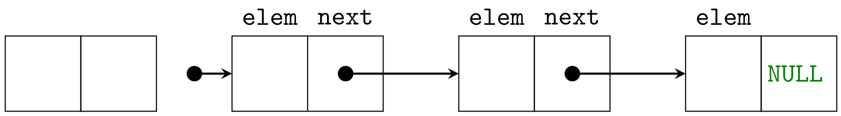
On considère un type `liste1` de liste simplement chaînée impérative (chaque élément a accès à l'élément suivant `next`):

```
type 'a case = { elem : 'a; mutable next : 'a liste1 }
and 'a liste1 = Vide | C of 'a case;;
```



Ici on utilise `and` pour définir 2 types simultanéments (de la même façon que l'on peut définir 2 fonctions récursives mutuellement dépendantes avec `and`). On a besoin de le faire afin de gérer la fin de liste (Vide).

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

> **Solution :** On propose deux solutions possibles (en une ou deux étapes):
>
> ```
> let l = C({elem = 1; next = C({elem = 2; next = Vide})});;
>
> let l1 = C({elem = 2; next = Vide});;
> let l = C{elem = 1; next = l1};;
> ```

2. Écrire une fonction `to_list : 'a liste1 -> 'a list` convertissant une liste simplement chaînée en `list` classique.

> **Solution :**
>
> ```
> let rec to_list l = match l with
>   | Vide -> []
>   | C(c) -> c.elem::to_list c.next;;
> ```

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 récursive `has_cycle : 'a liste1 -> 'a list -> bool` telle que `has_cycle l vus` détermine si `l` possède un cycle, en stockant les éléments déjà rencontrés dans `vus` (initialement on utilise une liste vide pour `vus`).
Si n est le nombre d'éléments de `l`, quelle est la complexité de cet algorithme, en temps et en mémoire?

> **Solution :** Il va falloir stocker dans `vus` au plus les n éléments de `l`, d'où une complexité en mémoire de $\boxed{O(n)}$.
> Il y a au plus n appels récursifs de `has_cycle`, chacun en $O(n)$ puisque `List.mem e vus` est en $O(n)$ (`List.mem` parcourt chaque élément de `vus` dont la taille est au plus n). D'où une complexité en temps de $\boxed{O(n^2)}$.
>
> ```
> let rec has_cycle l vus =
>   match l with
>   | Vide -> false
>   | C(c) -> let e = c.elem in
>             List.mem e vus || has_cycle c.next (e::vus);;
> ```

Il existe un algorithme plus efficace, appelé algorithme du lièvre et de la tortue (ou: algorithme de Floyd/algorithme rho de Pollard, très utile en cryptographie).
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?

> **Solution :** Supposons qu'il existe un cycle de taille p. Alors, au bout d'un moment, `lievre` et `tortue` seront dans le cycle, disons à des positions ℓ et t à partir du début du cycle. Comme `lievre` avance de 1 case relativement à `tortue`, `lievre` et `tortue` seront sur la même case au bout de $(\ell - t) \mod p$ itérations.
> S'il n'y a pas de cycle, il est évident que `lievre` et `tortue` ne seront jamais sur la même case.

> Complexité en espace: 2 (stocker `lievre` et `tortue`...). Complexité en temps: $O(n)$.

5. Comment obtenir la longueur du cycle, s'il existe?

> **Solution** : C'est la durée entre 2 rencontres de `lievre` et `tortue`.

6. Écrire une fonction utilitaire `step : 'a liste1 -> 'a liste1` telle que `step l` avance `l` d'une case ou renvoie `Vide` si `l = Vide`.

> **Solution** :

```
let step l = match l with
  | Vide -> Vide
  | C(c) -> c.next;;
```

7. Écrire une fonction récursive `has_cycle : 'a liste1 -> 'a liste1 -> 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**).

> **Solution** : Voici une solution récursive ainsi qu'une solution impérative (avec boucle `while`) pour comparer:

```
let rec has_cycle lievre tortue = match lievre, tortue with
  | Vide, _ -> false
  | C(l), C(t) -> l == t || has_cycle (step (step lievre)) (step tortue);;
```

```
let has_cycle l =
  let tortue = ref l in
  let lievre = ref (step l) in
  while !lievre <> Vide && !lievre != !tortue do
    tortue := step !tortue;
    lievre := step (step !lievre)
  done;
  !lievre == !tortue;;
```