**Exercice 1. Minimum de fenêtre glissante (Amin BELFKIRA)**

Soit `a` un tableau de $n$ entiers et $k \in \mathbb{N}$. L'objectif de cet exercice est de créer un tableau `a_min` de taille $n - k$ tel que `a_min[i]` soit le minimum des `a[j]` pour $j \in [\![i, i + k]\!]$.

1. Quelle serait la complexité de la méthode naïve ?
2. Écrire un meilleur algorithme en pseudo-code, en utilisant une file de priorité. Quelle serait la complexité ?

Dans la suite, on va implémenter un algorithme utilisant une file à deux bouts `q` (*deque* ou *double-ended queue* en anglais) qui va contenir des éléments de `a`. Une file à deux bouts possède les opérations suivantes :

- `add_right(q, x)` : ajoute l'élément $x$ à la fin de `q`.
- `add_left(q, x)` : ajoute l'élément $x$ au début de `q`.
- `peek_right(q)` : renvoie l'élément à la fin de `q`.
- `peek_left(q)` : renvoie l'élément au début de `q`.
- `pop_right(q)` : supprime l'élément à la fin de `q`.
- `pop_left(q)` : supprime l'élément au début de `q`.
- `is_empty(q)` : détermine si `q` est vide.

3. Donner des implémentations possibles pour une file à deux bouts et discuter de leurs complexités. Implémenter une file à deux bouts en C (on écrira seulement `add_right`, `peek_right`, `pop_right`).

À l'itération $i$, l'algorithme va considérer `a[i]` avec l'invariant de boucle suivant : `q` contient, dans l'ordre, des indices $i_1, ..., i_p$ de `a` (où $i_1$ est l'élément au début de `q` et $i_p$ l'élément de fin) tels que :

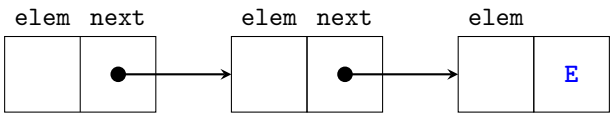
- $i_1 \leq i_2 \leq ... \leq i_p$.
- `a[i_1]` $\leq$ `a[i_2]` $\leq ... \leq$ `a[i_p]`.
- $i_1 > i - k$ (on considère seulement les $k$ derniers éléments).

4. Avec ces notations, que vaut `a_min[i]` ?
5. Comment mettre à jour `q` à l'itération $i + 1$ ?
6. En déduire une fonction `int* minimum_sliding(int* a, int n, int k)` renvoyant `a_min`.
7. Quelle est la complexité de `minimum_sliding` ?

**Exercice 2. Algorithme du lièvre et de la tortue (DERU Victor)**

On considère un type `linked_list` de liste simplement chaînée impérative (chaque élément a accès à l'élément suivant `next`) :

```
type 'a cell = { elem : 'a, next : 'a linked_list }
and 'a linked_list = E | C of 'a cell
```




1. Écrire une/des instruction(s) OCaml pour définir une liste simplement chaînée `l` contenant les entiers 1 et 2.
2. Écrire une fonction

   ```
   to_list : 'a linked_list -> 'a list
   ```

   convertissant une liste simplement chaînée en `list` classique.

Il est possible qu'une liste simplement chaînée possède un cycle, si l'on revient sur le même élément après avoir parcouru plusieurs successeurs. Dans ce cas, la fonction `to_list` précédente ne termine pas...
On souhaite donc déterminer algorithmiquement si une liste simplement chaînée `l` possède un cycle.

3. Écrire une fonction

   ```
   has_cycle : 'a linked_list -> bool
   ```

   déterminant si une `linked_list` possède un cycle.
   Si $n$ est le nombre d'éléments de `l`, quelle est la complexité de cet algorithme, en temps et en mémoire?

Il existe un algorithme plus efficace, appelé algorithme du lièvre et de la tortue (ou : algorithme de Floyd).
Il consiste à initialiser une variable `tortue` à la case de `l`, une variable `lievre` à la case suivante, puis, tant que c'est possible :

- Si `lievre` et `tortue` font référence à la même case, affirmer que `l` contient un cycle.
- Sinon, avancer `lievre` de deux cases et `tortue` d'une case.

4. Montrer que cet algorithme permet bien de détecter un cycle. Quelle est sa complexité en temps et en espace?
5. Comment obtenir la longueur du cycle, s'il existe?
6. Écrire une fonction utilitaire

   ```
   step : 'a linked_list -> 'a linked_list
   ```

   telle que `step l` avance `l` d'une case ou renvoie `E` si `l = E`.
7. Écrire une fonction récursive

   ```
   has_cycle : 'a linked_list -> 'a linked_list -> bool
   ```

   implémentant l'algorithme du lièvre et de la tortue, dont les deux arguments sont les positions actuelles du lièvre et de la tortue (pour savoir si `l` contient un cycle, on appelera donc `has_cycle (step l) l`).
   On pourra utiliser == qui compare 2 objets en mémoire (à ne pas confondre avec = qui les compare en valeur).