Devoir Non Surveillé 3

À rendre le 11 mars 2024

INFORMATIQUE MP2I

Tournez la page S.V.P.

1 Un peu de géométrie (types énumérés en OCaml)

On s'intéresse à des ensembles de points du plan $\mathbb{R}^2$. On se donne les types suivants en OCaml pour les représenter :

```
type point = {x : float; y : float}
type vector = {x : float; y : float}
type set =
  | Set of point list (* Ensemble fini de points *)
  | Line of point * point (* Droite, donnée par deux de ses points *)
  | Circle of point * float (* Cercle, donné par son centre et son rayon *)
type cardinal =
  | Finite of int
  | Infinite
```

De plus, comme nous travaillons avec des nombres flottants, nous définissons une fonction qui nous permet de déterminer si deux nombres sont assez proches pour que l'on considère qu'ils soient égaux. On se fixe aussi une précision ε à cet effet.

```
let epsilon = 0.001

let is_eq (f1 : float) (f2 : float) =
  let df = f1 -. f2 in df > -. epsilon && df < epsilon
```

☐ 1 – Écrire une fonction `len (l : 'a list) : int` qui calcule la taille d'une liste, et en déduire une fonction `card (s : set) : cardinal` qui calcule le cardinal d'un ensemble de points.

☐ 2 – En supposant que le type int peut représenter tous les entiers sans dépassement de capacité, proposer une relation d'ordre totale $\leq_c$ telle que l'ensemble des éléments de type `cardinal` munit de $\leq_c$ est un ensemble bien fondé.

On souhaite à présent calculer l'intersection de deux ensembles de points du type `set`.

☐ 3 – Écrire une fonction `vect (a : point) (b : point) : vector` qui renvoie le vecteur $\overrightarrow{AB}$, et une fonction `det (u : vector) (v : vector) : float` qui calcule le déterminant de deux vecteurs $\vec{u}$ et $\vec{v}$, et en déduire une fonction `are_aligned (a : point) (b : point) (c : point) : bool` qui détermine si trois points A, B, et C sont alignés.

☐ 4 – En déduire une fonction `lines_intersect (a, b : point * point) (c, d : point * point) : set` qui calcule l'intersection des droites (AB) et (CD). On prendra soin de justifier la correction de la fonction en détaillant les calculs qui y ont mené.

☐ 5 – Écrire une fonction `filter (p : 'a -> bool) (l : 'a list) : 'a list` qui prend en entrée un prédicat p une liste ℓ et qui renvoie la liste des éléments x de ℓ vérifiant $p(x)$.

☐ 6 – En déduire une fonction `on_line (a, b : point * point) (l : point list) : point list` qui renvoie la liste des points de ℓ appartenant à la droite (AB),
et une fonction `common_elements (l1 : 'a list) (l2 : 'a list) : 'a list` qui renvoie la liste des éléments en qui sont à la fois dans ℓ_1 et dans ℓ_2. Quelle est la complexité de `common_elements` ?

☐ 7 – Écrire une fonction `dist (a : point) (b : point) : float` qui calcule la distance AB, et en déduire une fonction `on_circle (o, r : point * float) (l : point list) : point list` qui renvoie la liste des points de ℓ appartenant au cercle de centre O de rayon r.

☐ 8 – Écrire des fonctions pour les cas restants,
et en déduire une fonction `intersection (s1 : set) (s2 : set) : set` qui calcule $S_1 \cap S_2$. [1]
On prendra soin de justifier la correction de ces fonctions en détaillant les calculs qui y ont mené.

On souhaite à présent étendre notre type `set` pour inclure les unions de droites, ou les unions de cercles, ou les unions d'unions de droites et d'unions de cercles, etc.
On modifie donc à présent notre déclaration de type :

```
type set =
  | Set of point list (* Ensemble fini de points *)
  | Line of point * point (* Droite, donnée par deux de ses points *)
  | Circle of point * float (* Cercle, donné par son centre et son rayon *)
  | Union of (set * set) (* Union de deux ensembles de points *)
```

☐ 9 – Proposer un ajout à la fonction `cardinal` pour gérer le nouveau cas.

☐ 10 – Proposer un ajout à la fonction `intersection` pour gérer le nouveau cas. On pourra utiliser la distributivité de l'intersection par rapport à l'union.

1. Si vous n'arrivez pas à calculer l'intersection dans un ou plusieurs des cas restants, remplacez ce calcul par `assert false`

2 À cheval sur deux tableaux triés

Le but de cette partie est de déterminer le $k+1$-ième plus petit élément, dans deux tableaux triés.
Par exemple, si on considère les tableaux `{0, 2, 4, 6, 8}` et `{1, 3, 5, 7, 9}`, le 4-ième plus petit élément est 3. C'est-à-dire que si on regroupait les deux tableau en un tableau trié, il aurait la position 3. On ne vérifiera pas que les tableaux donnés en entrée sont effectivement triés. On suppose dans toute la suite que $k < n_1 + n_2$, sinon le problème n'a pas de solution.

☐ 11 – Écrire une fonction `int *fusion(int tab1[], int tab2[], int n1, int n2)` qui renvoie un tableau trié de taille $n_1 + n_2$ contenant les éléments de t_1 et t_2, où n_i est la longueur de t_i. On prendra soin d'allouer la mémoire dynamiquement (`malloc`).

☐ 12 – En déduire une fonction `int select_naif(int k, int tab1[], int tab2[], int n1, int n2)` qui détermine le $k+1$-ième plus petit élément de $t_1 \cup t_2$ en utilisant `fusion`. On prendra soin de libérer la mémoire allouée par `fusion`.

Pour éviter l'allocation et l'initialisation de ce tableau, on se propose d'utiliser le code suivant, manipulant le même tableau, mais sans le créer.

```
int select_lin(int k, int tab1[], int tab2[], int n1, int n2){
  assert(k < n1 + n2);                          // 1
  int i = 0, j = 0;                             // 2
  while (i + j < k) {                           // 3
    if (j >= n2 || tab1[i] < tab2[j])           // 4
      { i = i + 1; }                            // 5
    if (i >= n1 || tab1[i] >= tab2[j])          // 6
      { j = j + 1; }                            // 7
  }
  if (tab1[i] < tab2[j])                        // 8
    { return tab1[i]; }                         // 9
  else { return tab2[j]; }                      // 10
}
```

☐ 13 – Dessiner le graphe de flot de contrôle de la fonction `select_lin` et proposer un jeu de test couvrant l'ensemble de ses branches.

☐ 14 – Donner un variant permettant de montrer la terminaison de la boucle `while` et donc de la fonction `select_lin`. En déduire la complexité temporelle dans le pire des cas en utilisant la notation $O(\cdot)$.

☐ 15 – Démontrer à l'aide d'un invariant de la boucle `while` la correction partielle de la fonction `select_lin`. On rappelle que les tableaux t_1 et t_2 sont supposés triés.

On souhaite désormais trouver un algorithme plus efficace, utilisant mieux le fait que les tableaux soient triés. Soit i, tel que $0 \leq i$, $i < n_1$, $i \leq k$. Posons $j = k - i$ (remarquons que $j < n_2$ car $k < n_1 + n_2$).

☐ 16 – Montrer que si $t_1[i-1] \leq t_2[j]$ et $t_2[j-1] \leq t_1[i]$, alors le $(k+1)$-ième plus petit élément est $\min(t_1[i], t_2[j])$.

☐ 17 – Montrer que si $t_1[i-1] > t_2[j]$, alors le $(k+1)$-ième plus petit élément est plus petit que $t_1[i]$.

☐ 18 – Déduire des deux questions précédentes une fonction

`int select_log(int k, int tab1[], int tab2[], int n1, int n2)`

qui détermine le $(k+1)$-ième plus petit élément en temps $O(\log(n_1))$.

☐ 19 – Expliquer comment on pourrait résoudre le problème dans le pire des cas en $O(\log(\min(n_1, n_2)))$.

3 Introduction aux fonctions de hachages

Dans cette partie, on pose $\Sigma = [\![0, 255]\!]$ l'alphabet composé des valeurs de type `char`. On note Σ^* l'ensemble des chaînes de caractères, c'est-à-dire des suites finies d'éléments de Σ, ainsi que ε la chaîne de caractère vide `""`. Pour $u \in \Sigma^*$ et $a \in \Sigma$, on note $u \cdot a$ le mot composé des lettres de u suivi de la lettre a. On cherche à construire une fonction $h : \Sigma^* \mapsto \Sigma$ et à étudier ses propriétés.

☐ 20 – Dans un premier temps, nous posons h définit comme suit :

$$h(\varepsilon) = 0, \qquad h(u \cdot a) = (128 \times (h(u) + a)) \mod 256$$

Donner les valeurs possibles pour $h(u)$, avec $u \in \Sigma^*$.

On considère à présent la fonction h définit comme suit :

$$h(\varepsilon) = 0, \qquad h(u \cdot a) = (17 \times (h(u) + a)) \mod 256$$

☐ 21 – Écrire une fonction `int hash(char u[])` qui prend en entrée une chaîne de caractères u et qui renvoie $h(u)$.

☐ 22 – Montrer que 17 est inversible dans $\mathbb{Z}/256\mathbb{Z}$ et donner son inverse.

☐ 23 – Montrer que pour tout mot $u \in \Sigma^*$, si a est tiré aléatoire uniformément dans Σ, alors $h(u \cdot a)$ est distribué uniformément dans $[\![0, 255]\!]$.
Reformulation : montrer que pour toute valeur $x \in [\![0, 255]\!]$ et tout mot $u \in \Sigma^*$, il existe un unique a tel que $h(u \cdot a) = x$.

☐ 24 – Donner deux chaînes de caractères u et v distinctes ($u \neq v$) telles que $h(u) = h(v)$.

☐ 25 – Soit $u_0, u_1, \ldots, u_n$ $n+1$ lettres de Σ. Notons $u = \varepsilon \cdot u_0 \cdot u_1 \cdot \ldots \cdot u_{n-1}$ et $u' = \varepsilon \cdot u_1 \cdot u_2 \cdot \ldots \cdot u_n$.
Exprimez $h(u')$ en fonction de $h(u)$, u_0 et u_n.

☐ 26 – On note a^n le mot défini par :

$$a^0 = \varepsilon, \qquad a^{n+1} = a^n \cdot a$$

Déterminer en justifiant $\{h(a^n) \mid n \in \mathbb{Z}\}$ pour les valeurs de a suivantes :

1. $a = 0$ (remarque : impossible en C car le caractère 0 signe la fin de la chaîne de caractère)
2. $a = 1$
3. $a = 17$
4. a quelconque.

Fin de l'épreuve