Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C et on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.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 : *Algorithme du drapeau hollandais*

> « *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
// 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;}
        }
    }
}

// Génère un tableau aléatoire de 1, 2, 3
```

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 ».

On note k le nombre de tours de boucle effectués. Montrons par récurrence sur k la propriété $P(k)$= « Après k tours de boucle P_1 ne contient que des 1, P_2 que des 2 et P_3 que des 3 »

- Initialisation : si $k = 0$ alors les parties P_1, P_2 et P_3 sont vides et donc $P(0)$ est vraie.
- Hérédité : soit $k \in \mathbb{N}$ tel que $P(k)$ soit vraie, 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 $P1$ qui par hypothèse de récurrence ne contenait que des 1. Ni $P2$ ni $P3$ ne sont modifiés et donc dans ce cas $P(k+1)$ est vérifiée.
  - 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 de recurrence ne contenait que des 2. Ni $P1$, ni $P3$ ne sont modifiés et donc $P(k+1)$ est encore vérifiée.
  - Si `tab[i1]` vaut 3, alors ce 3 est placé à l'indice `i3`, La partie $P2$ est décalée d'un rang à gauche (elle garde la même longueur) donc $i2$ et $i3$ sont décrémentés. Un 3 étant ajouté à P3 qui par hypothèse de récurrence ne contenait que des 3, l'invariant est préservé.

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

d) Justifier brièvement que l'algorithme du drapeau hollandais a une complexité temporelle linéaire.

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 (sans justification), la complexité de l'algorithme du tri par insertion.

   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 : *Représentation des ensembles d'entiers*

En OCaml, on représente une partie de $\mathbb{N}$ par la liste **triée** de ses éléments. Par exemple l'ensemble $\{2, 9, 1, 6, 8\}$ sera représenté par la liste `[1; 2; 6; 8; 9]`. Le but de l'exercice est d'étudier deux méthodes permettant de calculer l'union de deux ensembles ainsi représentés. Dans l'étude de la complexité, on notera n_1 la longueur de la première liste et n_2 celle de la seconde.

1. Union en ajoutant chaque élément successivement
   a) Ecrire une fonction `ajoute int -> int list -> int list` qui prend en argument un entier `n`, une liste triée `l` et insère `elt` dans `l` si `elt` n'y figure pas déjà et sinon ne fait rien. Par exemples :
   — `insere 3 [2; 6; 7]` donne `[2; 3; 6; 7]`,
   — `insere 4 [1; 4; 5]` donne `[1; 4; 5]`.

```
let rec insere n l =
  match l with
  | [] -> [n]
  | h::t -> if h<n then h::(insere n t) else
            if h=n then h::t else n::l;;
```

   b) Ecrire une fonction `union` qui calcule l'union de deux listes en insérant successivement chacun des éléments de la première liste dans la seconde à l'aide de la fonction écrite à la question précédente.

```
let rec union l1 l2 =
  match l1 with
  | [] -> l2
  | h::t -> union t (insere h l2);;
```

   c) Déterminer la complexité temporelle en fonction de n_1 et n_2 dans le pire des cas de cette fonction.

   La fonction `union` fait n_1 insertion dans `l2`, or une insertion dans `l2` a une complexité maximale en $O(n_1 + n_2)$ (puisque la taille de l2 augmente à chaque insertion), donc la complexité de la fonction `union` est un $O(n_1(n_1 + n_2))$.

2. Deuxième méthode

a) Recopier et compléter la fonction ci-dessous (pointillés des lignes 3, 4 et 5) qui à l'aide d'une correspondance de motifs sur les deux listes calcule directement leur union :

```
1  let rec union l1 l2 =
2    match l1, l2 with
3    | l1, [] -> l1
4    | [], l2 -> l2
5    | h1::t1, h2::t2 -> if h1<h2 then h1::(union t1 l2) else
6                        if h1=h2 then h1::(union t1 t2) else
7                        h2::(union l1 t2);;
```

b) Prouver que cette fonction termine.

On note n_1 la longueur de `l1` et n_2 celle de `l2`. Montrons que $n_1 + n_2$ est un variant :

(H1) $n_1 + n_2$ est un entier comme somme de deux entiers

(H2) $n_1 + n_2$ ne prend que des valeurs positives car c'est la somme de deux entiers positifs

(H3) A chaque appel récursif, $n_1 + n_2$ décroit strictement, car n_1 diminue de 1 ou n_2 diminue de 1 (ou non exclusif car dans le cas ou les deux listes commencent par le même élément, la taille des deux décroît lors de l'appel suivant).

$n_1 + n_2$ est donc bien un variant et donc la fonction termine.

c) Donner sa complexité temporelle en fonction de n_1 et n_2.

Dans le pire des cas (celui ou les deux listes n'ont aucun élément commun) la taille de l'une ou de l'autre diminue de 1 à chaque appel récursif, la complexité est en $O(n_1 + n_2)$.

❑ **Exercice 3** : *Evaluation d'un polynôme par la méthode de Horner*

En Ocaml, on représente un polynôme par la liste de ses coefficients (le coefficient de plus haut degré en premier). On suppose dans toute la suite de l'exercice qu'il s'agit de coefficients entiers. Par exemple le polynôme $x^2 - 11x + 30$ est représenté par la liste `[1, -11, 30]`. D'autre part on donne la fonction `puissance : int -> int -> int` ci-dessous qui prend en argument deux entiers naturels `a` et `n` et calcule `a` puissance `n`

```
1  (* Calcule a puissance n pour a et n entiers naturels*)
2  let rec puissance a n =
3    if n=0 then 1 else a * (puissance a (n-1));;
```

1. Evaluation naïve

a) Justifier rapidement qu'avec la fonction `puissance`, le calcul de a^n demande n multiplications.

Une récurrence rapide permet de prouver que le calcul de a^n demande n multiplications. En effet le calcul de a^0 ne demande pas de multiplications et celui de a^n en demande 1 de plus que celui de a^{n-1}.

b) On donne ci-dessous la fonction `naif` permettant de calculer un polynôme en `x` en donnant la liste de ses coefficients `lcoeff` :

```
1  (* Calcule la valeur en x du  du polynome de coefficient lcoeff*)
2    let rec naif lcoeff x =
3      match lcoeff with
4      | [] -> 0
5      | ak::t-> ak*puissance x (List.length t) + (naif t x );;
```

La fonction `naif` utilise la fonction `puissance` donnée en début d'exercice.
Justifier rapidement la terminaison de cette fonction.

> La taille de la liste `lcoeff` est un variant car elle décroît de 1 à chaque itération et est bien entière et positive.

c) En utilisant l'identité mathématique $\sum_{k=0}^{n} a_k x^k = a^n x^n + \sum_{k=0}^{n-1} a_k x^k$, prouver la correction de la fonction `naif`.

> L'appel récursif est la traduction de l'égalité mathématique $\sum_{k=0}^{n} a_k x^k = a^n x^n + \sum_{k=0}^{n-1} a_k x^k$ et la condition d'arrêt traduit que la somme sur un ensemble vide est 0. Donc cette fonction est correcte.

d) Déterminer la complexité la fonction `naif`.
🛟 *Indication : on pourra noter $C(n)$ le nombre de multiplications pour une liste de n coefficients puis établir que $C(n) = C(n-1) + n$ (utiliser le résultat de la question 1.a)*

> Si la liste contient n coefficients alors on doit faire $n-1$ multiplications (pour le calcul de x^{n-1}) plus une multiplication (pour multiplier par le coefficient a_k.) l'appel récursif va lui demander $C(n-1)$ multiplications. Donc on a $C(n) = C(n-1) + n$ comme de plus $C(0) = 0$, on montre par une récurrence immédiate que $C(n) = \dfrac{n(n+1)}{2}$. Donc la fonction `naif` a une complexité quadratique.

2. Méthode de Horner

a) Montrer que $\sum_{k=0}^{n} a_k x^k = a_0 + x \times (a_1 + x \times (\cdots + a_n))$

> $$\begin{aligned}\sum_{k=0}^{n} a_k x^k &= a_0 + x\left(\sum_{k=1}^{n} a_k x^{k-1}\right)\\ &= a_0 + x \times \left(a_1 + x\left(\sum_{k=1}^{n} a_k x^{k-1}\right)\right)\\ &= \dots\\ &= a_0 + x \times (a_1 + x \times (\cdots + a_n))\end{aligned}$$

b) En déduire un algorithme récursif pour calculer la valeur d'un polynôme sans calculer expliciterment les puissances de la variable.

> Pour calculer le polynome de coefficient $a_n, \dots, a_0$ en x alors on calcule $a_0 + x$ fois le polynôme de coefficient $a_n, \dots a_1$.

c) Ecrire une implémentation en OCaml de cet algorithme.
⚠ Dans la fonction OCaml suivante on doit donner les coefficients dans l'ordre inverse (`a0` est le premier élément de la liste.)

```
let rec horner lcoeff x =
    match lcoeff with
    | [] -> 0
    | ak::t -> ak + x*(horner t x);;
```

d) Justifier rapidement que cet algorithme a une complexité linéaire.

A chaque appel récursif la taille de la liste de coefficient diminue de 1, comme on effectue lors de chaque appel que des opéarations élémentaires, la complexité est linéaire.

3. **Bonus** : Rappeler (sans justification) la complexité de l'algorithme d'exponentiation rapide. Quelle sera la complexité de la fonction de la question 1.b si on utilise l'exponentiation rapide pour le calcul des puissances ?

   L'exponentiation rapide a une complexité logarithmique, donc dans ce cas la fonction de la question 1.b vérifira : $C(0) = 0$ et $C(n) = C(n-1) + \log(n)$. Il vient $C(n) \leq n\log(n)$ et donc l'algorithme aurait une complexité linéarithmique.