Patience

1 Définitions et notations

Dans tout le sujet, on s'intéresse à des séquences $s = s_0, \dots, s_{n-1}$ 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 $s = s_0, \dots, s_{n-1}$:

— la *longueur* de s, notée $|s|$, est son nombre d'éléments n ;

— s est dite croissante si $s_0 \leqslant s_1 \leqslant \dots \leqslant s_{n-1}$;

— une *sous-séquence* de longueur k de s est une séquence $s_{\varphi(0)}, \dots, s_{\varphi(k-1)}$ 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 :

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

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 $u = 1, 2, 4, 5, 8$:

2 Méthode par énumération

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

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).
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.
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]`.
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]; []]`.
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.
6. Quelle est la complexité de `l_seq_naif` (en fonction de $|s|$) ?

3 Méthode par programmation dynamique

Dans cette partie, on représente une séquence $s = s_0, \dots, s_{n-1}$ 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 $s_{\varphi(0)}, \dots, s_{\varphi(i)}$ 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

1. Montrer que, pour $0 \leqslant k < n - 1$, on a :

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

2. En déduire une fonction `aux_dyn (s : int array) : int array` qui renvoie le tableau `longueurs` associé à s. On demande une complexité en $\mathrm{O}\left(|s|^2\right)$, que l'on justifiera.
3. Écrire alors une fonction `l_seq_dyn (s : int array) : int` calculant $l_{seq}(s)$ et préciser sa complexité.
4. 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`.

4 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 :

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

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.

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

5				
2			2	
8		2	3	
9	$\longrightarrow$	5	8	9
3		p_1	p_2	p_3
2				

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

3. Montrer que quelle que soit la stratégie utilisée, on utilise au moins $l_{seq}(s)$ piles.
4. 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
```

5. 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 $\mathrm{O}\left(|s|^2\right)$ dans le pire des cas, ce que l'on justifiera.
6. Comment pourrait-on utiliser le résultat de la question 2 pour abaisser la complexité de la fonction `patience` ?
   À cette question, on demande seulement de donner l'idée de l'algorithme.

7. Écrire une fonction `patience_opt : int list -> config` ayant la même spécification que `patience` mais une complexité en $\mathrm{O}(|s| \cdot \log |s|)$, que l'on justifiera.
8. Écrire une fonction `l_seq_patience (s : int list) : int` qui calcule $l_{seq}(s)$ en temps $\mathrm{O}(|s| \cdot \log |s|)$.
9. Expliquer comment modifier la manière de stocker les configurations pour pouvoir reconstruire à la fin du calcul une sous-séquence croissante de longueur maximale.
10. Écrire une fonction `sous_sequence_patience (s : int list) : int list` qui renvoie une sous-séquence croissante de s de longueur maximale en temps $\mathrm{O}(|s| \ln |s|)$.

5 Bonus

1. Démontrer le théorème suivant :

Théorème 5.1: Erdös-Szekeres

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