Exercice 1. Terminaison
Montrer que les fonctions suivantes terminent, pour des arguments entiers :

```
let rec g a =
    if a < 0 then 1
    else if a mod 2 = 0 then g (a + 1)
    else g (a - 3)
```

```
let rec f a b =
    if a = 0 || b = 0 then 1
    else if a mod 2 = 0 then f (a/3) (2*b)
    else f (3*a) (b/5)
```

Solution : Supposons que l'on appelle `g a` et notons a_k les arguments de ses appels récursifs, avec $a_0 = \texttt{a}$.
Soit $k \in \mathbb{N}$. Si a_k est pair, $a_{k+1} = a_k + 1$ est impair et $a_{k+2} = a_{k+1} - 3 = a_k - 2$ est pair. Ainsi, a_{k+2p} est une suite strictement décroissante d'entiers, qui devient donc négatif et la fonction termine.
Si a_k est impair, a_{k+1} est pair et on se ramène au cas précédent.

Exercice 2. Invariant de boucle simple
En utilisant un invariant de boucle, prouver que la fonction suivante renvoie bien la somme des éléments d'un tableau :

```
let somme t =
    let s = ref 0 in
    for i = 0 to Array.length t - 1 do
        s := !s + t.(i)
    done;
    !s
```

On rappelle qu'un invariant de boucle est une propriété qui reste vraie à chaque itération de la boucle et qui permet de montrer que la fonction renvoie le bon résultat (ici, la somme des éléments de `t`).
On prouve cet invariant de boucle par récurrence sur le nombre d'itérations dans la boucle.

Solution : On pose H_i : "au début de l'itération i de la boucle `for`, `s` contient $\sum_{k=0}^{i-1} t.(k)$".

H_0 est vrai car $\sum_{k=0}^{i-1} t.(k) = 0 = s$ initialement.

Supposons H_i. Alors, s vaut $\sum_{k=0}^{i-1} t.(i)$ au début de l'itération i.

Comme l'itération i ajoute $t.(i)$ à s, s contient $\sum_{k=0}^{i} t.(k)$ au début de l'itération $i+1$, ce qui prouve H_{i+1}.

Par principe de récurrence, la valeur renvoyée par `somme` est bien la somme des éléments de `t`.

Exercice 3. Tranche maximum (algorithme de Kadane)

Soit `t` un tableau d'entiers. Une **somme consécutive** (ou tranche) dans `t` est de la forme $\sum_{k=i}^{j} \texttt{t.(k)}$ (où i et j sont des indices de `t`). On note s la valeur maximum d'une somme consécutive.

1. Écrire une fonction `tranche_max` prenant `t` en argument et renvoyant `s`, en complexité quadratique en la taille de `t`.
   Solution :

```
let tranche_max t =
    let maxi = ref t.(0) in
    let n = Array.length t in
    for i = 0 to n - 1 do
        let sum = ref 0 in
        for j = i to n - 1 do
            sum := !sum + t.(j); (* sum contient la somme de t.(i) à t.(j) *)
            maxi := max !maxi !sum
        done;
    done;
    !maxi
```

Si j est un indice de `t`, on note s_j la plus grande somme consécutive finissant en j. Dit autrement :

$$s_j = \max_{0 \leq i \leq j} \sum_{k=i}^{j} t.(k)$$

2. Calculer tous les s_j, si `t = [|1; -4; 1; 5; -7; 0|]`
   Solution :

indice	0	1	2	3	4	5
`t`	1	-4	1	5	-7	0
`s`	1	-3	1	6	-1	0

3. Si $j > 0$, montrer que :
$$s_j = \max(s_{j-1} + t.(j),\ t.(j))$$

Solution : Soit i le début de la somme s_j (c'est à dire $s_j = \sum_{k=i}^{j} t.(k)$).

- Si $i = j$: alors $\boxed{s_j = t.(j)}$
- Sinon, $i < j$ et $s_j = \sum_{k=i}^{j-1} t.(k) + t.(j)$. Clairement, $\sum_{k=i}^{j-1} t.(k)$ est une tranche finissant en $j-1$.
  Supposons que $\sum_{k=i}^{j-1} t.(k)$ ne soit pas une tranche maximum finissant en $j-1$ (c'est à dire $\sum_{k=i}^{j-1} t.(k) < s_{j-1}$).
  Alors $s_{j-1} + t.(j)$ est un tranche finissant en j qui est plus grande que s_j. C'est absurde par définition de s_j.
  Donc $\sum_{k=i}^{j-1} t.(k) = s_{j-1}$ et $\boxed{s_j = s_{j-1} + t.(j)}$.

On a donc soit $s_j = t.(j)$, soit $s_j = s_{j-1} + t.(j)$. Comme s_j est la somme maximum, on conserve le plus grand des deux :

$$\boxed{s_j = \max(s_{j-1} + t.(j),\ t.(j))}$$

4. Comment peut-on exprimer s en fonction de s_j ?
   Solution : s est le maximum des s_j.
5. En déduire une fonction `tranche_max` prenant `t` en argument et renvoyant `s`, en complexité linéaire en la taille de `t`.
   Solution :

```
let tranche_max t =
    let s = ref t.(0) in
    let sj = ref t.(0) in
    for j = 1 to Array.length t - 1 do
        sj := max (!sj + t.(j)) t.(j);
        s := max !s !sj
    done;
    !s;;
```

6. Donner un invariant de boucle permettant de prouver que `tranche_max t` est correct.

   Solution : Invariant de boucle H_j : "au début de l'itération j de la boucle `for`, sj est la tranche maximum finissant en `t.(j)` et s est le max des sk pour $k \leq j$"

7. Modifier votre fonction précédente pour obtenir les indices de début et fin de s.

   Solution :

```
let tranche_max t = (* renvoie (tranche max, son début, sa fin *)
    let s = ref t.(0) in
    let sj = ref t.(0) in
    let deb = ref 0 in (* indice du début de la somme dans s *)
    let fin = ref 0 in (* indice de fin de la somme dans s *)
    let deb_sj = ref 0 in (* indice du début de la somme dans sj *)
    for j = 1 to Array.length t - 1 do (* O(n) où n = taille de t *)
        sj := !sj + t.(j);
        if !sj < t.(j) then (sj := t.(j); deb_sj := j);
        if !s < !sj then (s := !sj; deb := !deb_sj; fin := j)
    done;
    (!s, !deb, !fin)
```

Exercice 4. Tri par insertion

1. Écrire une fonction `insere` telle que, si `l` est une liste triée et `e` un élément, `insere l e` renvoie une liste triée contenant `e` et les éléments de `l`.

   Solution :

```
let rec insere e l = match l with
    | [] -> [e]
    | x::q -> if e < x then e::l
                else x::insere e q
```

2. En déduire un algorithme de tri, en utilisant plusieurs fois `insere`. Prouver que ce tri est correct.

   Solution :

```
let rec tri_insertion = function
    | [] -> []
    | e::q -> insere e (tri_insertion q)
```

3. Quelle est la complexité de ce tri ? Pourrait-on l'améliorer en utilisant une recherche par dichotomie pour `insere` ?

   Solution : Si `l` est de taille n, `tri_insertion l` appelle n fois `insere` qui est en $O(n)$, d'où une complexité $\boxed{O(n^2)}$.

4. Réécrire `tri_insertion` avec un tableau au lieu d'une liste et en utilisant une recherche par dichotomie pour l'insertion. Que peut-on dire de sa complexité ? Et de son nombre de comparaisons ?

   Solution : La recherche par dichotomie permet de trouver la position où insérer l'élément en $O(\log(n))$ comparaisons.

Comme on le répète n fois, on a un tri en $\boxed{\mathrm{O}(n \log n)}$ comparaisons.
Par contre, pour insérer l'élément, il faut déplacer tous les éléments suivants ce qui demande $\mathrm{O}(n)$ opérations. La complexité reste donc $\boxed{\mathrm{O}(n^2)}$.

```
let swap t i j =
    let tmp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- tmp;;

let insere t pos =
    let rec dicho i j =
    (* renvoie l'indice où t.(pos) doit être inséré dans t *)
        if i >= j then i
        else let m = (i + j)/2 in
        if t.(m) < t.(pos) then dicho (m + 1) j
        else dicho i m in
    let k = dicho 0 pos in
    for i = pos downto k+1 do (* on met t.(pos) à sa place *)
        swap t (i-1) i
    done;;

let tri_insertion t =
    for i = 0 to Array.length t - 1 do
        insere t i
    done
```