TD XXXV

PLUS LONGUE SOUS-SÉQUENCE CROISSANTE

Définitions et notations

Dans tout le sujet, on s'intéresse à des séquences $s = s_0, \ldots, s_{n-1}$ d'entiers. Ces séquences peuvent être représentées en Caml soit par des listes (type `int list`), soit par des tableaux (type `int array`).

Les fonctions que l'on vous demande d'écrire accepteront en fait des types plus généraux (`'a list` et `'a array`), car les seules opérations que l'on fera sur les éléments sont des comparaisons, qui sont polymorphes en OCaml. Il n'y a pas de problème : si l'énoncé vous demande une fonction `f : int list -> bool` et que vous écrivez une fonction `f : 'a list -> bool` qui a le comportement attendu quand elle est appelée sur une liste d'entiers, vous avez évidemment répondu à la question.

Étant donnée une séquence $s = s_0, \ldots, s_{n-1}$:

- la *longueur* de s, notée $|s|$, est son nombre d'éléments n ;
- s est dite croissante si $s_0 \leqslant s_1 \leqslant \ldots \leqslant s_{n-1}$;
- une *sous-séquence* de longueur k de s est une séquence $s_{\varphi(0)}, \ldots, s_{\varphi(k-1)}$ où φ est strictement croissante. Si $k = 0$, la sous-séquence est vide.
  Par exemple, $u = 7, 2, 8$ est une sous-séquence de $s = 7, 1, 2, 6, 4, 5, 8$ (avec $\varphi(0) = 0$, $\varphi(1) = 2$ et $\varphi(3) = 6$). En revanche, ni $v = 2, 8, 1$ ni $w = 7, 7, 6$ ne sont des sous-séquences de s.
- on notera $l_{seq}(s)$ la longueur maximale d'une sous-séquence croissante de s. Autrement dit :

$$l_{seq}(s) \overset{\text{déf.}}{=} \max(|u|,\ u \text{ sous-séquence de } s \text{ et } u \text{ croissante})$$

Ce problème porte sur le calcul efficace de l_{seq} ainsi que sur l'extraction d'une sous-séquence croissante maximale. Pour $s = 7, 1, 2, 6, 4, 5, 8$, on a $l_{seq}(s) = 5$ réalisé pour $u = 1, 2, 4, 5, 8$:

1 Méthode par énumération

Dans cette partie, on représente les séquences par des listes d'entiers.

► **Question 1** Exprimer en fonction de $|s|$ le nombre de sous-séquences de s (en supposant que les éléments de s sont deux à deux distincts).

► **Question 2** Écrire une fonction `est_croissante : int list -> bool` qui renvoie `true` si son argument est une liste croissante, `false` sinon. La liste vide sera considérée comme croissante.

► **Question 3** Écrire une fonction `prefixe : 'a -> 'a list list -> 'a list list` telle que `prefixe x [u_1 ; ... ; u_n]` renvoie `[x :: u_1 ; ... ; x :: u_n]`.

► **Question 4** Écrire une fonction `sous_sequences : (s : 'a list) : 'a list list` qui renvoie la liste de toutes les sous-séquences de s. On doit donc avoir, à l'ordre près :
`sous_sequences [8; 3; 5] = [[8; 3; 5]; [8; 3]; [8; 5]; [8]; [3; 5]; [3]; [5]; []]`.

► **Question 5** Écrire une fonction `l_seq_naif (s : int list) : int` qui renvoie $l_{seq}(s)$. On procédera de manière brutale en générant toutes les sous-séquences de s.

► **Question 6** Quelle est la complexité de `l_seq_naif` (en fonction de $|s|$) ?

2 Méthode par programmation dynamique

Dans cette partie, on représente une séquence $s = s_0, \ldots, s_{n-1}$ par un `s : int array` de taille n. On associe à s un tableau `longueurs` de taille n tel que `longueurs.(k)` soit la longueur de la plus longue sous-séquence croissante de la forme $s_{\varphi(0)}, \ldots, s_{\varphi(i)}$ avec $\varphi(i) = k$ (autrement dit, la longueur de la plus grande sous-séquence croissante de s se terminant exactement en s_k). On peut par exemple avoir :

s	10	12	2	8	3	11	7	14	9	4
longeurs	1	2	1	2	2	3	3	4	4	3

▸ **Question 7** Montrer que, pour $0 \leqslant k < n-1$, on a :

$$\texttt{longueurs.(k + 1)} = \begin{cases} 1 & \text{si } s_{k+1} < \min(s_0, \ldots, s_k) \\ 1 + \max\{\texttt{longueurs.(i)} \mid 0 \leqslant i \leqslant k \text{ et } s_i \leqslant s_{k+1}\} & \text{sinon} \end{cases}$$

▸ **Question 8** En déduire une fonction `aux_dyn (s : int array) : int array` qui renvoie le tableau `longueurs` associé à s. On demande une complexité en $O\left(|s|^2\right)$, que l'on justifiera.

▸ **Question 9** Écrire alors une fonction `l_seq_dyn (s : int array) : int` calculant $l_{seq}(s)$ et préciser sa complexité.

▸ **Question 10** Écrire une fonction `sous_sequence_dyn (s : int array) : int array` qui renvoie une sous-séquence croissante de s de longueur maximale, et préciser sa complexité.
On reconstruira la séquence à partir du tableau `longueurs`.

3 Méthode de la *patience*

L'algorithme présenté dans cette partie est inspiré d'un jeu de cartes (d'une réussite, plus précisément) appelé *patience*, dont le principe est exposé ci-dessous.

On dispose d'un paquet de n cartes numérotées (les numéros sont des entiers, plusieurs cartes peuvent éventuellement porter le même numéro). On prend les cartes une par une, depuis le sommet du paquet, et l'on doit les organiser en piles en respectant les règles suivantes :

- au début du jeu, il n'y a aucune pile ;
- quand on tire une nouvelle carte, on peut :
  - soit la mettre sur une nouvelle pile (que l'on placera à droite de toutes les piles existantes) ;
  - soit la rajouter au sommet d'une pile existante, à condition qu'elle soit strictement plus petite que la carte actuellement au sommet de cette pile.

Par exemple, supposons que l'on tire une carte $\boxed{17}$ et que l'on soit dans la configuration suivante :

p_1	p_2	p_3	p_4
18		20	
20	17	25	
23	19	28	13

On peut rajouter le $\boxed{17}$ au sommet de p_1, au sommet de p_3 ou au sommet d'une nouvelle pile p_5, mais pas au sommet de p_2 ni de p_4.

Le but du jeu est de terminer le paquet en utilisant le moins de piles possibles. Pour ce faire, on utilise la stratégie *gloutonne*.

Stratégie gloutonne : à chaque fois qu'on tire une nouvelle carte, on la place le plus à gauche possible.

Dans l'exemple ci-dessus, on ajouterait donc le $\boxed{17}$ au sommet de la pile p_1.

▶ **Question 11** Vérifier qu'en utilisant la stratégie gloutonne, on a la transformation suivante (du paquet initial vers l'état en fin de partie) :

$$\left|\begin{array}{c}5\\2\\8\\9\\3\\2\end{array}\right| \longrightarrow \underset{p_1}{\left|\begin{array}{c}2\\5\end{array}\right|} \quad \underset{p_2}{\left|\begin{array}{c}2\\3\\8\end{array}\right|} \quad \underset{p_3}{\left|\begin{array}{c}9\end{array}\right|}$$

▶ **Question 12** Montrer que si l'on suit la stratégie gloutonne, alors après chaque étape les sommets des piles, lus de gauche à droite, forment une suite croissante.

On note $s = a_0, \dots, a_{n-1}$ la séquence correspondant à un paquet (a_0 est la première carte tirée).

▶ **Question 13** Montrer que quelle que soit la stratégie utilisée, on utilise au moins $l_{seq}(s)$ piles.

▶ **Question 14** Montrer que la stratégie gloutonne utilise exactement $l_{seq}(s)$ piles (et est donc optimale).

Étant donnée une séquence s, on peut donc déterminer $l_{seq}(s)$ en « jouant une partie » avec le paquet s en suivant la stratégie gloutonne et en comptant le nombre de piles utilisées.
La séquence s est donnée sous forme de liste, et l'on choisit d'utiliser le type suivant pour représenter les configurations :

```
type config = int list array
```

▶ **Question 15** Écrire une fonction `patience : int list -> config` qui prend en argument un paquet (sous la forme d'une liste) et renvoie la configuration obtenue à la fin du jeu en suivant la stratégie gloutonne. On doit donc avoir `patience [5; 2; 8; 9; 3; 2] = [|[2; 5]; [2; 3; 8]; [9]; []; []; []|]`
On demande une complexité en $O\left(|s|^2\right)$ dans le pire des cas, ce que l'on justifiera.

▶ **Question 16** Comment pourrait-on utiliser le résultat de la question 12 pour abaisser la complexité de la fonction `patience` ?
À cette question, on demande seulement de donner l'idée de l'algorithme.

▶ **Question 17** Écrire une fonction `patience_opt : int list -> config` ayant la même spécification que `patience` mais une complexité en $O(|s| \cdot \log |s|)$, que l'on justifiera.

▶ **Question 18** Écrire une fonction `l_seq_patience (s : int list) : int` qui calcule $l_{seq}(s)$ en temps $O(|s| \cdot \log |s|)$.

▶ **Question 19** Expliquer comment modifier la manière de stocker les configurations pour pouvoir reconstruire à la fin du calcul une sous-séquence croissante de longueur maximale.

▶ **Question 20** Écrire une fonction `sous_sequence_patience (s : int list) : int list` qui renvoie une sous-séquence croissante de s de longueur maximale en temps $O(|s| \ln |s|)$.

4 Bonus

▶ **Question 21** Démontrer le théorème suivant :

Théorème — Erdös-Szekeres

Toute séquence de $n^2 + 1$ entiers contient une sous-séquence monotone de longueur $n + 1$.

Solutions

▸ **Question 1** Choisir une sous-séquence de s, c'est choisir une partie de $[\![0, |s|-1]\!]$: il y a $\boxed{2^{|s|} \text{ sous-séquences.}}$

▸ **Question 2**

```
let rec est_croissante u =
  match u with
  | [] | [_] -> true
  | a :: b :: xs -> a <= b && est_croissante (b :: xs)
```

▸ **Question 3**

```
let prefixe x u =
  List.map (fun v -> x :: v) u
```

Il serait plus idiomatique d'écrire `let prefixe x = List.map (fun v -> x :: v)`.

▸ **Question 4** On a $\mathcal{P}(\{x\} \cup A) = \mathcal{P}(A) \cup \left\{\{x\} \cup B, B \in \mathcal{P}(A)\right\}$, et l'union est disjointe si $x \notin A$.

```
let rec sous_sequences u =
  match u with
  | [] -> [[]]
  | x :: xs ->
      let sans_x = sous_sequences xs in
      let avec_x = prefixe x sans_x in
      avec_x @ sans_x
```

▸ **Question 5**

```
(* max_croissante : 'a list list -> int
   max_croissante [u_1; ...; u_n] renvoie le max des
   longueurs de u_k pour u_k croissante *)
let rec max_croissantes u =
  match u with
  | [] -> 0
  | x :: xs when est_croissante x -> max (List.length x) (max_croissantes xs)
  | x :: xs -> max_croissantes xs

(* l_seq_naif : 'a list -> 'a list *)
let l_seq_naif u =
  max_croissantes (sous_sequences u)
```

▸ **Question 6**

- `max_croissantes [u_1; ... ; u_n]` effectue un parcours de chacune des listes $u_1, \ldots, u_n$ et a donc une complexité en $O\left(\sum_{i=1}^{n} |u_i|\right)$.
- Quand on l'appelle sur toutes les sous-séquences de u, on obtient donc (sachant qu'il y a $\binom{|s|}{k}$ sous-séquences de longueur k) :

$$O\left(\sum_{k=0}^{|s|} k\binom{|s|}{k}\right) = O\left(\sum_{k=0}^{|s|} |s|\binom{|s|-1}{k-1}\right) = O\left(|s| \cdot 2^{|s|-1}\right) = O\left(|s| \cdot 2^{|s|}\right)$$

- Il reste à prendre en compte le temps de construction des sous-séquences, mais il est clairement dominé par $O\left(|s| \cdot 2^{|s|}\right)$.
  Remarque
  Montrer que ce coût est en fait un $O(2^{|s|})$ est un bon exercice de calcul de complexité. Il faut appliquer le même genre de technique que pour les algorithmes « diviser pour régner ».

La complexité de `l_seq_naif` est donc en $O\left(|s| \cdot 2^{|s|}\right)$.

▶ **Question 7** Soit $0 \leqslant k \leqslant n-2$.

- Si $s_{k+1} < \min(s_0, \dots, s_k)$, alors la seule sous-séquence croissante se terminant en s_{k+1} est celle réduite à s_{k+1}, qui est de longueur 1.
- Sinon, il y a au moins un i vérifiant $0 \leqslant i \leqslant k$ et $s_i \leqslant s_{k+1}$.
  — Pour chacun de ces i, on peut obtenir une sous-séquence croissante de taille $1 + \text{longueur}[i]$ se terminant en s_{k+1} en prenant une sous-séquence maximale se terminant en s_i et en lui rajoutant s_{k+1}. On a donc $\text{longueur}[k+1] \geqslant 1 + \max\{\text{longueur}[i],\ 0 \leqslant i \leqslant k \text{ et } s_i \leqslant s_{k+1}\} (\geqslant 2)$.
  — Inversement, toute sous-séquence croissante de longueur supérieure ou égale à 2 se terminant en s_{k+1} est de la forme $\dots, s_i, s_{k+1}$ pour un certain i vérifiant $0 \leqslant i \leqslant k$ et $s_i \leqslant s_{k+1}$. La sous-séquence $\dots, s_i$ est croissante, donc de longueur au plus $\text{longueur}[i]$, et la séquence $\dots, s_{k+1}$ est donc de longueur au plus $1 + \text{longueur}[i]$. On a donc l'autre inégalité.

On a donc bien

$$\boxed{\text{longueur}[k+1] = \begin{cases} 1 & \text{si } s_{k+1} < \min(s_0, \dots, s_k) \\ 1 + \max\{\text{longueur}[i] \mid 0 \leqslant i \leqslant k \text{ et } s_i \leqslant s_{k+1}\} & \text{sinon} \end{cases}}$$

▶ **Question 8** Traduction immédiate de la récurrence :

```
(* aux_dyn : 'a array -> int array
   A l'étape i, longueur.(k) contient la longueur de la plus
   longue sous-suite croissante se terminant en k (ou 0 si k > i) *)
let aux_dyn t =
  let n = Array.length t in
  let longueur = Array.create n 0 in
  for i = 0 to n - 1 do
    let maxi = ref 1 in
    for k = 0 to i - 1 do
      if t.(k) <= t.(i) then maxi := max !maxi (longueur.(k) + 1)
    done;
    longueur.(i) <- !maxi
  done;
  longueur
```

Le corps de la boucle interne est en temps constant, donc la boucle interne est en $O(i)$ et la boucle externe en $O\left(\sum_{i=0}^{n-1} i\right) = O\left(n^2\right)$.

L'initialisation est en $O(n)$, on a bien une complexité totale en $O\left(n^2\right)$.

▶ **Question 9** Il suffit de prendre le maximum du tableau `longueur`. La fonction `ind_et_max` servira réellement à la question suivante.

```
(* indice d'une occurrence du maximum et valeur de ce maximum *)
let ind_et_max t =
  let ind_maxi = ref 0 in
  for i = 1 to Array.length t - 1 do
    if t.(i) > t.(!ind_maxi) then ind_maxi := i
  done;
  !ind_maxi, t.(!ind_maxi)

(* l_seq_dyn : 'a array -> int *)
let l_seq_dyn t = snd (ind_et_max (aux_dyn t))
```

► **Question 10**

```
(* On détermine à quel endroit se termine la plus grande sous-suite croissante
   (l'une des, en fait), ainsi que sa longueur. On parcourt le tableau vers la
   gauche à partir de ce point, en prenant à chaque fois un élément pour
   lequel la longueur est égale au nombre d'éléments restant à choisir et la
   valeur est inférieure à celle du dernier élément choisi. *)
let sous_sequence_dyn t =
  let tab_longueur = aux_dyn t in
  let ind_dernier, longueur = ind_et_max tab_longueur in
  let sous_suite = Array.create longueur t.(ind_dernier) in
  let k = ref (ind_dernier - 1) in
  let a_choisir = ref (longueur - 1) in
  while !a_choisir > 0 do
    if tab_longueur.(!k) = !a_choisir && t.(!k) <= sous_suite.(!a_choisir) then
      begin
        a_choisir := !a_choisir - 1;
        sous_suite.(!a_choisir) <- t.(!k)
      end;
    decr k (* équivaut à k := !k - 1 *)
  done;
  sous_suite
```

La reconstruction rajoute simplement un parcours du tableau en $O(n)$, $\boxed{\text{la complexité reste donc en } O(n^2)}$.

► **Question 11** On le vérifie. . .

► **Question 12** C'est bien sûr vrai au départ (suite vide), supposons que ce soit vrai au moment de tirer une carte x et notons $s_1 \leqslant \ldots \leqslant s_p$ les sommets.

- Si $x \geqslant s_p$, on rajoute une pile et l'on obtient $s_1 \leqslant \ldots \leqslant s_p \leqslant s_{p+1} = x$.
- Si $x < s_1$, on place x sur la première pile et obtient $s_1' = x < s_2 \leqslant \ldots \leqslant s_p$.
- Sinon, on trouve l'unique i tel que $s_i \leqslant x < s_{i+1}$ et l'on place x sur la pile $i+1$. On obtient alors $s_1 \leqslant \ldots \leqslant s_i \leqslant s_{i+1}' = x < s_{i+2} \leqslant \ldots$

Par récurrence, $\boxed{\text{la suite des sommets est croissante à tout instant.}}$

► **Question 13** Remarquons d'abord que les règles imposent que chaque pile, lue de bas en haut, soit strictement décroissante. Comme les cartes sont empilées dans l'ordre de tirage, cela implique que si $i < j$ et $a_i \leqslant a_j$, les deux cartes ne peuvent être dans la même pile. Ainsi, les cartes d'une sous-séquence croissante sont nécessairement rangées dans des piles deux à deux distinctes. En considérant une sous-séquence croissante de longueur maximale, $\boxed{\text{on en déduit qu'il faut au moins } l_{seq}(s) \text{ piles.}}$

► **Question 14** Imaginons qu'à chaque fois que l'on place une carte sur une pile, on rajoute un pointeur de cette carte vers le sommet de la pile située immédiatement à gauche de cette pile (si la carte est placée sur la première pile, disons qu'on ne crée pas de pointeur).

- Toute carte (sauf celles de la première pile) pointe vers une autre carte.
- Si a_i pointe vers a_j, alors :
  — $i > j$ (on pointe vers une carte déjà placée) ;
  — $a_i \geqslant a_j$ (si on avait pu placer a_i au-dessus de a_j, on l'aurait fait puisqu'on utilise la stratégie gloutonne).

Si l'on prend l'une des cartes situées sur la pile la plus à droite et que l'on suit les pointeurs jusqu'à arriver à la pile la plus à gauche, on obtient donc une sous-séquence croissante de longueur égale au nombre de piles.
Par maximalité de $l_{seq}(s)$, on a donc au plus $l_{seq}(s)$ piles pour la stratégie gloutonne, et en combinant avec la question précédente, $\boxed{\text{la stratégie gloutonne utilise exactement } l_{seq}(s) \text{ piles.}}$

► **Question 15** Il n'y a pas de difficulté particulière. Pour chaque carte, on parcourt les sommets de pile en cherchant le premier que l'on puisse utiliser puis l'on place la carte. Cela prend au pire un temps proportionnel au nombre de piles à cet instant, qui est majoré par $|s|$.
$\boxed{\text{Il y a } |s| \text{ cartes, la complexité est donc bien un } O\left(|s|^2\right).}$
Ce grand-O est clairement optimal si l'on considère un paquet de départ croissant.

```
type config = int list array
let patience (cartes : int list) : config =
  let piles = Array.create (List.length cartes) [] in
  let rec empile cartes actuel =
    match cartes, piles.(actuel) with
      | [], _ -> piles
      | x :: xs, y :: ys when x >= y -> empile cartes (actuel + 1)
      | x :: xs, u -> piles.(actuel) <- x :: u; empile xs 0
  in
  empile cartes 0
```

► **Question 16** Les sommets des piles étant en ordre croissant, on peut procéder par dichotomie.
La recherche de la pile est alors logarithmique en le nombre de piles et donc en $|s|$, ce qui, répété $|s|$ fois, donne $\boxed{\text{une complexité totale en } O(|s| \cdot \ln |s|).}$

► **Question 17**

```ocaml
(* Version avec recherche dichotomique de la première pile légale. *)
let patience_opt (cartes : int list) : config =
  let n = List.length cartes in
  let piles = Array.create n [] in
  (* Renvoie le + petit i tq deb <= i <= fin et
     (carte < List.hd piles.(i) OU piles.(i) = []) *)
  let rec num_pile carte deb fin =
    if fin = deb then
      fin
    else
      let mil = (deb + fin) / 2 in
      match piles.(mil) with
      | x :: xs when carte >= x -> num_pile carte (mil + 1) fin
      | _ -> num_pile carte deb mil in
  let rec empile cartes =
    match cartes with
    | [] -> piles
    | x :: xs -> let k = num_pile x 0 (n - 1) in
      piles.(k) <- x :: piles.(k);
      empile xs in
  empile cartes
```

► **Question 18** Il suffit de compter le nombre de piles utilisées (c'est-à-dire non vides). Une solution parmi d'autres :

```ocaml
(*
   nb_si_filtre : 'a array -> ('a -> bool) -> int
   Compte le nombre d'élément de t vérifiant le prédicat filtre
*)
let nb_si_filtre t filtre =
  let nb = ref 0 in
  for k = 0 to Array.length t - 1 do
    if filtre t.(k) then nb := !nb + 1
  done;
  !nb

let l_seq_patience_1 u =
  nb_si_filtre (patience u) (fun v -> v <> [])

let l_seq_patience u =
  nb_si_filtre (patience_opt u) (fun v -> v <> [])
```

► **Question 19** Il faut stocker les pointeurs imaginés plus haut. On n'empile donc plus des entiers, mais des cellules constituées d'un entier et d'un pointeur vers une autre cellule : autrement dit, des listes.

► **Question 20** Pour éviter d'avoir un cas particulier pour la première pile (pas de pointeur), on rajoute une pile fictive tout à gauche constituée uniquement d'un sommet égal à la liste vide (cette pile vaut donc `[[]]`). Ainsi, quand on rajoute un élément sur la « vraie » pile la plus à gauche, on peut le faire pointer vers le sommet de cette pile fictive. Il faut modifier un peu la dichotomie puisque la numérotation des piles va maintenant de 1 à n.

```
type pile_avec_pointeurs = int list list
type config_avec_pointeurs = pile_avec_pointeurs array


(* Renvoie le numéro de pile (entre deb et fin) sur lequel il
   faut empiler carte. *)
let rec num_pile_pointeurs carte deb fin (piles : config_avec_pointeurs) =
  if fin = deb then
    fin
  else
    let mil = (deb + fin) / 2 in
    match List.hd piles.(mil) with
    | x :: xs when carte >= x -> num_pile_pointeurs carte (mil + 1) fin piles
    | _ -> num_pile_pointeurs carte deb mil piles

let rec empile_pointeurs cartes (piles: config_avec_pointeurs) : unit =
  match cartes with
  | [] -> ()
  | x :: xs ->
    let k = num_pile_pointeurs x 1 (Array.length piles) piles in
    piles.(k) <- (x :: List.hd piles.(k - 1)) :: piles.(k);
    empile_pointeurs xs piles

let sous_sequence_patience_1 cartes =
  let piles = Array.create (List.length cartes + 1) [[]] in
  let rec recup piles actuel k =
    match piles.(k) with
      | [[]] -> actuel
      | x :: xs -> recup piles x (k + 1) in
  empile_pointeurs cartes piles;
  List.rev (recup piles [] 1)
```

Ce code n'est pas très satisfaisant (en particulier tous les `List.hd`). On peut en fait faire plus simple en remarquant que les cartes qui ne sont pas au sommet d'une pile sont « mortes » : on n'aura plus jamais besoin de créer un pointeur vers une telle carte. On obtient le code suivant :

```ocaml
let rec num_pile carte deb fin piles =
  if fin = deb then
    fin
  else
    let mil = (deb + fin) / 2 in
    match piles.(mil) with
    | x :: xs when carte >= x -> num_pile carte (mil + 1) fin piles
    | _ -> num_pile carte deb mil piles

let rec empile cartes piles =
  match cartes with
  | [] -> ()
  | x :: xs -> let k = num_pile x 1 (Array.length piles) piles in
    piles.(k) <- x :: piles.(k - 1);
    empile xs piles

let sous_sequence_patience_2 cartes =
  let piles = Array.create (List.length cartes + 1) [] in
  let rec recup piles actuel k =
    match piles.(k) with
    | [] -> actuel
    | _ -> recup piles (piles.(k)) (k + 1) in
  empile cartes piles;
  List.rev (recup piles [] 1)
```

1 Bonus

► **Question 21** Jouons une partie en utilisant la stratégie gloutonne avec ce paquet de n^2+1 cartes. Notons p la hauteur maximale des piles et q le nombre de piles.

- On a $pq \geqslant n^2+1$ (les cartes doivent rentrer dans un rectangle de taille p, q), et donc $\max(p, q) \geqslant n+1$.
- Une pile, lue de bas en haut, fournit une sous-séquence décroissante (nous l'avons déja remarqué plus haut), il y a donc une sous-séquence décroissante de longueur p.
- D'après la partie précédente, il y a une sous-séquence croissante de longueur q.

$\boxed{\text{On a donc bien une sous-séquence monotone de longueur supérieure ou égale à } n+1.}$