

Épreuve d'informatique de l'X - 2012 - PT/PSI

Partie I. Autour des ensembles

Question 1. Il s'agit de compter le nombre de **vrai** présents dans le tableau. On parcourt le tableau dans la boucle des lignes 4 à 6 et on incrémente le compteur **cmpt** si i appartient à l'ensemble représenté par le tableau t (ligne 5).

```
1 cardinal := proc(t)
2   local cmpt, i;
3   cmpt := 0;
4   for i from 0 to taille(t) - 1 do
5     if t[i] then cmpt := cmpt + 1 fi;
6   od;
7   return(cmpt)
8 end;
```

Question 2. Si i est plus grand que la taille du tableau t alors i n'appartient pas à l'ensemble représenté par le tableau t (ligne 4). Sinon, il suffit de retourner la valeur $t[i]$ (ligne 3).

```
1 appartient := proc(i, t)
2   if i < taille(t)
3     then return(t[i])
4     else return(false)
5   fi;
6 end;
```

Question 3 : On commence par créer un tableau de la même longueur que le tableau t_1 (ligne 4). Puis, par la boucle des lignes 5-7, on « remplit » le tableau résultat **res**. On met **true** dans **res**[i] si i appartient à S_1 mais n'appartient pas à S_2 . Dans le cas contraire, on met **false**.⁽¹⁾

1. remarque : **diff** et **union** sont des mots protégés de Maple et ne peuvent pas être utilisés comme noms de fonction. Ce corrigé suit les instructions données par l'énoncé

(PSI*)

```
1 diff := proc(t1, t2)
2   local n1, i, res;
3   n1 := taille(t1);
4   res := allouer(n1, false);
5   for i from 0 to n1-1 do
6     res[i] := t1[i] and not(appartient(i, t2))
7   od;
8   return(res)
9 end;
```

Question 4 : On commence récupérer dans la variable n la plus grande des deux longueurs de tableau (lignes 3 à 6). Ensuite, on crée un tableau de la même longueur que le plus grand des tableaux (ligne 7). Dans la boucle des lignes 8 à 10, on met **true** dans **res**[i] si i appartient à S_1 ou à S_2 . Dans le cas contraire, on met **false**. On retourne le résultat à la ligne 11.⁽¹⁾

```
1 union := proc(t1, t2)
2   local n, i, res;
3   if taille(t1) > taille(t2)
4     then n := taille(t1)
5     else n := taille(t2)
6   fi;
7   res := allouer(n, false);
8   for i from 0 to n-1 do
9     res[i] := appartient(i, t1) or appartient(i, t2)
10  od;
11  return(res);
12 end;
```

Partie II. Représentation par entiers

Question 5 : L'entier $2^n - 1$ est le plus grand entier représentant un sous-ensemble de $\{0, 1, \dots, n-1\}$, cet entier représentant le sous-ensemble $\{0, 1, \dots, n-1\}$. L'entier 0 est le plus petit et représente l'ensemble vide.

Question 6 : On parcourt le tableau en partant de la fin. On initialise le résultat à 0 (ligne 3). À chaque passage dans la boucle des lignes 5 à 10, on multiplie le résultat par 2 et on ajoute 1 si l'élément $n-1-i$ appartient à l'ensemble S représenté par le tableau t . Au final, on obtient l'entier représentant l'ensemble associé au tableau t en un temps linéaire (un seul parcours de boucle).

```

1  set2int := proc(t)
2      local s, n, i;
3      s := 0;
4      n := taille(t);
5      for i from 0 to n-1 do
6          if t[n-1-i]
7              then s := 1 + 2*s
8              else s := 2*s
9          fi;
10     od;
11     return(s);
12 end;

```

Question 7 : On commence par déterminer la taille du tableau à créer en fonction de n (boucle des lignes 5-8). Si n est compris entre 2^p et $2^{p+1} - 1$, le tableau créé aura pour longueur p (sauf dans le cas particulier $n = 0$). Ensuite, on crée le tableau résultat t (ligne 9) que l'on remplit correctement grâce à la boucle des lignes 12 à 16. Le temps d'exécution est en $O(\log_2(n))$, les deux boucles conditionnelles étant parcourues au plus $p + 1$ fois si n appartient à l'intervalle $\llbracket 2^p, 2^{p+1} \rrbracket$.

```

1  int2set := proc(n)
2      local taille, m, res, i;
3      taille := 1;
4      m := quotient_div_2(n);
5      while m <> 0 do
6          taille := taille + 1;
7          m := quotient_div_2(m);
8      od;
9      res := allouer(taille, false);
10     m := n;
11     for i from 0 to taille-1 do
12         if parite(m) <> 0 then res[i] := true;
13         fi;
14         m := quotient_div_2(m);
15     od;
16     return(res);
17 end;

```

Partie III. Familles, sous-familles, et couvertures

Question 8 : On considère U l'ensemble des étudiants et pour une activité donnée a , l'ensemble F_a des élèves voulant pratiquer cette activité. On considère l'ensemble F de

tous les F_a pour toutes les activités a . L'ensemble F constitue bien un recouvrement car chaque élève a choisi au moins une activité. Le problème de l'école est de donner au moins une activité à chaque élève tout en minimisant le nombre d'activités, c'est bien un problème de couverture optimale.

Question 9 : On considère $n = 6$ et $F = (\{0, 2, 4\}, \{0, 1\}, \{2, 3\}, \{4, 5\})$. L'algorithme glouton conduit à choisir F comme sous-famille couvrante alors que la sous-famille couvrante optimale est $(\{0, 1\}, \{2, 3\}, \{4, 5\})$.

Question 10 : On utilise les fonctions **diff** et **cardinal** agissant sur des entiers. Le résultat de la fonction **reste** n'est autre que le cardinal de l'ensemble $S_1 \setminus S_2$.

```

1  reste := proc(s1, s2)
2      return(cardinal(diff(s1, s2)))
3  end;

```

Question 11 : À la ligne 6, on calcule $2^n - 1$ qui est le nombre représentant l'ensemble $\{0, \dots, n - 1\}$. Les lignes 9 à 12 servent à construire un tableau contenant les puissances successives de 2. Dans la boucle conditionnelle des lignes 13 à 24, on applique l'algorithme glouton. À chaque passage dans la boucle, on détermine le sous-ensemble élément de F qui contient le plus d'éléments de U . Ce sous-ensemble étant trouvé, on ajoute au résultat son indice dans le tableau f (ligne 23) et on supprime ses éléments présents dans U (ligne 22).

```

1  glouton := proc()
2      global n, f;
3      local u, p, puiss2, i, c, indice, res;
4      p := taille(f);
5      # calcul de u=2^n-1 représentant l'ensemble U={0,1,...,n-1}
6      u := set2int(allouer(n, true));
7      res := 0;
8      # on construit le tableau des puissances de 2.
9      puiss2 := allouer(p, 1);
10     for i from 1 to p-1 do
11         puiss2[i] := 2*puiss2[i-1];
12     od;
13     while u <> 0 do
14         c := cardinal(u);
15         # On cherche l'élément de F contenant le plus d'éléments de u
16         for i from 0 to p-1 do

```

```

17         if reste(u,f[i]) < c then
18             c := reste(u,f[i]);
19             indice := i
20         fi ;
21     od ;
22     u := diff(u,f[indice]);
23     res := res + puiss2[indice];
24 od ;
25 return(res);
26 end ;

```

Le coût de « **taille(f)** » (ligne 4) est en $O(p)$ où p est la taille de f , celui du calcul de u (ligne 6) en $O(n)$. Le coût de la première boucle (lignes 10-12) est en $O(p)$. Dans le pire des cas, dans la boucle conditionnelle (lignes 13-24), on supprime un seul élément dans U à chaque passage et donc cette boucle sera parcourue au plus n fois. La boucle interne des lignes 16 à 21 est parcourue p fois et la fonction « **reste** » ayant un coût en $O(n)$, on obtient un coût global en $O(n^2.p)$.

Question 12 : Dans la boucle des lignes 6 à 10, on calcule l'entier représentant l'ensemble $\bigcup_{\ell \in \text{int2set}(g)} F_\ell$. En supposant que les F_ℓ sont bien tous inclus dans l'ensemble $\{0, \dots, n-1\}$, il suffit de regarder si le cardinal de l'ensemble calculé est bien égal à n pour avoir un recouvrement.

```

1  couverture := proc(g)
2      global f, n;
3      local res, l, G;
4      res := 0;
5      G := int2set(g);
6      for l from 0 to taille(G)-1 do
7          if G[l] then
8              res := union(res, f[l])
9          fi ;
10     od ;
11     return(cardinal(res) = n);
12 end ;

```

Question 13 : Pour trouver la solution optimale, on étudie toutes les parties de F (boucle des lignes 13 à 19) et parmi celles qui recouvrent l'ensemble $\{0, 1, \dots, n-1\}$, on retourne une solution de cardinal minimal. Dans le test des lignes 15 à 18, si g est associé à un recouvrement de $\{0, 1, \dots, n-1\}$ et que son cardinal est strictement inférieur à la meilleure des solutions trouvées auparavant, alors on actualise la solution provisoire « **res** » et le cardinal du meilleur recouvrement provisoire « **CardinalOptimal** ».

```

1  optimal := proc()
2      global n, f;
3      local p, cardPf, i, CardinalOptimal, cardinalG, res, g;
4      p := taille(f);
5      cardPf := 1;
6      # On calcule le nombre de parties d'un ensemble à p éléments,
7      # soit 2^p.
8      for i from 0 to p-1 do
9          cardPf := 2 * cardPf;
10     od ;
11     CardinalOptimal := cardPf;
12     res := cardPf - 1;
13     for g from 0 to cardPf-1 do
14         cardinalG := cardinal(g);
15         if couverture(g) and cardinalG < CardinalOptimal then
16             CardinalOptimal := cardinalG;
17             res := g;
18         fi ;
19     od ;
20     return(res);
21 end ;

```

Le coût du calcul de 2^p est p . La boucle des lignes 13 à 19 est parcourue 2^p fois. Le coût de la fonction **cardinal** est en $O(n)$ (d'après la solution proposée à la question 1), le coût de la fonction **couverture** étant en $O(np)$ (On calcule la réunion d'au plus p ensembles de cardinal au plus n), on obtient un coût global en $O(np2^p)$.