1 Analyse de programmes récursifs

Exercice 1 :

```
let rec fact n = match n with
  | 0 -> 1
  | _ -> n * fact (n-1)
```

1. Montrer que la fonction `fact` termine pour tout entier positif pris en argument.
2. Prouver que `fact n` renvoie bien la valeur $n!$ pour tout $n \in \mathbb{N}$.
3. Donner la complexité de `fact n` en fonction n.

Exercice 2 :

```
let rec min_list l = match l with
  | [] -> failwith "liste vide"
  | [x] -> x
  | t::q -> min t (min_list q)
```

1. Montrer que la fonction `min_list` termine pour toute liste prise en argument.
2. Prouver que `min_list l` renvoie bien le minimum de `l` pour toute liste `l` non vide.
3. Donner la complexité de `min_list l` en fonction de la longueur n de `l`.

Exercice 3 :

```
let rec expo x n = match n with
  | 0 -> 1
  | _ -> x * expo x (n-1)
```

```
let rec expo_rapide x n = match n with
  | 0 -> 1
  | _ -> let y = expo_rapide x (n/2) in
         if n mod 2 = 0 then y * y
         else x * y * y
```

1. Montrer que la fonction `expo` termine pour tous entiers positifs pris en arguments.
2. Prouver que `expo x n` renvoie bien x^n pour tous $(x, n) \in \mathbb{N}^2$.
3. Donner la complexité de `expo x n` en fonction de n.
4. Montrer que la fonction `expo_rapide` termine pour touts entiers positifs pris en arguments.
5. Prouver que `expo_rapide x n` renvoie bien x^n pour tous $(x, n) \in \mathbb{N}^2$.
6. Donner la complexité de `expo_rapide x n` en fonction de n.

Exercice 4 :

```
let rec concat1 l1 l2 = match l1 with
  | [] -> l2
  | t::q -> t :: concat1 q l2
```

```
let rec concat2 l1 l2 = match l2 with
  | [] -> l1
  | t::q -> concat2 (l1@[t]) q
```

1. Justifier que les fonctions `concat1` et `concat2` calculent toujours la même chose.
2. Donner la complexité de `concat1` et `concat2` en fonction des longueurs n_1 et n_2 de `l1` et `l2`.

Exercice 5 :

```
let rec fibo1 n = match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> fibo1 (n-1) + fibo1 (n-2)
```

```
let fibo2 n =
  let rec aux n (a,b) = match n with
    | 0 -> a
    | _ -> aux (n-1) (b,a+b)
  in aux n (0,1)
```

1. Justifier que les fonctions `fibo1` et `fibo2` calculent toujours la même chose.
2. Donner la complexité de `fibo1` et `fibo2` en fonction de n.

Exercice 6 :

On dispose de n disques troués en leur centre, numérotés de 1 à n, de diamètres croissants, ainsi que de 3 piquets. Initialement, tous les disques sont enfilés sur le premier piquet, du plus grand au plus petit.

Le but du jeu est de déplacer les disques sur le troisième piquet, en suivant les règles suivantes :

- déplacer les disques un par un, d'un piquet à un autre ;
- un disque ne doit jamais être posé sur un disque de diamètre inférieur.

1. Écrire une fonction `hanoi : int -> unit` affichant à l'écran les mouvements nécessaires pour résoudre ce problème de manière optimale.
2. Justifier que votre algorithme termine et est correct.
3. Quelle est la complexité de votre fonction ?

2 Analyse de programmes impératifs

Exercice 7 :

```
let expo x n =
  let v = ref 1 in
  for i = 0 to n-1 do
    v := !v * x
  done ;
  !v
```

```
let expo_rapide x n =
  let m = ref n in
  let y = ref x in
  let z = ref 1 in
  while !m > 0 do
    if !m mod 2 = 1 then z := !z * !y ;
    m := !m / 2 ;
    y := !y * !y
  done ;
  !z
```

1. Montrer que la fonction `expo` termine pour tous entiers positifs pris en arguments.
2. Prouver que `expo x n` renvoie bien x^n pour tous $(x, n) \in \mathbb{N}^2$.
3. Donner la complexité de `expo x n` en fonction de n.
4. Montrer que la fonction `expo_rapide` termine pour touts entiers positifs pris en arguments.
5. Prouver que `expo_rapide x n` renvoie bien x^n pour tous $(x, n) \in \mathbb{N}^2$.
6. Donner la complexité de `expo_rapide x n` en fonction de n.

Notation. Pour les exercices suivants, on introduit une notation pour faciliter nos raisonnements sur les tableaux : si `t` est un tableau, et si i et j sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`. En particulier, si `t` est de longueur n, `t[0:n]` désigne `t`.

Exercice 8 :

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

1. Justifier que la fonction `somme` termine sur toute entrée.
2. Prouver que pour tout tableau d'entiers `t`, `somme t` renvoie la somme des éléments de `t`.
3. Donner la complexité de `somme t` en fonction de la longueur n de `t`.

Exercice 9 :

```
let recherche x t =
  let n = Array.length t in
  let trouve = ref false in
  for i = 0 to n-1 do
    if t.(i) = x then trouve := true
  done ;
  !trouve
```

```
let recherche_dicho x t =
  let n = Array.length t in
  let trouve = ref false in
  let g = ref 0 in
  let d = ref n in
  while not !trouve && g < d do
    let m = (!g + !d) / 2 in
    if t.(m) = x then trouve := true
    else if t.(m) > x then d := m
    else g := m+1
  done ;
  !trouve
```

1. Justifier que la fonction `recherche` termine sur toute entrée.
2. Prouver que pour toute valeur `x` et pour tout tableau `t`, `recherche x t` renvoie `true` si et seulement si `x` $\in$ `t`.
3. Donner la complexité de `recherche x t` en fonction de la longueur n de `t`.
4. Prouver que la fonction `recherche_dicho termine` sur toute entrée.
5. Prouver que pour toute valeur `x` et pour tout tableau `t`, `recherche_dicho x t` renvoie `true` si et seulement si `x` $\in$ `t`.
6. Donner la complexité de `recherche_dicho x t` en fonction de la longueur n de `t`.

Exercice 10 (tri à bulles) :
Le **tri à bulles** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant :

(a) on parcourt le tableau de gauche à droite, en comparant les éléments consécutifs : s'ils ne sont pas dans le bon ordre, on les échange.

(b) à la fin du parcours :

- si on a effectué au moins un échange durant ce parcours, on recommence l'étape (a) ;
- sinon, le tableau est trié.

1. Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que `swap t i j` échange les valeurs des cases `t.(i)` et `t.(j)`.
2. Écrire une fonction `etape_bulles : 'a array -> bool` prenant en argument un tableau et qui le parcourt et fait des échanges comme indiqué dans l'étape (a). Elle renverra `true` si au moins un échange a été effectué (et `false` sinon).
3. Écrire une fonction `tri_bulles : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri à bulles.
4. Donner des invariants de boucles pour vos fonctions `etape_bulles` et `tri_bulles` permettant de prouver la correction de `tri_bulles`.
5. Prouver ces invariants. En déduire la correction de `tri_bulles` ainsi que sa terminaison.
6. Quelle est la complexité de `tri_bulles t` en fonction de la longueur n de `t`.

Exercice 11 (tri par sélection) :
Dans cet exercice, si `t` est un tableau, et si `i` et `j` sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`.

Le **tri par sélection** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant (on note `t` le tableau et n sa longueur) :

- pour $i \in [\![0, n-2]\!]$:
  (a) on cherche l'indice du minimum de `t[i:n]` (notons $i_{\min}$ l'indice obtenu) ;
  (b) on échange `t.(`i`)` et `t.(`$i_{\min}$`)`.

1. Écrire une fonction `indice_min : 'a array -> int -> int` prenant en arguments un tableau `t` et un indice i, et renvoyant l'indice $i_{\min}$ décrit à l'étape (a).
2. Écrire une fonction `tri_selection : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri par sélection.
3. Justifier que `tri_selection t` termine sur toute entrée `t`, et donner sa complexité en fonction de la longueur n de `t`.
4. Prouver que la fonction `tri_selection` est correcte.

Exercice 12 (tri par insertion) : Dans cet exercice, si `t` est un tableau, et si `i` et `j` sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`.

Le **tri par insertion** est un algorithme qui permet de trier un tableau d'éléments dans l'ordre croissant. Il repose sur le principe suivant (on note `t` le tableau et n sa longueur) :

- pour $i \in [\![1, n-1]\!]$:
  - on suppose que `t[0:i]` est déjà trié, et on insère `t.(i)` au bon endroit pour que `t[0:i+1]` devienne trié (il faut trouver l'indice j où l'insérer, et décaler tous les éléments à partir de ce j d'un cran vers la droite).

1. Écrire une fonction `tri_insertion : 'a array -> unit` prenant en argument un tableau et qui le trie par ordre croissant en utilisant le principe du tri par insertion.
2. Justifier que `tri_insertion t` termine sur toute entrée `t`, et donner sa complexité en fonction de la longueur n de `t`.
3. Prouver que la fonction `tri_insertion` est correcte.

Exercice 13 : Pour un $u_0 \in \mathbb{N}$ fixé, la suite de Syracuse est définie par récurrence comme suit :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3 \times u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

1. Écrire une fonction `temps_de_vol : int -> int` prenant en argument une valeur de u_0 et renvoyant le plus petit n tel que $u_n = 1$.
2. **(bonus)** Montrer que `temps_de_vol u0` termine pour tout $u_0 \in \mathbb{N}$.