Bug bounty : toute erreur signalée dans le corrigé peut rapporter des points, en fonction de sa gravité. Pour les erreurs les plus mineures (fautes de frappe / français), je ne paye pas à la faute mais à l'octet (arrondi à la valeur supérieure) de fautes.

Solutions des exercices

Solution de l'Exercice 1 – *Best of*

Dans cet exercice, et dans tout ce sujet, on note sans prime (e.g. `k`) la valeur d'une quantité au début d'une itération, et avec prime (e.g. `k'`) la valeur en fin de cette itération (c'est à dire au début de l'itération suivante, si elle existe).

1. (Ceci est une version où je détaille beaucoup les calculs dans la conservation ; vous pouvez aller plus vite.)
La fonction n'a pas d'effets secondaires, comme attendu. Montrons qu'elle renvoie la bonne valeur, c'est à dire qu'elle renvoie $\sum_{i=0}^{n} i$. Il suffit de montrer qu'en sortie de boucle, `somme` vaut cette valeur (puisque c'est celle qui est renvoyée).

Pour prouver cela, montrons que I : «$\mathtt{s} = \sum_{k=0}^{\mathtt{i}-1} k$» est un invariant.

- Initialisation : avant la première itération de la boucle, on a $\mathtt{s} = 0$. On a également $\mathtt{i} - 1 < 0$, d'où la somme de I est vide et vaut donc 0. On a bien $0 = 0$: l'invariant est initialisé.
- Conservation : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que I' : « $\mathtt{s'} = \sum_{k=0}^{\mathtt{i'}-1} k$ ».
  On a :

$$\begin{cases} \mathtt{s'} = \mathtt{s} + \mathtt{i} & \text{(ligne 13)} \\ \mathtt{i'} = \mathtt{i} + 1 & \text{(ligne 14)} \end{cases}$$

  Or d'après I, on a $\mathtt{s} = \sum_{k=0}^{\mathtt{i}-1} k$. Donc :

$$\begin{aligned} \mathtt{s'} &= \mathtt{s} + \mathtt{i} \\ &= \left(\sum_{k=0}^{\mathtt{i}-1} k\right) + \mathtt{i} && \text{d'après } I \\ &= \sum_{k=0}^{\mathtt{i}} k \\ &= \sum_{k=0}^{\mathtt{i'}-1} k && \text{car } \mathtt{i'} = \mathtt{i} + 1 \end{aligned}$$

  On a prouvé que I' est vrai : l'invariant se conserve.
- Conclusion : On a prouvé que I est un invariant. En particulier, comme à la fin de la dernière itération on a $\mathtt{i'} = \mathtt{n} + 1$, on a en sortie de boucle :

$$\begin{aligned} \mathtt{s} &= \sum_{k=0}^{\mathtt{n}+1-1} k \\ &= \sum_{k=0}^{\mathtt{n}} k \end{aligned}$$

Il s'ensuit que :

`somme_entiers` est partiellement correcte.

---

2. (Ici je fais très proprement et longuement les calculs ; j'accepte aussi une version avec beaucoup de $\simeq$)
D'après les données rappelées dans l'énoncé, le type `int` peut stocker les valeurs $[\![-2^{32/2}; 2^{32}/2[\![$ c'est à dire $[\![-2^{31}; 2^{31}[\![$.

Il y aura donc un surpassement de capacité lorsque $\sum_{k=0}^{n} l \geq 2^{31}$, c'est à dire lorsque $\frac{n(n+1)}{2} \geq 2^{31}$, donc quand :

$$n^2 + n - 2^{32} \geq 0$$

Le discriminant de ce polynome est :

$$\Delta = 1 + 2^{34} > 0$$

Ses deux racines sont donc :

$$\frac{-1 - \sqrt{1 + 2^{34}}}{2} \simeq \frac{-1 - 2^{17}}{2} \simeq -2^{16} \text{ et } \frac{-1 + \sqrt{1 + 2^{34}}}{2} \simeq 2^{16}$$

Comme le coefficient dominant du polynôme de degré 2 est positif, le polynôme est négatif entre ses racines et positif en dehors. Donc :

$$\boxed{\text{Pour } n \geq 0, \text{ la variable } \texttt{somme} \text{ débordera à peu près à partir de } n = 2^{16}.}$$

Solution de l'Exercice 2 – *Tours de Hanoï*

1. On utilise les appels récursifs pour « mettre de côté » les k-1 premiers disques afin de pouvoir déplacer le k-ème. On les réutilise ensuite pour déplacer ces k-1 premiers disques sur le k-ème.
Cela donne le pseudo-code suivant :

Algorithme 1 : `HANOI`

Entrées : `depart`, `arrive`, `k`
Sorties : Aucune (déplace des disques)

```
1 si n > 0 alors
2 |   autre ← l'emplacement qui n'est ni depart ni arrivee
3 |   HANOI(depart, autre, k-1)
4 |   DÉPLACE(depart, arrivee)
5 |   HANOI(autre, arrivee, k-1)
```

2. D'après le pseudo-code, on a :

$$\begin{cases} D(0) = 0 \\ D(n) = 1 + 2 * D(n-1) \end{cases}$$

Calculons comme indiqué dans l'énoncé, calculons le point fixe $\ell : \ell = 1 + 2\ell \iff \ell = -1$. Posons la suite auxiliaire $v_n = D(n) - 1$. On a donc $v_0 = 0 - (-1) = 1$ et $v_{n+1} = D(n+1) - 1 = 2D(n) = 2(D(n) - 1 - (-1)) = 2v_n$. Il s'ensuit que $v_n = 2^n$, et donc que :

$$\boxed{D(n) = 2^n - 1}$$

Ainsi, $D(n) = \Theta(2^n)$.

3. Il s'agit du même algorithme que précédemment, mais où on exploite autant que possible la tige penchée :

Algorithme 2 : PISE

Entrées : *depart*, *arrive*, k
Sorties : Aucune (déplace des disques)

```
1  si k > 0 alors
2      autre ←l'emplacement qui n'est ni depart ni arrivee
3      si depart est la tige penchée : alors
           // La tige penchée trivialise ce cas
4          SUPER-DÉPLACE(depart, arrivee, k)
5      sinon si arrive est la tige penchée alors
           // On ne peut pas exploiter la tige penchée
6          PISE(depart, autre, k − 1)
7          DÉPLACE(depart, arrivee)
8          PISE(autre, arrivee, k − 1)
9      sinon
           // autre est la tige penchée : elle évite le second appel rec
10         PISE(depart, autre, k − 1)
11         DÉPLACE(depart, arrivee)
12         SUPER-DÉPLACE(autre, arrivee, k − 1)
```

4. Les seuls appels à DEPLACE ou SUPER-DEPLACE sont dans les branches des Si/Sinon. Notons $A(k)$ la complexité de la première branche, $B(k)$ celle de la seconde et $C(k)$ celle de la troisième. En lisant le code, et en faisant bien attention à quelle tige est la tige penchée lors de chaque appel, on a :

$$\begin{pmatrix} A(k) \\ B(k) \\ C(k) \end{pmatrix} = \begin{pmatrix} 1 \\ C(k-1) + B(k-1) + 1 \\ B(k-1) + 2 \end{pmatrix}$$

Avec comme cas de base, $B(0) = C(0) = 0$.

Ceci est la fin de la question. Mais pour ne pas vous laisser sur votre fin, voici la résolution mathématique détaillée.

$A(k)$ étant déjà calculé, il faut calculer $B(k)$ et $C(k)$. Subsitutions dans l'équation de $B(k)$ à l'aide de l'équation de $C(k)$ (en toute formalité, il faudrait savoir faire une analyse synthèse, soit réécrire à chaque fois toutes les autres équations pour procéder par équivalence. Dans les deux cas, flemme[i].). On obtient :

$$\begin{aligned} B(k) &= (B(k-2) + 2) + B(k-1) + 1 \\ &= B(k-1) + B(k-2) + 3 \end{aligned}$$

Cherchons un point fixe de l'équation de récurrence (une *solution particulière*) : $\ell = \ell + \ell + 3$ a pour unique solution $\ell = -3$. On retranche ensuite cette solution particulière en posant $B'(k) = B(k) - \ell = B(k) + 3$. B' vérifie une l'*équation homogène* :

$$\begin{aligned} B'(k) &= B(k) + 3 \\ &= (B(k-1) + B(k-2) + 3) + 3 \\ &= (B(k-1) + 3) + (B(k-2) + 3) \\ &= B'(k-1) + B'(k-2) \end{aligned}$$

On reconnait une suite récurrente linéaire d'ordre 2 de polynôme caractéristique $X^2 - X - 1 = 0$. Son discriminant est $\Delta = 1 - 1(-1) = 5 > 0$. Les racines de ce polynôme sont :

$$x_{0,1} = \frac{1 \mp \sqrt{5}}{2}$$

Donc $B'(k)$ est de la forme $\lambda x_0^k + \mu x_1^k$ avec $(\lambda, \mu) \in \mathbb{R}^2$ des constantes. Or, $|x_0| < 1$ et $|x_1| > 1$. Comme la complexité ne tend pas vers 0 (si $k > 0$, il faut toujours faire au moins 1 déplacement), on en déduit $\mu \neq 0$ et donc $B'(k) = \Theta(x_1^k)$. Donc :

$$\boxed{B(k) = \Theta(x_1^k)}$$

Et donc :

$$\boxed{C(k) = \Theta(x_1^{k-1}) = \Theta(x_1^k)}$$

Comme le pire des cas est le maximum entre $A(k)$, $B(k)$ et $C(k)$, on a :

$$\boxed{D(k) = \Theta(x_1^k) \text{ où } x_1 \simeq 1,618}$$

Pour trouver la constante cachée par le Θ, finissez les calculs mathématiques en calculant λ et μ, puis déduisez-en $B(k)$ et $C(k)$.

i. Moi j'ai déjà réussi mes concours : je n'ai plus à prouver que je sais le faire.

Solution de l'Exercice 3 – *Compteur binaire*

1. **a.** Lorsque l'on flippe un bit, la propagation de retenue emmène au bit suivant. D'après la gestion des débordements, il n'y a pas de cycle : durant un `INCR`, on ne peut pas « revenir » au début et re-flipper des bits. Ainsi, on flippe au plus L bits. Cette valeur est atteinte lorsque l'on incrémente le compteur dont tous les bits sont à 1.

Donc :

$$\boxed{\text{Un appel à } \texttt{INCR} \text{ flippe au plus } L \text{ bits.}}$$

---

b. Chaque `INCR` flippe au plus L bits, donc :

$$\boxed{K \texttt{ INCR} \text{ flippent au plus } KL \text{ bits}}$$

Cependant, cette borne est trop grossière. En effet, les L bits flippés ne le sont en fait *que* lorsque le compteur est rempli de 1 (ce qui arrive rarement. De plus, en manipulant on se rend compte qu'une opération sur deux coute 1 flip, une sur quatre deux flip, etc : on est presque jamais dans le pire des cas. On doit donc faire de l'analyse amortie.

Remarque. La méthode de l'agrégat est ici la plus confortable ; il suffit de prouver le « on se rend compte ». Cependant, nous n'avions pas encore vu cette méthode lors du DS.

---

2. L'incrémentation fonctionne comme à l'école primaire, par propagation de retenue. Or, lorsque l'on flippe un bit de 0 à 1, cela ne génère pas de retenue : c'est donc la fin de l'incrémentation. Il y a donc au plus 1 flip 0->1.

(En fait, il y en a exactement 1 sauf lors de l'incrémentation qui cause un débordement.)

---

3. Considérons une suite de K appels à `INCR` à partir du compteur à zéro. Numérotons ces appels $1, 2, \dots, K$ et notons c_i le coût réel de l'`INCR` numéro i. On peut décomposer c_i ainsi :

$$c_i = (\text{nombre de flip 0 -> 1}) + (\text{nombre de flip 0 -> 1})$$

(où ces nombres de flips sont les nombres de flips de l'`INCR` numéro i)

On vient de montrer que le premier terme vaut 1. Le second terme, par contre, dépend des propagations de retenue. On veut montrer que son coût se lisse sur les coûts des appels précédents.

Appliquons la méthode du comptable. À chaque appel à `INCR`, on va payer 1 pièce pour le flip 0->1. On pose 1 pièce de plus sur ce bit que l'on flippe à 1. Reste à financer les flips 1->0.

Cependant, comme les bits sont initialement à 0, sur tout bit valant 1 est posée 1 pièce (elle a été posée lorsque ce bit a été flippé de 0->1 pour la dernière fois) : c'est cette pièce que l'on consomme pour payer le flip 1->0 de ce bit. Notons qu'à chaque flip 1->0 on consomme la pièce posée lors du précédent 0->1 : en particulier, on ne consomme pas deux fois la même pièce et tout va bien.

Ainsi, la méthode du comtpable trouve comme coût amorti $\hat{c}_i = 1 + 1 = 2$.

$$\boxed{\text{Au sein d'une suite d'appels à } \texttt{INCR}\text{, chaque appel à } \texttt{INCR} \text{ a un coût amorti constant.}}$$

Solution de l'Exercice 4 – *Écart maximal dans une liste*

1. Voici la fonction demandée :

max-ecart.ml
```
(** Renvoie |x-y| *)
let ecart = fun x y ->
  if x > y then x-y else y-x
```

---

2. Considérons la paire (x, y) d'éléments de E qui maximise $|x-y|$. Quitte à échanger les rôles, supposons que $x \leq y$. On va montrer que $x = \min E$ et $y = \max E$.
Comme $y \leq x$ et $\min E \leq x$:

$$|x-y| = y - x \leq y - \min E = |y - \min E|$$

Or, $|x-y|$ est l'écart maximal. Donc le $\leq$ est en fait un $=$ et donc $x = \min E$.
De même, $y = \max E$. On a donc bien prouvé :

$$|\min(E) - \max(E)| = \max\{|x-y| \; t.q. \; x \in E \; et \; y \in E\}$$

NB : on peut enlever les valeurs absolues en faisant $\max E - \min E$; je les ai juste mises pour voir si vous allez penser à faire cette simplification.

---

3. Le type de sortie de `min` est le type de ses entrées. Or, `trop_facile` renvoie ce que renvoie `min` : si l'on donne des listes à `min`, `min_lst` renverra donc une liste. Ce qui n'est pas le comportement attendu (et le compilateur indique une erreur car `print_int` attend un entier).

Notez que l'erreur est encore plus grosse, car on n'a donné qu'un seul argument à `min`. Comme OCaml fait l'application partielle des fonctions, le terme `min lst` est en fait `fun y -> min lst y` : ce n'est même pas une liste, c'est une fonction !
Donc il y a un *gros* problème de type (aka d'homogénéité).

---

4. On peut utiliser `min t (min_list q)` : le minimum de toute la liste est le plus petit élément entre l'élément de tête et le minimum des autres éléments.
On obtient donc le code suivant :

max-ecart.ml
```
(** Renvoie le minimum de lst *)
let rec min_list = fun lst ->
  match lst with
  | []      -> failwith "min liste vide"
  | t :: [] -> t
  | t :: q  -> min t (min_list q)
```

---

5. Voici la fonction demandée. Notons qu'il n'y a pas besoin de la fonction `ecart`, puisque le maximum est toujours supérieur au minimum :

max-ecart.ml
```
(** Renvoie l'écart maximal dans une liste *)
let ecart_maximal = fun lst ->
  let maxi = max_list lst in
  let mini = min_list lst in
  maxi - mini
```

---

6. L'idée est de parcourir la liste une seule fois, en mémorisant au fur et à mesure le minimum et le maximum des valeurs déjà lues. Quand on arrive à la fin de la liste, on renvoie maximum-minimum.
Voici un programme qui réalise cela (notez que l'unique parcours de liste est réalisé par la sous-fonction) :

max-ecart.ml

```
(** Renvoie l'écart maximal dans une liste en un seul parcours *)
let ecart_max_1_seul_parcours = fun lst ->

  (** Parcourt la liste pour calculer le minimum
    * et le maximum.
    * mini et maxi sont les min/max des valeurs
    * déjà vues. *)
  let rec parcours mini maxi l = match l with
    | [] -> maxi-mini
    | t :: q -> parcours (min t mini) (max t maxi) q
  in
  match lst with
  | [] -> failwith "ecart liste vide"
  | t::q  -> parcours t t q
```

Solution de l'Exercice 5 – ~~*quick*~~*slowselect*

1. **a.** Voici l'implémentation de `length` vue en TP :

slowselect.ml

```
(** length, vue en TP *)
let rec length lst =
  match lst with
  | [] -> 0
  | t :: q -> 1 + length q
```

---

b. Voici une implémentation de `filter` :

slowselect.ml

```
(** filter, avec l'indication de l'énoncé *)
let rec filter f lst =
  match lst with
  | [] -> []
  | t::q -> if f t then t :: (filter f q) else filter f q
```

---

2. Voici les appels successifs (dans leur ordre d'ouverture), ainsi que les valeurs des variables :

i	*lst*	*pivot*	*smaller*	*bigger*	branche
3	`[0; 22; 15; 20; -2; 35; 7; 3; 28]`	0	`[-2]`	`[22; 15; 20; 35; 7; 3; 28]`	C
1	`[22; 15; 20; 35; 7; 3; 28]`	22	`[15; 20; 7; 3]`	`[35; 28]`	B
1	`[15; 20; 7; 3]`	15	`[7; 3]`	`[20]`	B
1	`[7; 3]`	7	`[3]`	`[]`	A

---

3. Cette fonction ne contient ni boucle ni appels à d'autres fonctions. Elle peut par contre effectuer par contre 1 appel récursif. Montrons que la suite d'appels récursifs termine, en prouvant que la longueur de l'argument `lst` est un variant de cette suite d'appels. On notera avec des primes les quantités au début de la prochaine itération.

La longueur de `lst` est :

- Entière par définition.
- Minorée par 0 par définition.
- Strictement décroissante. Pour le prouver, remarquons que l'appel récursif est fait avec `smaller` ou `bigger` jouant le rôle de `lst`. Or, les éléments de ces deux listes sont des éléments de `lst` qui ne sont pas `t` puisque les comparaisons sont stricts. Ainsi, leur longueur est au moins inférieure de 1 à celle de `lst` . D'où la stricte décroissante.

Comme la suite d'appels récursifs admet un variant, elle termine, et donc il s'ensuit :

`slowselect` termine.

4. Prouvons par récurrence forte sur la longueur de *lst* que : « `slowselect` respecte bien ses post-conditions lorsque ses pré-conditions sont respectées. » Je noterai `List.length` *lst* la longueur de *lst*.

Notons que comme *lst* est non-vide d'après les pré-conditions, le cas de base sera les listes-singletons. Notons également qu'il n'y a aucun effet secondaire, ce qui respecte les spécifications : il faut simplement prouver que la sortie est la bonne.

- Initialisation : Si *lst* est réduit à 1 élément, alors on a $0 \leq i <$ `List.length`*lst* et donc $i = 0$.
  On a également *smaller* $= []$, donc `List.length`*smaller* $= 0 = i$. On rentre donc dans la branche A et on renvoie *pivot* (qui est l'unique élément de la liste). C'est bien le 0ème élément de la liste puisque la liste est un singleton.
- Hérédité : soit $n \in \mathbb{N}$, $n > 1$ et supposons que pour toute liste `lst` ayant *strictement* moins de n éléments, `slowselect` respecte bien ses post-conditions lorsque ses pré-conditions sont respectées.
  Prouvons cette propriété vraie au rang n. Soit donc `lst` une liste à n éléments, et considérons un i tel que $0 \leq i <$ `List.length`*lst* . La valeur renvoyée est décidée par un SiSinonSinon (lgn 5-10) : montrons que dans les 3 branches, la valeur renvoyée est la bonne.

  — Branche A (lgn 5-6) : Dans ce cas, on sait qu'il y a exactement i éléments plus petits que le pivot. Il s'ensuit que le pivot est le $i$ème plus petit élément, et c'est bien la valeur que l'on renvoie.

  — Branche B (lgn 7-8) :
  On renvoie dans ce cas `slowselect`(i, *smaller*). Or, $0 \leq i$ et d'après la condition de la branche B $i <$ *card_smaller*, donc cet appel récursif est fait sur une liste plus petite (cf terminaison) et non-vide. Donc par hypothèse de récurrence renvoie le $i$ème plus petit élément de *smaller* .
  Montrons que le $i$ème plus petit élément de *smaller* (que l'on nommera s) est bien le $i$ème plus petit élément de *lst* (que l'on nommera χ).
  Remarquons tout d'abord que $\chi \leq$ *pivot* puisque χ a i éléments qui lui sont strictement inférieurs, et *pivot* en a $> i$ (pour le prouver rigoureusement, on peut faire un raisonnement par l'absurde). En particulier, $\chi \in$ *smaller*.
  De plus, tous les éléments de *lst* strictement inférieurs à χ sont strictement inférieurs à *pivot* et donc dans *smaller*. En particulier, dans *smaller* χ a au moins i éléments qui lui sont strictement inférieurs. Mais comme réciproquement tous ces éléments sont des éléments de *lst*, χ a au plus i éléments strictement inférieurs dans *smaller*. On a donc prouvé que $\chi \in$ *smaller* et qu'il y a exactement i éléments strictements inférieurs dans *smaller*, c'est à dire que $\chi = s$.
  Ainsi, on vient de terminer de prouver que dans la branche B, la valeur renvoyée est la bonne.

  Remarque. En réalité, j'utilise ici sans le dire le caractère total de la relation d'ordre pour conclure. Mais nous n'avons pas encore vu les relations d'ordre en info, donc mes attentes sont modérées en termes de rigueur.

  — Branche C :
  La preuve est similaire à la branche B. J'irai donc plus vite et ne m'attarderai que sur les différences.
  D'après leurs définitions, *smaller*, $[t]$, et *bigger* forment une partition de *lst*. En particulier, le cardinal de *bigger* est $n -$ *card_smaller* $- 1$. Or :

  $$\begin{aligned} n - \textit{card\_smaller} - 1 &> n - i - 1 && \text{car } \textit{card\_smaller} < i \text{(branche C)} \\ &\geq n - i && \text{car la longueur est un entier} > n - n \quad \text{car } i < n \text{(pré-conditions)} \\ &\geq 1 && \text{car la longueur est un entier} \end{aligned}$$

  Donc l'appel récursif est fait sur une liste plus petite et non-vide, et par H.R. renvoie b, la ($i - 1 -$ *card_smaller*)ème valeur de *bigger*.
  Montrons que $b = \chi$. Comme *bigger* $\subset$ *lst*, et comme b est supérieur à i valeurs de *bigger* mais aussi à *pivot* (car $b \in$ *bigger*) (et donc par transitivité aussi aux *card_smaller* valeurs de *smaller*.), b est une valeur de *lst* supérieure à i valeurs de *lst* : c'est bien χ.
  Ainsi, dans le cas de la branche C on renvoie bien la bonne valeur.

  D'où l'hérédité.
- Conclusion : on a prouvé par récurrence forte sur la longueur de *lst* que :

  `slowselect` est partiellement correcte.

5. Voici une traduction en OCaml du pseudo-code. Notons que le cas "de base" de la liste vide ne devrait jamais arriver d'après la preuve de correction.

slowselect.ml

```
29 (** Renvoie le ième plus petit élément de lst (sans doublons) *)
30 let rec slowselect = fun i lst ->
31   match lst with
32   | []   -> failwith "quiselect : empty list"
33   | t::q ->
34       let smaller = List.filter (fun x -> x < t) q in
35       let card_smaller = List.length smaller in
36       if i = card_smaller  then
37         (* t est le ième plus petit *)
38         t
39       else if i < card_smaller then
40         (* t est plus grand que le ième plus petit *)
41         slowselect i smaller
42       else
43         (* t est plus petit que le ième plus petit *)
44         let bigger = List.filter (fun x -> t < x) q in
45         slowselect (i -1 -card_smaller) bigger
```

6. On supposera `List.length` et `List.filter` implémentées comme proposées en début d'exercice (c'est à peu près le cas).

`List.length` ne fait donc aucune comparaison.

`List.filter` en elle-même ne fait pas de comparaison, mais la fonction qu'on lui passe en argument peut en faire. En l'occurence, il s'agit de `(fun x -> x < t)` qui effectue donc 1 comparaison à chaque appel. Or, `filter` appelle cette fonction sur chacun des éléments de la liste : elle effectue donc autant de comparaison qu'il y a d'éléments dans la liste. *On pourrait donner une preuve plus détaillée de cela, mais je m'apprête à faire une preuve détaillée d'un autre point plus difficle, donc...*

Étudions maintenant la complexité de `slowselect` . S'il ne termine pas immédiatement, chaque appel fait deux comparaisons dans ses Si-Sinon (l'égalité et le <), plus les comparaisons de `List.filter` , et ensuite fait un appel récursif. Ainsi, en notant n la longueur de *lst*, le nombre $C(n)$ de comparaisons dans le pire des cas vérifie :

$$C(n) = C(appel\,rec) + n + 2$$

Or, dans le pire des cas, l'appel récursif est fait sur une liste ayant $n-1$ éléments (si *pivot* est le min ou le max de la liste). Ainsi, dans le pire des cas :

$$C(n) = C(n-1) + n + 2$$

Pour $C(1)$, `List.filter` fait 1 comparaison. On teste l'égalité, qui est vraie (cf preuve de correction), et on ne teste donc pas l'inégalité lgn 7. Donc $C(1) = 2 = 1 + 1$. On vérifie par une récurrence simple que :

$$C(n) = \left(\sum_{i=1}^{n} i\right) + \left(\sum_{i=2}^{n} 2\right) + 1$$

Donc :

$$\boxed{C(n) = \frac{n(n+5)}{2} - 1}$$

Remarque.

- J'accepte des calculs fait en $O()$ ou, mieux, en $\Theta()$.
- Pour la médiane en temps linéaire, voir la section 1.8 du livre `http://algorithms.wtf/` (en anglais, mais excellentissime ouvrage).