Question I.1) *Définition inductive de la concaténation en Ocaml*

Pour toutes listes `l` et `q`, et tout élément `e` : $\begin{cases} \texttt{[] @ l = l} \\ \texttt{(e::q @ l = e::(q @ l))} \end{cases}$

Question I.2) *Rédaction de démonstration par induction structurelle*

Soit `l2` une liste. Montrons par récurrence structurelle sur `l1` que `|l1 @ l2|` = `|l1 + l2|` pour toutes listes `l1` et `l2`.

- Si `l1 = []`, `|l1 @ l2|` = `|l2|` = `|l1 + l2|`
- Si `l1` = `e::q`, supposons que `|q @ l2|` = `|q|` + `|l2|`, pour toute suite l2 (*hypothèse d'induction*) ()

$$\begin{aligned} \texttt{|l1 @ l2|} &= \texttt{|(e::q) @ l2|} & \\ &= \texttt{|e::(q @ l2)|} & \text{par définition de } \texttt{@} \\ &= 1 + \texttt{|q @ l2|} & \text{par définition de } \texttt{l} \mapsto \texttt{l} \\ &= 1 + \texttt{|q|} + \texttt{|l2|} & \text{par hypothèse d'induction} \\ &= \texttt{|e::q|} + \texttt{|l2|} & \\ \texttt{|l1 @ l2|} &= \texttt{|l1|} + \texttt{|l2|} & \end{aligned}$$

Ainsi, par induction structurelle, `|l1 @ l2|` = `|l1|` + `|l2|`, pour toutes listes `l1` et `l2`, et ce, indépendamment du choix de `l2`

Question II.1) *|reverse(l1)| = |l1|*

Montrons par induction structurelle sur `l1` que pour toute liste `l1`, `|reverse(l1)|` = `|l1|`.

- Si `l1` = [], `reverse(l1)` = `reverse([])` = `[]` = `l1`
- Si `l1` = `e::q`, on suppose que `|reverse(q)|` = `|q|`

$$\begin{aligned} \texttt{|reverse(l1)|} &= \texttt{|reverse(e::q)|} & \\ &= \texttt{|reverse(q) @ [e]|} & \text{définition de } \texttt{reverse} \\ &= \texttt{|reverse(q)|} + \texttt{|[e]|} & \\ &= \texttt{|q|} + \texttt{|[e]|} & \text{hypothèse d'induction} \\ &= \texttt{|e::q|} & \\ \texttt{|reverse(l1)|} &= \texttt{|l1|} & \end{aligned}$$

Ainsi, par induction structurelle, pour toute liste `l1`, `|reverse(l1)|` = `|l1|`.

Question II.1) *reverse(l1 @ l2) = reverse(l2) @ reverse(l1)*

Montrons, par induction structurelle sur `l1`, que pour toute liste `l1`,

$$\texttt{reverse(l1 @ l2)} = \texttt{reverse(l2) @ reverse(l1)}$$

.

- Si `l1` = `[]`, alors `reverse([] @ l2)` = `reverse(l2)` = `reverse(l2) @ reverse([])`
- Si `l1` = `e::q`, supposons que `reverse(q @ l2)` = `reverse(l2) @ reverse(q)`.

$$\begin{aligned}
\text{Alors, } \texttt{reverse(l1 @ l2)} &= \texttt{reverse(e::q @ l2)} && \\
&= \texttt{reverse(q @ l2) @ [e]} && \text{par définition} \\
&= \texttt{reverse(l2) @ reverse(q) @ [e]} && \text{hypothèse d'induction} \\
&= \texttt{reverse(l2) @ reverse(e::q)} && \text{par définition} \\
&= \texttt{reverse(l2) @ reverse(l1)} &&
\end{aligned}$$

Donc, par induction structurelle sur `l1`, $\forall$`l1`, `reverse(l1 @ l2)` = `reverse(l2) @ reverse(l1)`

Question II.2)

Pour tout $n \in \mathbb{N}$ on pose C_n la complexité en pire cas d'un appel `reverse liste`, où $|\texttt{liste}| = n$.
Un appel sur une liste non vide `liste` $=$ `e::q` engendre un appel récursif sur la liste `q`. De plus, l'opérateur `@` a une complexité linéaire par rapport à la taille de la première liste.

Évaluation de la complexité :

- Version courte
  Donc $C_n = \mathcal{O}(n^2)$
- Version longue

  Soit $A > 0$ (*A est la grandeur qui prend en facteur l'argument du $\mathcal{O}$*) tel que : $\begin{cases} \forall n \in \mathbb{N}, C_{n+1} \leq C_n + An \\ C_0 \leq A \end{cases}$

$$\begin{aligned} \text{Donc} \quad \forall n \in \mathbb{N}, C_n &= [\sum_{k=0}^{n-1} (C_{k+1} - C_k)] + C_0 \\ &\leq [\sum_{k=0}^{n-1} An] + A \\ &= A\frac{n(n-1)}{2} + A \\ &= A(\frac{n(n-1)}{2} + 1) \end{aligned}$$

Question II.3)

```
let renv liste =
    let rec transfert l q = match l with
    (* renvoie reverse(l) @ q *)
        | [] -> q
        | e::t -> transfert t (e::q)
    in transfert liste []
```

- Correction de la fonction `transfert`
  On prouve par induction structurelle sur `l` que `transfert l q` termine et renvoie `reverse(l) @ q`, pour toutes listes `l` et `q`.
  - Si `l` = `[]`, `transfert l q` termine et renvoie `q = reverse([]) @ q`, pour toute liste `q`
  - Sinon, `l` = `e::t` : on suppose que pour toute liste `q`,
    `transfert t q` termine et renvoie `reverse(t) @ q`
    Pour toute liste `q`, l'appel `transfert l q` effectue un appel récursif `transfert t (e::q)` qui termine et renvoie `reverse(t) @ (e::q)` par hypothèse d'induction.
    Or `|l|` = `|e::t|` = `reverse(t) @ [e]`
    Donc

    $$\begin{aligned} \texttt{reverse(l) @ q} &= \texttt{reverse(t) @ [e] @ q} \\ &= \texttt{reverse(t) @ (e::q)} \end{aligned}$$

    Donc `transfert l q` termine et renvoie [...]
    D'où, par induction structurelle sur `l`, pour toutes listes `l` et `q`, `transfert l q` termine et renvoie [...]
    D'où la correction de `transfert`. On termine par la correction de `renv`.

Question II.5) *Complexité de* `transfert`

Un appel `transfert (e::t) q` évalue `e::q` en temps constant pour effectuer un appel récursif `transfert t (e::q)`.
En notant C_n la complexité d'un appel `transfert l q` où `l` est une liste de taille n :

$$C_{n+1} = C_n + \Theta(1)$$

Donc $C_n = \Theta(n)$. Un appel `renv l` a donc une complexité $\Theta(\texttt{|l|})$

Question III.1)

```
let rec separe l l1 l2 = match l with
    |[] -> (l1,l2)
    | e::q -> separe q (e::l2) l1
```

Question III.2)

À chaque appel récursif, la taille ($\in \mathbb{N}$) de la première liste passée en argument diminue de 1, celle-ci constitue donc un variant aux appels issus de l'appel initial `separe l l1 l2`, ce qui prouve la terminaison de cet appel.

On montre par induction structurelle la correction de `transfert`.
Soit l1, l2

Question III.3)