MP2I
2021/2022

Informatique
Devoir surveillé 2
13 novembre 2021

CIV

1 Manipulation de sous-ensembles de $[\![0, n-1]\!]$

Q01

```
bool appartient(bool *S, int x)
{
    return S[x];
}
```

Q02

```
int cardinal(bool *S)
{
    int s = 0;

    for(int i = 0; i < n; i++)
    {
        if (S[i])
        {
            s++;
        }
    }

    return s;
}
```

Q03

```
void complementaire(bool *e, bool *ce)
{
    for(int i = 0; i < n; i++)
    {
        ce[i] = !e[i];
    }
}
```

Q04

Il n'y a pas de problèmes de terminaison ici dans la mesure où on est sur une boucle `for` effectuant un nombre fini d'itérations.
Pour la correction, on va montrer l'invariant suivant :

$$\mathrm{I}(\mathtt{i}) = \forall \mathtt{j} \in [\![0, \mathtt{i}-1]\!], \mathtt{ce[j]} = \mathtt{true} \iff \mathtt{j} \notin \mathtt{e}$$

en identifiant `e` et le sous-ensemble qu'il représente.

Initialisation Comme $[\![0, -1]\!] = \emptyset$, on a directement I(0).

Hérédité Si I(i) est vérifiée en début d'itération, en fin d'itération on passe à $\mathtt{i}+1$ et on a placé `true` dans `ce`[i] si et seulement si il y avait `false` dans `e`[i] et donc si et seulement si $\mathtt{i} \notin \mathtt{e}$, ce qui permet de prolonger l'invariant et d'obtenir I($\mathtt{i}+1$).

On a donc l'invariant I(n) en sortie de boucle et ainsi `ce` contient bien le complémentaire.
Remarque : Ici, la vérification est triviale, et c'était plus une question portant sur la méthode.

Q05

```
bool disjoint(bool *e1, bool *e2)
{
    bool dis = true;

    for(int i = 0; i < n; i++)
```

```
    {
        dis = dis && !(e1[i] && e2[i]);
    }

    return dis;
}
```

Attention : à l'écrit cela ne pose pas de problèmes mais on ne peut pas utiliser `union` en C car c'est un mot clé du langage.

Q06

```
void reunion(bool *e1, bool *e2, bool *ue)
{
    for(int i = 0; i < n; i++)
    {
        ue[i] = e1[i] || e2[i];
    }
}
```

Q07

```
void intersection(bool *e1, bool *e2, bool *ie)
{
    for(int i = 0; i < n; i++)
    {
        ie[i] = e1[i] && e2[i];
    }
}
```

Q08

```
void difference(bool *e1, bool *e2, bool *de)
{
    for(int i = 0; i < n; i++)
    {
        de[i] = e1[i] && !e2[i];
    }
}
```

Q09

```
int ensemble_vers_entier(bool *e)
{
    int u = 0;
    int p = 1;

    for(int i = 0; i < n; i++)
    {
        if (e[i])
        {
            u = u + p;
        }

        p = p * 2;
    }

    return u;
}
```

Q10

```
void entier_vers_ensemble(int u, bool *e)
{
    for(int i = 0; i < n; i++)
    {
        e[i] = u % 2 == 1;
```

```
            u = u / 2;
        }
    }
```

Q11

```
bool couverture(int *f, int p)
{
    bool couv = true;

    for(int i = 0; i < n; i++)
    {
        bool present = false;
        for(int j = 0; j < p; j++)
        {
            bool fj[n];
            entier_vers_ensemble(f[j], fj);
            if (fj[i])
            {
                present = true;
            }
        }

        couv = couv && present;
    }
```

On va montrer la correction de la boucle intérieure, comme elle termine directement, cela signifie de montrer qu'elle vérifie

$$\mathrm{C}(\mathtt{i}) = \mathtt{present} = \mathtt{true} \iff \exists \mathtt{k} \in [\![0, \mathtt{p}-1]\!], \mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$$

où $\mathrm{S}(\mathtt{k})$ est l'ensemble tel que $\mathtt{u}(\mathrm{S}(\mathtt{k})) = \mathtt{k}$.
Pour cela, on va prouver l'invariant

$$\mathrm{I}(\mathtt{j}) = \mathtt{present} = \mathtt{true} \iff \exists \mathtt{k} \in [\![0, \mathtt{j}-1]\!], \mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$$

Q12

Initialisation Encore une fois, c'est vide donc direct d'avoir $\mathrm{I}(0)$ car on a initialisé `present` à `false`.

Hérédité Si on a $\mathrm{I}(\mathtt{j})$ en début d'itération, deux cas : soit $\mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{j}])$ et on met `present` à `true`, soit $\mathtt{i} \notin \mathrm{S}(\mathtt{f}[\mathtt{j}])$ et on ne change pas sa valeur, comme par hypothèse elle valait $\mathtt{true} \iff \exists \mathtt{k} \in [\![0, \mathtt{j}-1]\!], \mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$ on a encore $\mathtt{true} \iff \exists \mathtt{k} \in [\![0, \mathtt{j}]\!], \mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$.
On a donc bien $\mathrm{I}(\mathtt{j}+1)$.

Ainsi $\mathrm{I}(\mathtt{p}) = \mathrm{C}(\mathtt{i})$ est vérifié.
On montre alors facilement que $\mathrm{I}'(\mathtt{i}) = \mathtt{couv} = \bigwedge_{\mathtt{l}=0}^{\mathtt{i}-1} \exists \mathtt{k} \in [\![0, \mathtt{p}-1]\!], \mathtt{l} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$ est un invariant car vu ce qu'on vient de montrer, le corps de boucle devient $\mathtt{couv} = \mathtt{couv} \wedge \exists \mathtt{k} \in [\![0, \mathtt{p}-1]\!], \mathtt{i} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$.
On a donc $\mathrm{I}'(\mathtt{n}) = \mathtt{couv} = \bigwedge_{\mathtt{l}=0}^{\mathtt{n}-1} \exists \mathtt{k} \in [\![0, \mathtt{p}-1]\!], \mathtt{l} \in \mathrm{S}(\mathtt{f}[\mathtt{k}])$ ce qui est effectivement la correction.

Q13

```
bool partition(int *f, int p)
{
    bool couv = true;

    for(int i = 0; i < n; i++)
    {
        int compte = 0;
        for(int j = 0; j < p; j++)
        {
            bool fj[n];
            entier_vers_ensemble(f[j], fj);
            if (fj[i])
            {
                compte++;
```

```
            }
        }

        couv = couv && compte == 1;
    }

    return couv;
}
```

Il suffit de reprendre rapidement la preuve précédente en remplaçant l'invariant de la boucle intérieure par

$$I(j) = \texttt{compte} = |\{\ k \in [\![0, j-1]\!] \mid i \in S(f[k])\ \}|$$

Q14 ce qui permet d'obtenir la correction de la boucle intérieure

$$C(i) = \texttt{compte} = |\{\ k \in [\![0, p-1]\!] \mid i \in S(f[k])\ \}|$$

et donc $\texttt{compte} = 1 \iff$ i appartient à un unique sous-ensemble.
On conclut par le même invariant que précédemment, vu que c'est à nouveau une conjonction.

2 Points fixes de fonctions à domaine fini

2.1 Recherche de point fixe : cas général

Q15

```
bool admet_point_fixe(int *t)
{
    for(int i = 0; i < n; i++)
    {
        if(t[i] == i)
        {
            return true;
        }
    }
    return false;
}
```

Q16

```
int nb_points_fixes(int *t)
{
    int nb = 0;

    for(int i = 0; i < n; i++)
    {
        if(t[i] == i)
        {
            nb = nb + 1;
        }
    }

    return nb;
}
```

Q17

```
int itere(int *t, int x, int k)
{
    for(int i = 0; i < k; i++)
    {
        x = t[x];
    }
```

```
    return x;
}
```

Q18

```
int nb_points_fixes_iteres(int *t, int k)
{
    int nb = 0;

    for(int i = 0; i < n; i++)
    {
        if(itere(t, i, k) == i)
        {
            nb = nb + 1;
        }
    }

    return nb;
}
```

Q19

```
bool admet_attracteur_principal(int *t)
{
    for(int z = 0; z < n; z++)
    {
        bool attracteur = true;

        for(int x = 0; z < n; z++)
        {
            /* f^0(x), f^1(x), ..., f^n(x) prennent n+1 valeurs
             * dans 0, 1, ..., n-1 qui compte n valeurs distinctes
             * donc on a deux valeurs égales. Or, s'il existe
             * k<l tels que f^k(x) = f^l(x) et qu'on n'a pas vu
             * z, on va rester bloqué sur les memes valeurs à partir
             * de k, donc on ne verra jamais z. Si au contraire, z apparait
             * alors on reste sur z car c'est un point fixe.
             * Dans tous les cas, soit z n'est pas accessible, soit f^n(x) = z
             */
            attracteur = attracteur && itere(t, x, n) == z;
        }

        if (attracteur)
        {
            return true;
        }
    }

    return false;
}
```

Q20

```
int temps_de_convergence(int *t, int x)
{
    if (t[x] == x)
        return 0;
    return 1 + temps_de_convergence(t, t[x]);
}
```

Q21

```
/* Version naive et quadratique */
int temps_de_convergence_max_n2(int *t)
```

```
{
    int m = 0; // atteint pour l'attracteur

    for(int x = 0; x < n; x++)
    {
        int tx = temps_de_convergence(t, x);
        if(tx > m)
        {
            m = tx;
        }
    }

    return m;
}

/* Version linéaire en calculant rapidement les temps
 * de convergence le long des valeurs prises par les itérés de f
 */
int temps_de_convergence_max(int *t)
{
    int m = 0; // max atteint pour l'attracteur
    int temps[n]; // va contenir soit le temps précalculé, soit -1

    for(int x = 0; x < n; x++)
    {
        if (t[x] == x)
        {
            temps[x] = 0; // c'est un point fixe
        }
        else
        {
            temps[x] = -1;
        }
    }

    for(int x = 0; x < n; x++)
    {
        int iteres[n]; // les itérés en partant de x
        int k = 0;
        int u = x;
        iteres[k] = u;
        while(temps[u] < 0)
        {
            u = t[u];
            k = k + 1;
            iteres[k] = u;
        }
        // on est tombe sur un point fixe
        // donc sur l'attracteur principal
        for(int r = k-1; r >= 0; r--)
        {
            temps[iteres[r]] = 1 + temps[iteres[r+1]];
        }

        if(temps[iteres[0]] > m) // c'est le plus grand temps de tous
        {
            m = temps[iteres[0]];
        }
    }
```

```
    return m;
}
```

2.2 Recherche efficace de points fixes

Premier cas

Q22

```
bool est_croissante(int *t)
{
    for(int i = 0; i < n-1; i++)
    {
        if(t[i] > t[i+1])
        {
            return false;
        }
    }

    return true;
}
```

Q23

```
// simple recherche dichotomique
int point_fixe(int *t)
{
    int i = 0;
    int j = n-1;
    while (j - i > 1)
    {
        int m = (i+j)/2;
        if (t[m] == m)
        {
            return m;
        }
        else if(t[m] < m)
        {
            j = m;
        }
        else
        {
            i = m;
        }
    }

    return -1; // unreachable
}
```

Q24 Voir cours, $j - i$ est un variant de boucle.

Deuxième cas

Q25 Comme f est croissante, alors f^k est croissante, donc $m \preceq x \to f^k(m) \preceq f^k(x) = x$.

Q26 Si p est le pgcd des x_i et que y est le plus petit point fixe, on a $p|y$ car y est l'un des x_i. On va montrer que $y|p$.
Pour cela, on remarque que pour $i \in [\![1, m]\!]$, $p|x_i$ et par croissance de f, $f(p)|f(x_i) = x_i$. Donc $f(p)$ divise leur pgcd : $f(p)|p$. La suite $(f(p)^k)$ est décroissante et minorée (par 1), donc elle converge par le théorème de la limite monotone : elle converge mais comme E_n est fini, elle est nécessairement stationnaire donc il existe $f(p)^k$ point fixe de f. Comme y est le plus petit point fixe, on a $y|f(p)^k$ et $y|p$ par transitivité.

Q27

```
int pgcd_points_fixes(int *t)
{
    // il suffit de trouver le point fixe accessible
    // depuis 1
    int x = 1;

    while(t[x] != x)
    {
        x = t[x];
    }

    return x;
}
```

Q28

Quand on boucle, comme $f^k(1)|f^{k+1}(1)$ et que ces deux nombres sont différents : soit $f^{k+1}(1) \geqslant 2f^k(1)$ (diviseur propre) soit $f^{k+1}(1) = 0$ (cas trivial qu'on peut ignorer pour le moment).
Comme on double $f^k(1)$ à chaque étape et que les valeurs sont $\leqslant n$, on en effectue au plus $\log n$.
Dans le cas où on obtient 0, on s'arrête à l'itération suivante.