissante.

```
let rec ajouter intervalle liste =
  match liste with
   | [] -> [intervalle]
   | tete :: queue when not (disjoint tete intervalle) ->
      ajouter (fusion tete intervalle) queue
   | tete :: queue when tete < intervalle ->
      tete :: (ajouter intervalle queue)
   | tete :: queue ->
      intervalle :: tete :: queue
;;
```

II.9) Dans le meilleur cas, l'intervalle à ajouter est inférieur à la tête de la liste et la complexité est en $\Theta(1)$. Comme la fusion de deux intervalles est en $\Theta(1)$ il en va de même, du point de vue de la complexité, qu'il y ait fusion ou que l'on ajoute en queue. Dans le pire des cas, il faut parcourir toute la liste, avec une complexité en $\Theta(n)$ (il en va de même du nombre d'appels récursifs). Ce pire cas est atteint par exemple avec un intervalle à ajouter qui contient tous ceux de la liste ou encore avec un intervalle plus grand que tous ceux de la liste. Encore une fois on parcourt au plus une fois la liste.

II.10) On donne une formulation inductive (récursive) de la propriété « liste bien formée » en utilisant le résultat de la question II.6) qui permet de ne vérifier que le caractère strictement croissant pour montrer (b) :

- La liste vide est bien formée.
- Une liste formée d'un unique intervalle bien formé est une liste bien formée.
- Une liste $(I_1, I_2, \ldots, I_{n+1})$ avec $n \geq 1$ est bien formée si :
  - I_1 est bien formé ;
  - $I_1 < I_2$;
  - La liste $(I_2, \ldots, I_{n+1})$ est bien formée.

Notre programme CAML traduit directement cette formulation :

```
let rec verifier liste =
  match liste with
   | [] -> true
   | (a, b) :: [] -> a <= b
   | (a, b) :: (c, d) :: queue ->
      (a <= b) && (b < c) && (verifier ((c, d) :: queue))
;;
```

La complexité est linéaire, en $\Theta(n)$, et on parcourt au plus (et même exactement) une fois la liste.

III Autour du tri rapide

III.1) La fonction `partition` est de type `'a list -> 'a -> 'a list * 'a * 'a list` et la fonction `tri` de type `'a list -> 'a list`.

III.2) Les étapes du calcul avec les appels récursifs sont détaillées sur la figure 1.

Remarque 1. *On ne connaît pas l'ordre dans lequel `tri petits` et `tri grands` vont être effectués. Cela peut dépendre de la machine, de la version de* CAML*, etc.*

III.3) L'appel `partition liste pivot` partitionne la liste `liste` en deux sous-listes : les éléments plus petits ou égaux à `pivot`, qui sont renvoyés dans la liste `petits` ; et ceux strictement plus grands qui sont renvoyés dans la liste `grands`. Il est immédiat que `milieu` est toujours égal à `pivot` ce que l'on prouvera formellement dans la question suivante. La fonction `tri`, comme son nom l'indique, va trier la liste par ordre croissant. L'appel `tri liste` commence par partitionner la liste, puis trie récursivement les deux sous-listes. Comme les deux sous-listes sont triées, que les éléments de `tri petits` sont inférieurs à `milieu` et que tous ceux-ci sont inférieurs à `tri milieu`, la concaténation renvoyée est bien triée.

III.4) On retrouve bien les trois étapes de l'approche *diviser pour régner* :

- On partitionne le problème initial en sous-problèmes de tailles plus petites, ici deux, de tailles divisées par environ 2. C'est le rôle de la fonction `partition` (diviser) ;
- On résout récursivement les deux sous-problèmes (régner) ;
- On combine le résultat des sous-appels récursifs. Ici, c'est très simple puisque l'on peut directement concaténer dans l'ordre.

III.5) Si $n = 0$, `p` est une séquence vide. On note g de taille r et d de taille s, les deux séquences telles que `(g, v, d) = (partition e p)`. Il s'agit du cas de base de la fonction récursive `partition`, et donc g et d sont vides et $r = s = n = 0$, ce qui montre (c) et (d). On a également, dans le cas de base $v = e$ ce qui donne (a). Les propriétés (b), (e) et (f) sont vérifiées puisque $\forall i \in \emptyset$, $\mathcal{P}(i)$ est toujours vrai quelle que soit la proposition $\mathcal{P}$.

III.6) On a choisi les notations pour que l'hypothèse de récurrence, appliquée ici sur p de taille n et e, s'écrive exactement comme la propriété de l'énoncé :

```
tri <-- [3; 1; 4; 2]
    (* partition queue tete) *)
    partition <-- [1; 4; 2] 3
        partition <-- [4; 2] 3
            partition <-- [2] 3
                partition <-- [] 3
                partition --> ([], 3, [])
            partition --> ([2], 3, [])
        partition --> ([2], 3, [4])
    partition --> ([1; 2], 3, [4])
    (* tri grands *)
    tri <-- [4]
        partition <-- [] 4  (* partition queue tete) *)
        partition --> ([], 4, [])
        tri <-- []  (* tri grands *)
        tri --> []
        tri <-- []  (* tri petits *)
        tri --> []
    tri --> [4]
    (* tri petits *)
    tri <-- [1; 2]
        partition <-- [2] 1  (* partition queue tete) *)
            partition <-- [] 1
            partition --> ([], 1, [])
        partition --> ([], 1, [2])
        tri <-- [2]  (* tri grands *)
            partition <-- [] 2  (* partition queue tete) *)
            partition --> ([], 2, [])
            tri <-- []  (* tri grands *)
            tri --> []
            tri <-- []  (* tri petits *)
            tri --> []
        tri --> [2]
        tri <-- []  (* tri petits *)
        tri --> []
    tri --> [1; 2]
tri --> [1; 2; 3; 4]
```

FIGURE 1 – Étapes du calcul de `tri [3; 1; 4; 2]` pour la question III.2).

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

III.7) On suppose que $p'_{n+1} \leq v$ c'est donc le premier cas de l'expression conditionnelle qui est évalué et donc $g' = (p'_{n+1}, g_r, \ldots, g_1)$ est de taille $r' = r + 1$, $d' = d = (d_s, \ldots, d_1)$ de taille $s' = s$ et $v = v'$.

- Par hypothèse de récurrence $e = v$ donc $e = v'$. On a ainsi (a).
- Soit $i \in [\![1, n+1]\!]$, on veut montrer $|p'|_{p'_i} = |g'|_{p'_i} + |d'|_{p'_i}$. Si $p'_i = p'_{n+1}$, alors $|d'|_{p_{n+1}} = |d|_{p_{n+1}} = 0$ car $p'_{n+1} \leq v$ et le point (f) de l'hypothèse de récurrence impose $\forall j \in [\![1, s]\!],\ v < d_j$. On a donc $|p'|_{p'_{n+1}} = 1 + |p|_{p'_{n+1}} \overset{HR}{=} 1 + |g|_{p'_{n+1}} + |d|_{p'_{n+1}} = 1 + |g|_{p'_{n+1}} = |g'|_{p'_{n+1}} = |g'|_{p'_{n+1}} + |d'|_{p'_{n+1}}$. Si $p'_i \neq p'_{n+1}$ alors $|p'|_{p'_i} = 1 + |p|_{p'_i} \overset{HR}{=} |g|_{p'_i} + |d|_{p'_i} = |g'|_{p'_i} + |d'|_{p'_i}$. Ainsi, on a bien (b).
- On a $0 \leq s' = s < s + 1 \leq n + 1$ et $0 \leq r' = r + 1 \leq n + 1$ et $n + 1 = s + r + 1 = s' + r'$ et donc (c) et (d).
- Soit $i \in [\![1, r']\!]$. Si $i \leq r$, alors $g_i \leq v$ par hypothèse de récurrence. Si $i = r + 1$, alors $g_i = p'_{n+1} \leq v$ ce qui montre (e).
- Le cas (f) de l'hypothèse de récurrence se transmet à l'identique.

On a donc (laborieusement) montré que H_{n+1} était vérifiée, si $p'_{n+1} \leq v$.

III.8) On remarque simplement que le cas $p'_{n+1} > v$ se traite de manière analogue, sans le détailler, et donc H_{n+1} est vérifiée dans tous les cas, ce qui assure l'hérédité. Le raisonnement par récurrence permet donc bien de conclure que la propriété H_n est vraie pour tout $n \in \mathbb{N}$ et que la fonction `partition` se comporte comme nous l'avons expliqué de manière informelle précédemment.

III.9) On note P_m cette propriété et on effectue une récurrence sur l'entier $m \in \mathbb{N}$.

Initialisation : On suppose $m = 0$. On se donne une séquence p de taille m. Soit la séquence r de taille n telle que `r = tri p`. Dans ce cas p est la séquence vide et il en est donc de même pour r et on a directement (a), (b) et (c).

Hérédité : Soit $m \in \mathbb{N}$. On suppose la propriété P_k vérifiée pour tout $0 \leq k \leq m$. Soit une séquence $p' = (p'_{m+1}, p'_m, \ldots, p'_1)$ de taille $m + 1$ et la séquence $q = (q_n, \ldots, q_1)$ telle que `q = tri p'`. L'appel `tri p'` engendre l'appel `partition v p` où $e = p'_{m+1}$ et $p = (p_m, \ldots, p_1) = (p'_m, \ldots, p'_1)$. Le retour de `partition v e` est $g = (g_r, \ldots, g_1)$ de taille r, $d = (d_s, \ldots, d_1)$ de taille s et v. D'après la question précédente, g, d et e vérifient :

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

On peut donc appliquer l'hypothèse de récurrence à `tri g` et `tri d` qui renvoient respectivement les séquences g' et d' de taille s' et r' telles que

(a) $r' = r$ et $s' = s$
(b) $\forall i \in [\![1, r]\!], |g|_{g_i} = |g'|_{g_i}$ et $\forall i \in [\![1, s]\!], |d|_{d_i} = |d'|_{d_i}$
(c) $\forall i \in [\![1, r-1]\!],\ g'_i \leq g'_{i+1}$ et $\forall i \in]\!]1, s-1[\![,\ d'_i \leq d'_{i+1}$.

La séquence `q` est alors $(d'_1, \ldots, d'_s, v, g'_1, \ldots, g'_r)$ de taille $n = r + s + 1$ et on vérifie directement que

(a) $m + 1 = n$
(b) $\forall i \in [\![1, m+1]\!], |q|_{p_i} = |p|_{p_i}$
(c) $\forall i \in [\![1, m]\!],\ q_i \leq q_{i+1}$.

Conclusion : On a montré par récurrence la correction de la fonction `tri` en fonction de la taille de la séquence argument.

III.10) Notons n la taille de la liste reçue en argument pour `partition`. Si $n = 0$, la fonction termine clairement. Si $n \in \mathbb{N} \setminus \{0\}$, il y a un unique appel récursif sur une liste de taille strictement inférieure à n, les autres opérations ne posant pas de problème de terminaison. Ceci montre que `partition` termine sur toute entrée. Le cas de la fonction `tri` est similaire. Si $n = 0$ la terminaison est claire. Si $n \in \mathbb{N} \setminus \{0\}$, il y a un nombre fini d'appels récursifs (deux) sur des listes de tailles strictement inférieures à n et les autres opérations effectuées, en particulier l'appel à `partition`, terminent bien.

III.11) Pour une liste de taille $n \in \mathbb{N}$, il y a exactement n appels récursifs et donc n comparaisons. Ainsi, pour tout $n \in \mathbb{N}$, $P(n) = n$.

III.12) On a immédiatement $T(0) = 0$. Pour une liste de taille 1 le nombre de comparaisons est $P(1-1) = 0$ et donc $T(1) = 0$.

III.13) Si une liste est triée par ordre croissant et que tous les éléments sont distincts, le plus petit élément est choisi comme pivot puisque l'on prend le premier. L'appel `partition queue pivot` découpe la liste en `([], pivot, queue)` puisque tous les éléments sont strictement plus grands que le pivot. Ainsi, les listes restent triées par ordre strictement croissant pour tous les appels récursifs suivants et le pivot est toujours le plus petit élément de la liste. On peut aussi prendre la liste triée par ordre décroissant (non nécessairement strictement).

III.14) Comme l'une des deux listes est vide, on a

$$T(n) = T(n-1) + T(1) + (n-1) = T(n-1) + n - 1$$

III.15) On écrit $T(n) - T(n-1) = n-1$ et par télescopage et avec $T(0) = 0$ on obtient le résultat demandé.

III.16) Notons $C(s)$ le nombre de comparaisons effectuées par `tri` sur une séquence $s = (s_n, \dots, s_1)$, avec $n \geq 1$. Notons $g = (g_p, \dots, g_1)$ et $d = (d_{n-p-1}, \dots, d_1)$ les listes renvoyées par un appel à `partition` sur $(s_{n-1}, \dots, s_1)$ et s_n comme pivot. On a $C(s) = C(g) + C(d) + p + (n-p-1) = C(g) + C(d) + n - 1$. Par définition de la complexité dans le pire des cas $C(g) \leq T(p)$ et $C(d) \leq T(n-p-1)$. Ainsi

$$C(s) \leq T(p) + T(n-p-1) + n - 1 \leq \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1)$$

Ce dernier terme ne dépendant que de la taille de s, on a donc

$$T(n) \leq \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1)$$

Ce maximum étant atteint pour une liste triée décroissante, il y a en fait égalité.

III.17) Pour $n = 1$ on a $T(1) = 0 = \frac{n(n-1)}{2}$. Pour $n \geq 2$, supposons le résultat établi pour $0 \leq k \leq n-1$ et montrons-le au rang n. D'après la question précédente et par hypothèse de récurrence,

$$\begin{aligned} T(n) &= \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1) \\ &= \max_{0 \leq p \leq n-1} \left(\frac{p(p-1)}{2} + \frac{(n-p-1)(n-p-2)}{2} + n - 1 \right) \end{aligned}$$

La formule reste vraie pour $p = 0$. La fonction quadratique $x \mapsto x(x-1) + (n-x-1)(n-x-2)$ sur $[0, n-1]$ atteint son minimum en $\frac{n-1}{2}$ et son maximum à ses deux bornes, qui vaut $(n-1)(n-2)$, ce que l'on montre avec une étude de fonction. On a donc $T(n) = \frac{(n-1)(n-2)}{2} + n - 1 = \frac{n(n-1)}{2}$ et l'hérédité est prouvée. Le raisonnement par récurrence permet de conclure que pour tout $n \in \mathbb{N}^*$ (et même $n \in \mathbb{N}$) on a $T(n) = \frac{n(n-1)}{2}$.

III.18) Soit $n \geq 1$. On a

$$\sum_{k=0}^{n-1} \left(T_{\text{moy}}(k) + T_{\text{moy}}(n-k-1) \right) = 2 \sum_{k=0}^{n-1} T_{\text{moy}}(k)$$

et donc

$$n T_{\text{moy}}(n) = 2 \sum_{k=0}^{n-1} T_{\text{moy}}(k) + n - 1$$

On a ainsi

$$\begin{aligned} n T_{\text{moy}}(n) + (n-1) T_{\text{moy}}(n-1) &= 2 T_{\text{moy}}(n-1) + n(n-1) - (n-1)(n-2) \\ n T_{\text{moy}}(n) + (n+1) T_{\text{moy}}(n-1) &= 2(n-1) \\ \frac{T_{\text{moy}}(n)}{n+1} + \frac{T_{\text{moy}}(n-1)}{n} &= \frac{2(n-1)}{n(n+1)} \end{aligned}$$

III.19) On part de la relation précédente, on somme pour obtenir une série télescopique, et on décompose la fraction rationnelle de droite en éléments simples.

$$\begin{aligned} \sum_{k=1}^{n} \left(\frac{T_{\text{moy}}(k)}{k+1} + \frac{T_{\text{moy}}(k-1)}{k} \right) &= \sum_{k=1}^{n} \frac{2(k-1)}{k(k+1)} \\ \frac{T_{\text{moy}}(k)}{k+1} + \frac{T_{\text{moy}}(0)}{1} &= \sum_{k=1}^{n} \left(\frac{-2}{k} + \frac{4}{k+1} \right) \\ \frac{T_{\text{moy}}(k)}{k+1} &= -2 \sum_{k=1}^{n} \frac{1}{k} + 4 \sum_{k=1}^{n} \frac{1}{k+1} \\ \frac{T_{\text{moy}}(n)}{n+1} &= -2 \sum_{k=1}^{n} \frac{1}{k} + 4 \left(\sum_{k=0}^{n-1} \frac{1}{k+1} + \frac{1}{n+1} - 1 \right) \\ \frac{T_{\text{moy}}(n)}{n+1} &= 2 \sum_{k=1}^{n} \frac{1}{k} + \frac{4}{n+1} - 4 \\ T_{\text{moy}}(n) &= 2(n+1)\Theta(\log n) + 4 - 4(n+1) \\ T_{\text{moy}}(n) &= \Theta(n \log n) \end{aligned}$$