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 : *Tris*

On rappelle ci-dessous en pseudo-langage, l'algorithme du tri par sélection (aussi appelé tri par recherche itérée du minimum)

Algorithme : Tri par sélection

Entrées : tableau $tab = \{t_0, \dots, t_{n-1}\}$ de n élements

Sorties : aucune, le tableau tab est modifié et trié en place

```
pour i ← 0 à n − 1 faire
    i_m ← indice du minimum du sous tableau {t_i, ... t_{n-1}}
    échanger dans t les éléments d'indice i et i_m
fin
```

Q1– En recopiant et complétant le tableau suivant. Donner l'état des variables de l'algorithme lorsqu'on utilise cet algorithme pour trier le tableau `{2, -5, -4, 4, 1}`

	i_m	tab
après un tour de boucle	1	{-5, 2, -4, 4, 1}
après deux tours de boucle	2	{-5, -4, 2, 4, 1}
après trois tours de boucle	4	{-5, -4, 1, 4, 2}
après quatre tours de boucle	4	{-5, -4, 1, 2, 4}

.

Q2– Ecrire en langage C une fonction de signature `indice_min_depuis(int tab[], int n, int ind)` qui renvoie l'indice d'une occurrence du minimum du tableau `tab` de taille `n` à partir de l'indice `ind` (inclus). On vérifiera les conditions sur `ind` à l'aide d'instructions `assert`.

```c
int indice_min_depuis(int tab[], int taille, int ind)
{
    assert(ind < taille && ind >= 0);
    int vmin = tab[ind];
    int imin = ind;
    for (int i = ind; i < taille; i++)
    {
        if (tab[i] < vmin)
        {
            imin = i;
            vmin = tab[i];
        }
    }
    return imin;
}
```

Q3– En langage C, on suppose *déjà écrite* une fonction de signature `void echange(int tab[], int i, int j)` qui échange les éléments situés aux indices `i` et `j` dans le tableau `tab`. Ecrire une fonction de signature

`tri_selection(int tab[], int n)` qui tri le tableau `tab` de taille `n` en utilisant l'algorithme du tri par sélection.

```
void tri_selection(int tab[], int size)
{
    int im;
    for (int i = 0; i < size; i++)
    {
        im = indice_min_depuis(tab, size, i);
        echange(tab, i, im, size);
    }
}
```

Q4– Montrer que l'algorithme du tri par sélection a une complexité quadratique en fonction de la taille n du tableau.

La fonction de recherche du minimum a une complexité en $\mathcal{O}(n)$ car la boucle de recherche s'exécute au plus n fois et elle ne contient que des opérations élémentaires. Le tri appelle n fois la fonction de recherche du minimum et donc a une complexité en $n\mathcal{O}(n) = \mathcal{O}(n^2)$.

Q5– On suppose qu'on a trié un tableau de taille 500 000 en 1.2 secondes avec l'algorithme du tri par sélection, donner une estimation du temps nécessaire pour trier un tableau contenant un million d'éléments.

La taille du tableau est doublé et la complexité est quadratique donc le temps de calcul sera approximativement multiplié par 4, on doit donc s'attendre à un temps de calcul d'environ 4.8 secondes.

Q6– Ecrire en OCaml une fonction `separe : int list -> int list* int list` qui prend en argument une liste d'entiers `lst` et renvoie un couple de deux listes contenant chacune une moitié des éléments de `lst` (a une unité près). Par exemple `separe [2; 8; 1; 4; 5]` peut renvoyer le couple `[2; 1; 5], [8; 4]`.

```
let rec separe l =
  match l with
  |[] -> [], []
  |h::[] -> [h],[]
  |h1::h2::t  -> let l1,l2 = separe t in h1::l1, h2::l2;;
```

Q7– Ecrire en OCaml une fonction `fusion : int list -> int list -> int list` qui prend en argument deux listes d'entiers triées et renvoie la liste triée de leur fusion. Par exemple `fusion [1; 2; 5] [4; 8]` renvoie la liste `[1; 2; 4; 5; 8]`.

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

Q8– On rappelle que l'algorithme du tri fusion est un algorithme de tri récursif qui consiste en séparer en deux la liste des éléments à trier, à trier récursivement chacune de ces listes puis à les fusionner. Déduire des questions précédentes une fonction `tri_fusion : int list -> int list` qui prend en argument une liste d'entiers et renvoie cette liste triée.

```
let rec tri_fusion l =
  match l with
  | [] -> []
  | h::[] -> [h]
  | _ -> let l1,l2 = separe l in
    let tl1 = tri_fusion l1 in
    let tl2 = tri_fusion l2 in
    fusion tl1 tl2;;
```

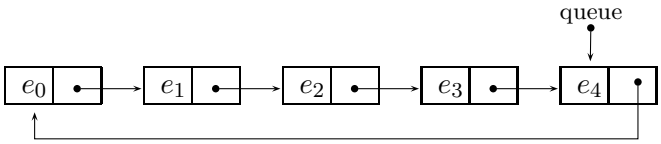
Q9– On admet qu'il faut au plus n comparaisons pour fusionner deux listes dont la somme des longueurs est n. On cherche à déterminer la complexité du tri fusion en *nombre de comparaison effectuées* et on note $C(n)$ le nombre de comparaisons afin de trier une liste de longueur n. Déterminer $C(0), C(1)$ et justifier rapidement que $C(n) = 2C\left(\frac{n}{2}\right) + n$.

Q10– Déduire de la relation précédente que pour tout $i \in \mathbb{N}^*$, $C(n) = 2^i C(\frac{n}{2^i}) + in$ et en déduire la complexité du tri fusion en nombre de comparaisons.

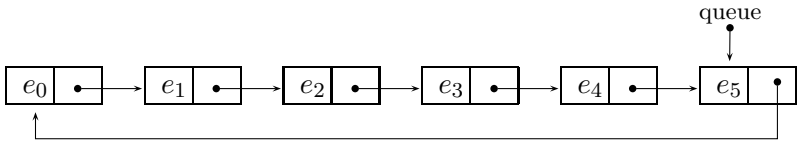
❑ **Exercice 2** : *Liste chaînée circulaire*

Le langage utilisé dans cet exercice est le langage C.

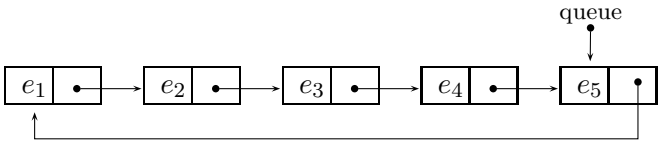
On veut implémenter une structure de données de liste chainée circulaire avec un pointeur sur la queue telle que représentée ci-dessous :



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

```
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.

Q11– En partant d'une liste chainée circulaire initialement vide, donner les étapes de son évolution (en schématisant l'état de la liste), 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

Q12– 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;
}
```

Q13– 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. Une solution entièrement correcte devra libérer l'espace mémoire alloué au maillon que l'on supprime.

```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;
}
```

Q14– 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.

Q15– 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;
    }
}
```

Q16– 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 3 : *dépouillement d'un vote*

Le langage utilisé dans cet exercice est OCaml. On s'intéresse à la comptabilisation des résultats d'une élection.

Chaque candidat est désigné par un caractère (`'A'`, `'B'`, ...) et on suppose qu'on dispose des votes sous la forme d'une liste de caractère, par exemple la liste `['A'; 'C'; 'B'; 'B'; 'A'; 'A']` signifie qu'il y a eu 6 votants, le candidat `'A'` a obtenu 3 voix, le `'B'` deux voix et le `'C'` une voix. On souhaite dans un premier temps à partir de la liste des votes construire la liste des couples (candidat, nombre de votes), c'est à dire que si la liste des votes est `['A'; 'C'; 'B'; 'B'; 'A'; 'A']` alors on veut construire la liste `[('A',3); ('C',1); ('B',2)]`.

Q17– Ecrire une fonction `appartient : char -> char list -> bool` qui prend en argument un caractère `c` et une liste de caractères `lst` et qui renvoie `true` si et seulement si `c` est dans `lst`.

```
let rec appartient c votes =
  match votes with
  | [] -> false
  | h::t -> h=c || appartient c t;;
```

Q18– En déduire une fonction `candidats : char list -> char list` qui prend en argument une liste `votes` et renvoie cette liste sans aucun doublon, par exemple `candidats ['A'; 'C'; 'B'; 'B'; 'A'; 'A']` renvoie la liste `['A', 'C', 'B']`.

```
let rec candidats votes =
  match votes with
  | [] -> []
  | h::t -> let cr = candidats t in if appartient h cr then cr else h::cr;;
```

Q19– Ecrire une fonction `nb_votes : char -> char list -> int` qui prend en argument un caractère `c` et une liste de caractères `votes` et renvoie le nombre d'apparition de `c` dans `votes`. Par exemple `nb_votes 'A' ['A'; 'C'; 'B'; 'B'; 'A'; 'A']` renvoie 3.

```
let rec nb_votes c votes =
  match votes with
  | [] -> 0
  | h::t -> if h=c then 1 + nb_votes c t else nb_votes c t;;
```

Q20– En déduire une fonction `comptabilise : char list -> (char*int) list` qui répond au problème posé, c'est à dire qu'elle prend en argument une liste de caractères `votes` et renvoie une liste de couples `(char*int)` de chacun des caractères apparaissant dans `votes` et de son nombre d'occurrences dans `votes`.

```
let comptabilise votes =
  let cd = candidats votes in
  List.map (fun c -> (c,nb_votes c votes)) cd;;
```

Q21– On veut maintenant écrire une version de la fonction `comptabilise` qui parcourt une seule fois la liste des votes. Pour cela on veut écrire une fonction `ajoute : char -> char*int list -> char*int list` qui prend en argument un caractère `un_vote`, une liste de couples `(char*int) list` qui représente un résultat partiel du dépouillement des votes et renvoie le nouveau décompte des voix après prise en compte de `un_vote` . Par exemple :

- `ajoute 'B' [('A',2); ('C',1)]` renvoie `[('A',2); ('C',1); ('B', 1)]` (premier vote obtenu par le candidat `'B'`)
- `ajoute 'C' [('A',2); ('C',1)]` renvoie `[('A',2); ('C',2)]` (incrémente le nombre de vote du candidat `'C'`)

. Ecrire la fonction `ajoute`.

```
let rec ajoute un_vote resultat =
  match resultat with
  | [] -> [(un_vote, 1)]
  | (v,r)::t -> if v=un_vote then (v, r+1)::t else (v,r)::(ajoute un_vote t)
```

Q22– En déduire une nouvelle version de la fonction `comptabilise` qui parcourt une seule fois la liste des votes.

```
let rec comptabilise2 votes acc =
  match votes with
  | [] -> acc
  | h::t -> comptabilise2 t (ajoute h acc);;
```

❑ **Exercice 4** : *Représentation d'ensemble d'entiers*

En OCaml, on propose de représenter un ensemble d'entiers, par la liste *triée* (dans l'ordre croissant) de ses éléments. Par exemple l'ensemble `{7; 2; 5; 3}` sera représenté par la liste `[2; 3; 5; 7]`. Par contre, les listes `[2; 3; 3; 5; 7]` (élément en double) ou `[2; 5; 3; 7]` (non triée) ne représentent pas correctement un ensemble. La liste vide `[]` représente l'ensemble vide.

Q23– Ecrire une fonction `est_ensemble: int list -> bool` qui renvoie `true` lorsque la liste d'entier fournie en argument représente correctement un ensemble. Par exemple `est_ensemble [2; 8; 1]` doit renvoyer `false` car la liste n'est pas triée dans l'ordre croissant et `est_ensemble [1; 2; 7; 7; 10]` doit aussi renvoyer `false` car 7 est présent deux fois.

```
let rec est_ensemble l =
  match l with
  | [] -> true
  | h::[] -> true
  | h1::h2::t -> h1<h2 && est_ensemble (h2::t);;
```

Q24– Ecrire une fonction `intersection : int list -> int list -> int list` qui prend en argument deux listes d'entiers (en supposant que ces deux listes représentent correctement des ensembles) et renvoie la liste d'entiers représentant l'intersection des deux ensembles représentés par ces listes. Par exemple `intersection [2; 5; 7] [5; 6; 7; 10];;` renvoie `[5; 7]`.

```
let rec intersection ens1 ens2 =
  match ens1, ens2 with
  | [], ens2 -> []
  | ens1, [] -> []
  | e1::r1, e2::r2 -> if e1=e2 then e1::(intersection r1 r2) else
    if e1<e2 then intersection r1 ens2 else
      intersection r2 ens1;;
```

Q25– *Sans les écrire*, donner les complexités des fonctions suivantes en les justifiant brièvement :

— `minimum : int list -> int` qui renvoie le plus petit élément d'un ensemble représenté par une liste d'entiers (on suppose l'ensemble non vide)

— `maximum : int list -> int` qui renvoie le plus grand élément d'un ensemble représenté par une liste d'entiers (on suppose l'ensemble non vide)

> La complexité serait linéaire car dans le pire des cas, on doit parcourir toute la liste afin de savoir si l'élément cherché s'y trouve ou non.

On souhaite maintenant, de façon similaire, représenter un ensemble d'entier en langage C par le tableau trié de ses éléments. Cette structure de donnée est *non mutable* c'est à dire que si on souhaite par exemple ajouter un élément à un ensemble alors on crée un nouvel ensemble.

Q26– Proposer un type structuré `ensemble` en C qui réalise cette représentation.

```c
struct ensemble_s
{
    int cardinal;
    int *elements;
};
typedef struct ensemble_s ensemble;
```

Q27– Donner la complexité des fonctions de signature `int minimum(ensemble ens)` et `int maximum(ensemble ens)` qui renvoie respectivement le minimum et le maximum de l'ensemble `ens` donné en argument. On ne *demande pas* d'écrire ces fonctions.

Ces fonctions renvoient respectivement le premier et le dernier élément d'un tableau, l'accès aux éléments d'un tableau étant en $\mathcal{O}(1)$, ces fonctions ont une complexité en $\mathcal{O}(1)$.

Q28– Ecrire la fonction de signature `bool appartient(ensemble ens, int elt)` qui renvoie `true` si et seulement si `elt` appartient à `ens`. Cette fonction *doit procéder* en parcourant un à un les éléments du tableau représentant l'ensemble. On donnera la complexité de cette fonction (aucune justification n'est demandée).

```c
bool appartient(ensemble ens, int elt)
{
    for (int i = 0; i < ens.cardinal; i++)
    {
        if (ens.elements[i] == elt)
        {
            return true;
        }
    }
    return false;
}
```

Cette fonction est de complexité linéaire en la taille de l'ensemble.

Q29– Donner en pseudo-langage un algorithme vu en cours et qui permettrait d'écrire une version de complexité logarithmique de la fonction `appartient`. Cette version doit s'appuyer sur le fait que les éléments du tableau sont triés.

Algorithme : Recherche dichotomique

Entrées : Ensemble E représenté par le tableau trié $tab = \{t_0, \ldots, t_{n-1}\}$ de ses éléments et un entier x.

Sorties : Un booléen indiquant si x appartient à E.

```
1  a ← 0
2  b ← n − 1
3  while b − a ⩾ 0 do
4  |  m ← (a + b)/2
5  |  if tab[m] = x then
6  |  |  return true
7  |  end
8  |  else
9  |  |  if tab[m] > x then
10 |  |  |  b ← m − 1
11 |  |  end
12 |  |  else
13 |  |  |  a ← m + 1
14 |  |  end
15 |  end
16 end
17 return false
```

Q30– Proposer une implémentation de cet algorithme en langage C.

```c
bool dichotomie(ensemble ens, int elt)
{
    int a = 0;
    int b = ens.cardinal - 1;
    int m;
    while (b - a >= 0)
    {
        m = (a + b) / 2;
        if (ens.elements[m] == elt)
        {
            return true;
        }
        else
        {
            if (ens.elements[m] > elt)
            {
                b = m - 1;
            }
            else
            {
                a = m + 1;
            }
        }
    }
    return false;
}
```