Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ **Exercice 1** : *Analyse d'un algorithme*

> « *Le problème du drapeau hollandais est un problème de programmation, présenté par Edsger Dijkstra, qui consiste à réorganiser une collection d'éléments identifiés par leur couleur, sachant que seules trois couleurs sont présentes (par exemple, rouge, blanc, bleu, dans le cas du drapeau des Pays-Bas). Étant donné un nombre quelconque de balles rouges, blanches et bleues alignées dans n'importe quel ordre, le problème est à les réarranger dans le bon ordre : les bleues d'abord, puis les blanches, puis les rouges.* »
>
> (Wikipedia)

On suppose déjà écrite la fonction `echange` de prototype `void echange(int tab[], int i, int j)` qui échange les éléments d'indice `i` et `j` dans le tableau `tab` et on considère dans la suite que cet échange s'effectue en temps constant. On donne ci-dessous une implémentation de l'algorithme du drapeau hollandais en langage C permettant de réarranger les éléments d'un tableau ne contenant les trois valeurs entières 1, 2 et 3 :

```c
// Prend en entrée un tableau (ne contenant que les valeurs 1, 2 et 3)
// Ne renvoie rien
// Réarrange les éléments du tableau de façon à avoir les 1 puis les 2 et les 3
void drapeau_hollandais(int tab[], int taille)
{
    int i1 = 0;
    int i2 = taille - 1;
    int i3 = taille - 1;
    while (i1 <= i2){
        if (tab[i1] == 1){
            i1 = i1 + 1;}
        else{
            if (tab[i1] == 2){
                echange(tab, i1, i2);
                i2 = i2 - 1;}
            else{
                echange(tab, i1, i2);
                echange(tab, i2, i3);
                i3 = i3 - 1;
                i2 = i2 - 1;}
        }
    }
}

```

1. Etude de l'algorithme du drapeau hollandais.

a) Faire fonctionner cet algorithme sur le tableau `tab = {1, 3, 2, 2, 3, 1}`, et donner le contenu de `tab` ainsi que celui des variables `i1`, `i2`, `i3` lors du déroulement de l'algorithme en recopiant et complétant le tableau suivant :

	tab	i1	i2	i3
Initialisation	{1, 3, 2, 2, 3, 1}	0	5	5
Etape 1	{1, 3, 2, 2, 3, 1}	1	5	5
Etape 2	{1, 1, 2, 2, 3, 3}	1	4	4
Etape 3	{1, 1, 2, 2, 3, 3}	2	4	4
Etape 4	{1, 1, 3, 2, 2, 3}	2	3	4
Etape 5	{1, 1, 2, 2, 3, 3}	2	2	3
Etape 5	{1, 1, 2, 2, 3, 3}	2	1	3

b) Prouver la terminaison de cet algorithme.

Montrons que `i2-i1` est un variant :

(H1) `i2-i1` est entier comme différence de deux entiers

(H2) `i2-i1` est positif avant d'entrer dans la boucle et reste positif par condition d'entrée dans la boucle `while`

(H3) `i2-i1` décroit à chaque tour de boucle car soit `i1` augmente (si `tab[i1]==1`) soit `i2` diminue (si `tab[i1]==2` ou `tab[i1==3]`).

Donc `i2-i1` est bien un variant et donc la fonction termine.

c) Prouver la correction de cet algorithme.

Indication : on pourra noter n la taille du tableau et :

— P_1 *la tranche du tableau* `tab` *comprise entre les indices 0 et* `i1` *(exclu)*

— P_2 *la portion du tableau* `tab` *comprise entre les indices* `i2` *et* `i3` *(exclu)*

— P_3 *la portion du tableau* `tab` *comprise entre les indices* `i3` *et* $n-1$ *(exclu)*

Et prouver l'invariant suivant : « P_1 *ne contient que des 1,* P_2 *que des 2 et* P_3 *que des 3 ».*

Avec les notations proposées pour P_1, P$_2$ et P_3, montrons l'invariant = « P_1 ne contient que des 1, P_2 que des 2 et P_3 que des 3 »

— Initialisation : avant d'entrer dans la boucle, $i_1 = 0$, $i_2 = n-1$, $i_3 = n-1$ donc P_1, P_2 et P_3 sont vides et donc I est vraie.

— Conservation : On suppose I vérifié et on montre qu'il reste vraie au tour de boucle suivant, on distingue alors 3 cas suivant la première valeur non encore triée c'est à dire `tab[i1]` :

  — Si `tab[i1]` vaut 1, alors `i1` augmente de 1 donc un 1 est ajouté à P_1 qui puisque I est supposé vérifié en entrant dans la boucle ne contenait que des 1. Ni P_2 ni P_3 ne sont modifiés et donc dans ce cas I rest vrai.

  — Si `tab[i1]` vaut 2, alors ce 2 est placé à l'indice `i2` qui est décrémenté. Cela revient donc à ajouter un 2 à $P2$ qui par hypothèse ne contenait que des 2. Ni P_1, ni P_3 ne sont modifiés et donc I reste vrai.

  — Sinon `tab[i1]` vaut nécessairement 3, alors ce 3 est placé à l'indice `i3`, La partie P_2 est décalée d'un rang à gauche (elle garde la même longueur), i_2 et i_3 sont décrémentés. Un 3 étant ajouté à P_3 qui par hypothèse ne contenait que des 3, l'invariant est préservé.

On en conclut la fonction est totalement correcte (elle termine et elle est correcte)

d) Donner, en la justifiant brièvement, la complexité de l'algorithme du drapeau hollandais.

> En notant n la taille du tableau, la boucle `while` s'exécute n fois et comme elle ne contient que des opérations élémentaires, la complexité de l'algorithme est un $O(n)$.

2. Comparaison avec le tri par insertion

a) Rappeler la complexité de l'algorithme du tri par insertion (on ne demande pas de justification).

> Le tri par insertion a une complexité quadratique.

b) On a mesuré qu'en utilisant l'algorithme du tri par insertion un ordinateur trie une liste de dix million d'éléments en 5 secondes. Quel est le temps prévisible approximatif pour trier une liste contenant un milliard d'éléments ?

> Le nombre d'élements est multiplié par 100 et l'algorithme ayant une complexité quadratique, le temps prévisible sera multiplé par $100^2 = 10\,000$. Donc l'exécution prendra environ 50 000 secondes (environ 14 heures).

c) On a mesuré qu'en utilisant l'algorithme du drapeau hollandais un ordinateur trie une liste de dix million d'éléments en 0.2 secondes. Quel est le temps prévisible approximatif pour un trier une liste contenant un milliard d'éléments ?

> Le nombre d'élements est multiplié par 100 et l'algorithme ayant une complexité linéaire, le temps prévisible sera multiplé lui aussi par 100. Donc l'exécution prendra environ 20 secondes.

❑ **Exercice 2** : *Manipulation de listes en OCaml*

On considère la fonction `est_triee` suivante :

```
1 let rec est_trie lst =
2   (* Renvoie true si lst est triée dans l'ordre croissant et false sinon*)
3   match lst with
4   | [] -> true
5   | h::[] -> true
6   | h::i::t -> if h<=i then est_trie t else false;;
```

1. Proposer au moins un test permettant de montrer que cette fonction ne répond pas à sa spécification donnée en commentaire dans le code.

> La fonction teste si les deux premiers éléments sont dans l'ordre croisant puisse passe au reste de la liste *à partir du 3e élément*, on ne vérifie donc pas que le 2e et le 3e élement sont bien dans l'ordre croissant. Le test suivant permettrait de faire apparaitre le problème : `est_triee [3; 4; 1; 2]`

2. Corriger le code de cette fonction afin qu'elle soit conforme à sa spécification (on pourra simplement indiquer le numéro de la ligne à modifier et donner son nouveau contenu).

> On doit relancer la récursivité sur la suite de la liste à partir du 2e élément dont sur `i::t`, ce qui donne :

```
1 let rec est_trie lst =
2   (* Renvoie true si lst est triée dans l'ordre croissant et false sinon*)
3   match lst with
4   | [] -> true
5   | h::[] -> true
6   | h::i::t -> if h<=i then est_trie (i::t) else false;;
```

3. Ecrire une fonction `fusion: int list -> int list -> int list` qui prend en argument deux listes *supposées triées* et renvoie la fusion de ces deux listes (avec répétition éventuelle des éléments). Par exemples :
   - `fusion [4; 8; 9] [0; 2; 3; 10]` renvoie `[0; 2; 3; 4; 8; 9; 10]`
   - `fusion [4; 4; 5; 7] [4; 6]` renvoie `[4; 4; 4; 5; 6; 7]`
   - `fusion [3] [1; 2; 2; 4; 6]` renvoie `[1; 2; 2; 3; 4; 6]`

```
let rec fusion l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | h1::t1, h2::t2 -> if h1<h2 then h1::(fusion t1 (h2::t2)) else  h2::(fusion
  ↪  (h1::t1) t2);;
```

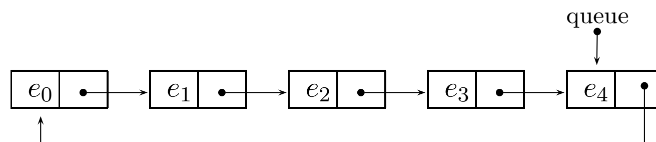
4. Prouver que la fonction `fusion` termine.

On note n_1 la longueur et `l1` et n_2 celle de `l2`, $n_1 + n_2$ est bien un entier positif comme somme de deux entiers positifs (des longueurs de listes). A chaque appel récursif, n_1 décrémente $n2$ ne change pas ou c'est l'inverse, dans les deux cas, n_1+n_2 décroît strictement. Par conséquent n_1+n_2 est un entier positif qui décroît strictement à chaque appel récursif donc cette fonction termine.

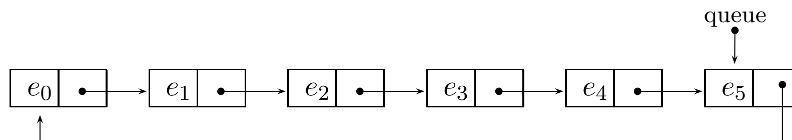
❑ **Exercice 3** : *Liste chaînée circulaire*

Le langage utilisé dans cet exercice est le langage C.

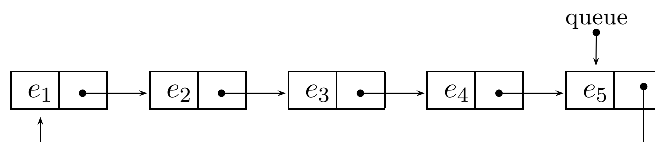
On veut implémenter une structure de données de liste chainée circulaire avec un pointeur sur la queue telle que représentée ci-dessous :



La queue pointe toujours vers *le dernier élément inséré* ainsi, après l'ajout d'un nouvel élément e_5, la structure de données ci-dessus devient :



Lorsqu'on retire un élément de cette structure de données, on retire le maillon qui *suit le pointeur de queue.* Par conséquent, si on retire un élément de la structure de donnée ci-dessus, c'est le maillon contenant e_0 qui est retiré et on obtient :



Afin d'implémenter cette structure de données, on propose d'utiliser les types suivants

```c
struct maillon_s
{
    int valeur;
    struct maillon_s *suivant;
};
typedef struct maillon_s maillon;
typedef maillon *liste_circulaire;
```

La liste chainée circulaire vide est alors représentée par le pointeur NULL.

1. En partant d'une liste chainée circulaire initialement vide, donner les étapes de son évolution après les opérations suivantes :
   1. `ajouter 12`
   2. `ajouter 6`
   3. `ajouter 7`
   4. `retirer`
   5. `ajouter 42`
   6. `retirer`

   On précisera la valeur des entiers renvoyés par la fonction `retirer`.

   > 1. `ajouter 12` : 12
   > 2. `ajouter 6` : 12 → 6
   > 3. `ajouter 7` : 12 → 6 → 7
   > 4. `retirer` : 6 → 7, l'opération renvoie 12
   > 5. `ajouter 42` : 6 → 7 →42
   > 6. `retirer` : 7 →42, l'opération renvoie 6

2. Ecrire la fonction `ajouter` de signature `void ajouter(liste_circulaire *lc, int v)` qui modifie la liste circulaire donnée en argument en lui ajoutant un nouveau maillon contenant la valeur `v`.
   Indication : on fera attention à traiter le cas particulier d'une liste circulaire initialement vide.

```c
void ajouter(liste_circulaire *lc, int v)
{
    maillon *nm = malloc(sizeof(maillon));
    nm->valeur = v;
    if (est_vide(*lc))
    {
        nm->suivant = nm;
    }
    else
    {
        nm->suivant = (*lc)->suivant;
        (*lc)->suivant = nm;
    }
    *lc = nm;
}
```

3. Ecrire la fonction `retirer` de signature `int retirer(liste_circulaire *lc)` qui prend en argument une liste circulaire *supposée non vide* et renvoie la valeur du maillon situé après le pointeur de queue en le retirant de la liste circulaire.

```c
int retirer(liste_circulaire *f)
{
    int res = ((*f)->suivant)->valeur;
    maillon *old = (*f)->suivant;
    if ((*f)->suivant == *f)
    {
        *f = NULL;
    }
    else
    {
        (*f)->suivant = ((*f)->suivant)->suivant;
    }
    free(old);
    return res;
}
```

4. Quelle structure de données connue a-t-on implémenté ici ? Justifier et proposer des noms plus appropriés pour les fonctions `ajouter` et `retirer`.

> Les premiers éléments ajoutés à la structure sont aussi les premiers à être retirés, il s'agit donc d'une struture de données de type FIFO, c'est à dire une file. L'opération ajouter correspond à enfiler et l'opération retirer à défiler.

5. Ecrire une fonction `longueur` de signature `int longueur(liste_circulaire lc)` qui renvoie le nombre d'éléments d'une liste chaînée circulaire.

```c
int longueur(liste_circulaire lc)
{
    if (est_vide(lc))
    {
        return 0;
    }
    else
    {
        liste_circulaire start = lc;
        int nb = 1;
        while (lc->suivant != start)
        {
            nb += 1;
            lc = lc->suivant;
        }
        return nb;
    }
}
```

6. Donner, en les justifiant, les complexités des opérations `retirer`, `ajouter` et `longueur`.

> Retirer et ajouter sont en $O(1)$ car on ne fait que des opérations élémentaires et longueur est en $O(n)$ où n est la longueur de la liste car celle ci doit être parcourue en entier (on détecte un retour vers le pointeur initial).

❑ **Exercice 4** : *Plus petit entier manquant*
Le langage utilisé dans cet exercice est le langage Ocaml.

Etant donné une liste d'entiers *naturels*, on cherche à déterminer le plus petit entier manquant dans cette liste. Par exemples :

— si la liste est `[6; 4; 2; 0; 1; 2; 5]` alors le plus petit entier manquant est 3

— si la liste est `[0; 2; 1; 1; 3; 2; 1]` alors le plus petit entier manquant est 4

— si la liste est `[1; 3; 7;]` alors le plus petit entier manquant est 0

— si la liste est vide alors le plus petit entier manquant est 0.

Dans toute la suite de l'exercice, on notera N la longueur de la liste donnée en argument et M le maximum de ses éléments.

1. Dans cette partie, on propose de répondre au problème en testant successivement la présence de chaque entier dans la liste.

   a) Ecrire une fonction `est_dans 'a -> 'a list -> bool` qui prend en entrée un élément `elt` et une liste `lst` et renvoie `true` lorsque `elt` est dans `lst` et `false` sinon.

```
let rec est_dans elt lst =
  (* Le cas de base est la liste vide, sinon elt est dans lst
  lorsque elt est la tête de lst ou qu'il est dans la queue de lst*)
  match lst with
  | [] -> false
  | h::t -> h=elt || est_dans elt t;;
```

   b) Ecrire une fonction `manquant: intlist -> int` qui renvoie le plus petit entier manquant dans la liste donnée en argument en testant successivement la présence des entiers $0, 1, 2, \dots$ et en s'arrêtant dès que l'un deux n'est pas trouvé.
   Indication : on pourra utiliser une fonction auxiliaire récursive.

   c) Donner en la justifiant la complexité de la fonction `manquant`.

   La complexité de `est_dans` est un $O(N)$ et on appelle cette fonction au plus N fois, donc la fonction `manquant` a une complexité quadratique $\mathcal{O}(N^2)$

2. Dans cette partie, on propose de trier au préalable la liste d'entiers. On suppose *déjà écrite* une fonction `trie int list -> int list` qui prend en argument une liste d'entiers et renvoie cette liste triée.

   a) Ecrire une fonction `manquant_trie : int list -> int` qui prend en argument une liste triée d'entiers et renvoie le plus petit entier manquant dans cette liste.

```
let manquant_trie lst =
  let rec aux lst n =
    match lst with
    | [] -> n
    | h::t -> if h=n-1 then aux t n
              else if h=n then aux t (n+1)
              else n in
    aux lst 0;;
```

   b) Citer au moins un algorithme permettant de trier une liste avec une complexité en $\mathcal{O}(N \log N)$ où N est la longueur de la liste.

   Le tri fusion a une complexité linéarithmique (on pouvait aussi citer le tri rapide).

   c) On suppose que la fonction `trie` a une complexité linéarithmique, donner alors en la justifiant la complexité de la méthode consistant à trier au préalable la liste avec la fonction `trie` puis à utiliser la fonction `manquant_trie`.

Le tri a une complexité linéarithmique par hypothèse, on appelle ensuite la fonction `manquant_trie` qui a une complexité linéaire en la longueur N de la liste (la fonction `aux` effectue au plus N appels récursifs et ne contient que des opérations élémentaires). Comme $\mathcal{O}(N \log N) + \mathcal{O}N$ est un $\mathcal{O}(N \log N)$ la fonction `manquant_trie` a une complexité linéarithmique.

3. Proposer un algorithme permettant de résoudre le problème de la recherche du plus petit entier manquant en complexité $\mathcal{O}(M)$ où M est le maximum des éléments de la liste. On *ne demande pas de coder* cet algorithme, mais simplement d'en décrire le fonctionnement (ou de l'écrire en pseudo-langage) et d'en justifier la complexité.
   Indication : on pourra utiliser un tableau de booléens de taille M.

   On propose l'algorithme suivant :

   1. on détermine le maximum M des éléments de la liste, pour cela un parcours des éléments de la liste suffit, la complexité est donc en $\mathcal{O}(N)$ où N est la longueur de la liste.
   2. on crée un tableau de `tab` de $M + 1$ booléens qu'on initialise à `Faux`, il faut pour cela parcourir la totalité du tableau, cette étape est donc en $\mathcal{O}(M)$
   3. on parcourt de nouveau la liste, et pour chaque élément `x` trouvé on affecte `tab[x]` à `Vrai`. Cette nouvelle étape est en $\mathcal{O}(N)$.
   4. on parcourt le tableau par indice croissant et on renvoie l'indice du premier `Faux` rencontré, cette étape est donc en $\mathcal{O}(M)$.