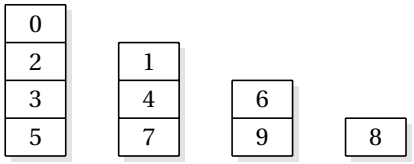
Tri patience – écriture décimale de grands entiers – Corrigé

1 Tri patience

1.1 Jeu de patience

1. On obtient le résultat suivant :



2. Par exemple :

```
let rec add x = function
  | [] -> [[x]]
  | h::t when List.hd h >= x -> (x::h)::t
  | h::t -> h::add x t
```

3. De très nombreuses possibilitées, par exemple :

```
let play lst =
  let rec itere = function
    | [] -> []
    | h::t -> add h (itere t)
  in itere (List.rev lst)
```

```
let play lst =
  let rec itere etat = function
    | [] -> etat
    | h::t -> itere (add h etat) t
  in itere [] lst
```

```
let play lst =
  List.fold_left (fun etat x -> add x etat) [] lst
```

4. Cet invariant est vrai initialement (pas de tas) et après la dépose du premier élément (un seul tas, donc nécessairement trié).

Considérons que l'invariant est vrai à un instant donné et que l'on cherche à déposer x.

- Si l'on dépose x sur un nouveau tas à droite, c'est qu'il est strictement plus grand que tous les sommets de tous les tas, donc en particulier plus grand que le sommet du dernier tas, aussi l'invariant reste vrai ;
- Si l'on dépose x sur un tas existant, comme on cherche à le déposer le plus à gauche possible, c'est que l'élément au sommet du tas immédiatement à sa gauche si un tel tas existe est strictement plus petit ; de même, l'élément du tas immédiatement à sa droite s'il existe était strictement plus grand que l'élément sous x, lui-même supérieur ou égal à x, donc l'invariant reste vrai également.

Les éléments au sommet des tas sont donc rangés dans un ordre croissant *strict*.

5. Il peut n'y avoir à la fin qu'un seul tas (les éléments sont considérés par ordre décroissant) ou jusque n tas (les éléments sont triés par ordre croissant).

6. Le pire cas arrive lorsque x est systématiquement placé sur un nouveau tas tout à droite, car il faut parcourir la liste de tous les tas pour vérifier qu'il est plus grand que chacun des sommets. La complexité est alors quadratique en le nombre n d'éléments, $O(n^2)$.

Le meilleur cas arrive lorsque l'on peut systématiquement poser x sur le premier tas, ce qui donne une complexité linéaire $O(n)$.

1.2 Obtenir un tri

7. On souhaite simplement le sommet du premier tas, donc la tête de la tête de l'argument :

```
let smallest lst =
  List.hd (List.hd lst)
```

On peut également utiliser un filtrage :

```
let smallest = function
  | (h::_)::_ -> h
  | _ -> failwith "cas impossible"
```

8. Il faut bien distinguer deux cas : si le premier tas n'a qu'un élément, on souhaite ne garder que la queue de la listes de listes (car après avoir retiré cet élément, on ne peut pas garder un tas vide). S'il y a au moins deux éléments dans le premier tas, on retire simplement la tête de la première liste. Par exemple :

```
let remove = function
  | (h1::h2::t)::ts -> (h2::t)::ts
  | lst -> List.tl lst
```

9. Il s'agit de la fonction vue en cours, avec une simple adaptation sur la comparaison (on compare les têtes des éléments) :

```
let rec insert stack lst = match lst with
  | [] -> [stack]
  | h::t when List.hd h < List.hd stack -> h::insert stack t
  | lst -> stack::lst
```

10. Reste à mettre tous les éléments en place. On commence par construire les tas avec les règles de la patience, puis tant qu'il reste des tas, on prend le plus petit élément que l'on place en tête, on le retire, et s'il reste des éléments, on insère le premier tas dans le reste (même si ce n'est pas le tas d'où vient l'élément retiré, ça n'a pas d'importance), et on poursuit. Par exemple :

```
let sort lst =
  let stacks = play lst in
  let rec reconstruct = function
    | [] -> []
    | lst -> smallest lst :: (match remove lst with
                              | h::t -> reconstruct (insert h t)
                              | [] -> [])
  in reconstruct stacks
```

11. Après une première partie de complexité quadratique dans le pire cas, l'obtention d'un plus petit élément est en $O(1)$, son retrait en $O(1)$, l'insertion d'un tas dans une liste de tas sera en $O(n)$. Comme il y a n éléments à retirer, dans le pire des cas, la seconde partie de l'algorithme a un coût quadratique ($O(n^2)$), comme la première. La fonction `sort` a donc une complexité quadratique dans le pire cas.

12. Le tri n'est pas stable. En effet, si l'on a par exemple deux éléments égaux qui apparaissent consécutivement dans la liste, la patience placera le second au-dessus du premier. Lorsque l'on extraiera les éléments, le second élément ressortira donc avant le premier, et sera placé avant lui dans la liste triée.

1.3 Sous-séquences strictement croissantes

13. La longueur des plus longues sous-séquences strictement croissantes est égale au nombre de tas. Pour le montrer, on va justifier qu'à tout instant, la longueur de la plus longue sous-séquence croissante dans les k premiers éléments (ceux déjà posés) est égale au nombre de tas à cet instant.

Initialement, lorsqu'aucun élément n'a été posé, c'est vrai (sous-séquence de longueur nulle). De même, après la pose du premier élément, la plus longue sous-séquence croissante est réduite à ce seul élément.

(fin de la preuve en cours de clarification)

14. Puisque l'on a obtenu quatre tas avec la patience, les plus longues sous-séquences strictement croissantes sont de longueur 4. Il n'en existe en fait qu'une, $[3;4;6;8]$.

15. Pour construire une sous-séquence de longueur maximale, on prend pour élément y_{p-1} le plus bas de la dernière pile, puis, parmi les éléments de l'avant dernière pile le plus grand des éléments strictement inférieurs à y_{p-1} (on a la garantie qu'il a été posé avant y_p, car s'il n'était pas encore posé, y_p n'aurait pas été posé dans la dernière pile), et ainsi de suite de droite à gauche.

16. On commence par définir deux fonctions, l'une donnant le dernier élément d'une liste, l'autre l'élément le plus loin dans une liste croissante strictement inférieur à son premier argument :

```
let rec last = function
  | [] -> failwith "vide"
  | [h] -> h
  | _::t -> last t

let rec last_under v = function
  | h1::h2::t when h2 < v -> last_under v (h2::t)
  | lst -> List.hd lst
```

Ceci fait, on peut appliquer l'idée de la question précédente :

```
let rec long_incr_subseq lst =
  let stacks = play lst in
  let rec reconstruct = function
    | [] -> []
    | [h] -> [last h]
    | h::t -> let lst = reconstruct t in
              last_under (List.hd lst) h :: lst
  in reconstruct stacks
```

2 Grands entiers

2.1 Introduction

1. On divise n par d tant que d est un diviseur de n, en comptant les itérations. Sous la forme d'une fonction récursive, cela donne par exemple :

```
let rec exponent d n =
  if n mod d <> 0 then 0
                  else 1 + exponent d (n/d)
```

2. `foo` prend un entier `d` puis, au travers de `fonction`, un second entier `n`, et retourne un entier. On a donc une fonction de signature `int -> int -> int`.

3. La fonction `foo d n` compte les i tels que définis précédemment pour tous les entiers de 1 à n. Pour connaître le nombre de zéros à droite dans l'écriture décimale de $n!$, il faut et suffit de savoir combien de fois on peut diviser $n!$ par 5 (car la multiplicité de 2 dans $n!$ est forcément supérieure à celle de 5, et le seul moyen d'avoir un facteur 10, causant un zéro à droite dans l'écriture décimale, est 2×5).

`foo 5 n` donne donc très exactement le nombre de zéros recherché, donc simplement :

```
let nb_zeros n = foo 5 n
```

4. On peut déterminer la multiplicité de 5 dans $n!$ plus rapidement : elle correspond à

$$\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{5^2} \right\rfloor + \left\lfloor \frac{n}{5^3} \right\rfloor + \dots$$

On peut donc écrire

```
let nb_zeros n =
  let rec itere k =
    if k>n then 0
           else n/k + itere (5*k)
  in itere 5
```

2.2 Grands entiers

5. C'est simplement une conversion d'un entier n en base m, en représentant ses chiffres sous forme d'une liste. On retrouve donc une fonction similaire à celles vues en cours :

```
let rec convert n =
  if n = 0 then [] else
  if n < m then [n] else
  n mod m :: convert (n/m)
```

6. Une fois le cas de 0 traité, on envisage simplement d'incrémenter la tête de la liste. Mais elle doit rester strictement inférieure à m, donc si après l'incrémentation on atteint m, on place un 0 en tête et on incrémente la suite. Il s'agit simplement du principe de la retenue !

```
let rec inc = function
  | [] -> [1]
  | h::t when h = m-1 -> 0::inc t
  | h::t -> h+1::t
```

7. Même chose, on pose l'addition comme vu en primaire : on additionne les chiffres de poids faible, et si leur somme dépasse m, on utilise une retenue (et on peut utiliser `inc` pour cela) :

```
let rec add a b = match a, b with
  | [], _ -> b
  | _, [] -> a
  | ha::ta, hb::tb -> let s = ha + hb in
                      if s < m then s :: add ta tb
                               else s mod m::inc (add ta tb)
```

8. On cherche à décrémenter la tête. Si elle atteint 0, on décrémentera récursivement la queue. Mais attention, dec présente une difficulté supplémentaire : si la tête atteint 0, on doit impérativement vérifier que la queue n'est pas vide : en effet, `dec [1]` doit retourner `[]` et non `[0]` ! Il y a donc pas mal de cas à considérer :

```
let rec dec = function
  | [] -> failwith "résultat négatif !"
  | [1] -> []
  | 0::t -> m-1::dec t
  | h::t -> h-1::t
```

9. Même chose, on implémente l'algorithme du primaire, en utilisant `dec` pour les éventuelles retenues. Là aussi, si les deux chiffres de poids faible sont égaux, il y a des risques de se retrouver avec des 0 à l'extrémité droite de la liste, ce qui est interdit. On peut par exemple écrire :

```
let rec sub a b = match a, b with
  | _, [] -> a
  | [], _ -> failwith "négatif sub"
  | ha::ta, hb::tb when ha > hb -> ha-hb::sub ta tb
  | ha::ta, hb::tb when ha = hb
     -> (match sub ta tb with | [] -> []     (* Attention ! *)
                              | t -> 0::t)
  | ha::ta, hb::tb -> m+ha-hb::sub (dec ta) tb
```

10. La solution la plus simple consiste à remarquer que si une liste est plus longue, elle représente un entier strictement supérieur, et si elles sont de même taille, on peut

comparer les éléments *de la droite vers la gauche*. Par exemple :

```
let cmp a b =
  List.length a > List.length b ||
  let rec cmp_aux = function
    | ha::ta, hb::tb -> ha > hb || cmp_aux (ta, tb)
    | _ -> true (* égalité *)
  in cmp_aux (List.rev a, List.rev b)
```

On peut cependant quand même traiter les listes sans les retourner, mais c'est un peu plus difficile. On peut écrire une fonction qui compare deux listes et retourne 1 si la première est strictement plus grande, 0 si elles sont égales, et -1 si elle est strictement plus petite :

```
let rec cmp_i a b = match a, b with
  | [], [] -> 0
  | _, [] -> 1
  | [], _ -> -1
  | ha::ta, hb::tb -> match cmp_i ta tb with
                      | 0 -> if ha>hb then 1 else if ha<hb then -1 else 0
                      | k -> k
```

On a alors simplement

```
let cmp a b = cmp_i a b >= 0
```

2.3 Multiplications

11. On a $q_0 \in [\![0\,..\,(m^c-1)^2]\!]$ (et généralement $q_0 > 0$ sinon l'algorithme n'est pas intéressant), $q_1 \in [\![0\,..\,2(m^c-1)^2]\!]$ et $q_2 \in [\![0\,..\,(m^c-1)^2]\!]$.

12. On a $q_1 = \big(n_1p_0 + p_1n_0\big) = q_k - n_1p_1 - n_0p_0 = q_k - q_2 - q_0$.

13. Si p est la longueur de la liste représentant p, alors le nombre n est strictement inférieur à m^p (mais pas à m^{p-1}). Il suffit donc de prendre le plus petit entier c tel que $2c \geqslant p$, soit $\lceil p/2 \rceil = \lfloor (p+1)/2 \rfloor$.

14. On souhaite avoir un couple contenant les c premiers éléments de la liste, et les éléments restants (qui sont au plus c). Attention, à nouveau, à la condition qui impose que 0 est représenté par `[]` et qu'il ne doit pas y avoir de 0 à l'extrémité droite de la liste. Par exemple :

```
let rec split c n =
  if c = 0 then [], n else
  match n with | [] -> [], []
               | 0::t -> let lower, upper = split (c-1) t in
                         (match lower with | [] -> [], upper
                                           | t -> 0::t, upper)
               | h::t -> let lower, upper = split (c-1) t in
                         h::lower, upper;;
```

15. Il suffit de rajouter c zéros à gauche de la liste... sous réserve que l'argument n n'est pas nul! Par exemple :

```
let rec pow c n =
  if n = [] then []
            else if c = 0 then n
                          else 0 :: pow (c-1) n
```

16. L'essentiel du travail est fait, reste à implémenter le processus décrit :

```
let rec mul n p = match n, p with
  | [], _ -> []
  | _, [] -> []
  | [hn], [hp] -> convert (hn*hp)
  | _ -> let c = (max (List.length n) (List.length p) + 1) / 2 in
         let n0, n1 = split c n
         and p0, p1 = split c p in
         let q2 = mul n1 p1
         and q0 = mul n0 p0 in
         let qk = mul (add n0 n1) (add p0 p1) in
         let q1 = sub (sub qk q0) q2 in
         add (pow c (add (pow c q2) q1)) q0
```

2.4 Retour à notre problème

17. En fait, la fonction est très proche de la fonction factorielle vue en cours, si ce n'est qu'on utilise `mul` et `convert` pour gérer les grands entiers naturels :

```
let rec fact = function
  | 0 -> [1]
  | n -> mul (convert n) (fact (n-1))
```

18. Comme vu en travaux dirigé, on peut aisément compter le nombre d'apparitions d'un chiffre récursivement :

```
let rec count7_int = function
  | 0 -> 0
  | n when n mod 10 = 7 -> 1 + count7_int (n/10)
  | n -> count7_int (n/10)
```

19. On appelle la fonction précédente sur chaque élément de la liste représentant le grand entier naturel, et on somme :

```
let rec count7_big_nat = function
  | [] -> 0
  | h::t -> count7_int h + count7_big_nat t
```

Ou bien pour les amateurs de fonctionnelles :

```
let count7_big_nat lst =
  List.fold_left (+) 0 (List.map count7_int lst)
```