Soit un  $\mathit{demi-groupe}\,(S,+)$ , c'est-à-dire que

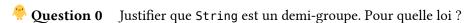
• S est stable par +

• La loi + est associative

On considère que tous les éléments de S occupent une taille constante en mémoire.

Soit  $L \in S^{[\![1,n]\!]}$  une liste d'éléments de S, et  $w \leq n$ .

# Un algorithme insatisfaisant



# Correction

String est un demi-groupe pour la loi de concaténation.

🤗 Question 1 🛮 Est-il possible d'avoir un demi-groupe sans élément neutre ?

# Correction

 $(\mathbb{N}^\star,+)$  est un demi-groupe sans être un monoïde.

On définit la liste W de longueur n - w + 1,

$$W[i] := \sum_{k=0}^{w-1} L[i+k]$$

**Question 2** Si L = [a, b, c, d, e, f] et w = 4, que vaut W?

### Correction

W = [abcd, bcde, cdef].

 $\stackrel{\P}{\hookrightarrow}$  **Question 3** Déterminer un algorithme qui calcule la liste W.

### Correction

```
(* somme dans notre demigroupe *)
let sum list =
  let rec aux acc = function
    | [] -> acc
    \mid x :: xs \rightarrow sum (x + acc) xs
 in aux (List.hd list) (List.tl list)
(* fonction utilitaire prenant les n premiers éléments *)
let take n l =
  if n \le 0 then []
  else match l with
    | [] -> []
    | x :: xs -> x :: take (n-1) xs
(* calcul naïf *)
let calcul_w w l =
  let prefix = take w l in
  (* si on a encore assez de termes *)
  if List.length prefix = w
    then fold prefix (* la somme demandée *)
         (* on continue le calcul *) :: calcul w w (List.tl l)
    else []
On aurait aussi pu implémenter cette fonction avec des arrays et deux boucles for.
```

# Souvenirs, souvenirs

**Question 4** En considérant W[2], W[3], ..., déterminer un ordre judicieux d'évaluation de la somme W[1].

#### Correction

On peut reprendre l'exemple précédent pour se donner une intuition. W[1] vaut abcd et on cherche un ordre de calcul approprié. On remarque que W[2] vaut bcde et que W[3] vaut cdef. Dans l'optique de faire de la programmation dynamique, il est alors judicieux d'évaluer et stocker bcd et cd. On cherche donc à évaluer W[1] dans l'ordre suivant  $L[1]+(L[2]+(L[3]+\ldots+(L[w-1]+L[w])))$  pour stocker les résultats intermédiaires.

**Question 5** Dans le cas  $w = \frac{n}{2} + 1$ , déterminer un algorithme s'exécutant en temps linéaire.

#### Correction

On coupe la liste en deux, comme indique l'énoncé. On remarque que l'exemple donné correspond à ce cas. On sépare ainsi L en a,b,c et d,e,f. En évaluant la première moitié de droite à gauche (d'après la question précédente) et la deuxième moitié de gauche à droite, on obtient les précalculs suivants: c,bc,abc et d,de,def. Si on croise les termes en recombinant, on obtient W.

abc	bc	a
+	+	+
d	de	def
=	=	=
abcd	bcde	cdef

De cet exemple on généralise immédiatement au cas  $w=\frac{n}{2}+1.$ 

 $\upghtarrow$  Question 6 En déduire un algorithme calculant W.

Une complexité temporelle en  $\mathcal{O}(n)$  et spatiale (on ne compte pas W) en  $\mathcal{O}(w)$  sont attendues.

#### Correction

On segmente la liste en blocs de taille  $m \coloneqq w-1$ . On applique l'algorithme précédent à chaque paire de blocs consécutifs. Les éléments hors de blocs sont des cas de bord et n'influencent pas sur la complexité (en réalité, on peut traîter ceux-ci en même temps que les autres avec une implémentation appropriée).

La complexité en espace est garantie car à chaque étape on ne stocke que deux tableaux de taille w-1. Enfin, pour chaque paire de blocs consécutifs on passe un temps de 3(w-1) (création des tableaux puis recombinaison). On passe au plus sur un bloc deux fois, donc le coût total est au plus  $6\frac{b}{w-1}$  où b est le nombre de blocs  $b:=\frac{n}{w-1}$ . On conclut donc que le coût est en  $\mathcal{O}(n)$ .

Afin d'offrir une solution complète, on propose la solution suivante en OCaml, en supposant l'existence de fonctions auxiliaires

```
(* découpe la liste en blocs *)
val chunk : int -> 'a list -> 'a list list;;
(* applique une opération à deux listes terme à terme *)
val zipWith : ('a -> 'a -> 'a) -> 'a list -> 'a list -> 'a list;;

(* agit comme `sum` mais renvoie une liste des résultats intermédiaires *)
val scanL : S list -> S list;;
(* idem, mais en sommant de droite à gauche *)
val scanR : S list -> S list;;

let calcul_w w liste =
    (* opération valide dans le cas w=n/2-1 *)
    let combine gauche droite = zipWith (+) (scanR gauche) (scanL droite) in
    let fenetres = chunk (w - 1) liste in
    zipWith combine (* combine les blocs successifs *)
    fenetres (List.tl fenetres) (* donne les paires de blocs successifs *)
    |> List.concat
```