X2012 — INF421
Examen de rattrapage / 25 novembre 2013

Tous documents autorisés (poly, notes de cours, notes de PC). Dictionnaires électroniques autorisés pour les élèves étrangers.

L'énoncé est composé de 2 parties indépendantes, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes de la même partie.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

---

1 Le compte est bon

Dans ce problème, on s'intéresse à la question suivante : quel est l'ensemble $T(S)$ des nombres entiers que l'on peut construire, en utilisant uniquement l'addition, la soustraction et la multiplication, à partir des éléments d'un ensemble fini S d'entiers, en utilisant chaque élément de S une fois et une seule ? Ainsi, à partir de l'ensemble $S = \{2, 3, 7\}$ on peut construire l'entier -19 (comme $2 - 3 \times 7$), l'entier 6 (comme $7 - (3 - 2)$) ou encore l'entier 27 (comme $3 \times (2 + 7)$). Au total, on peut construire 24 nombres différents à partir de S, à savoir

$$T(S) = \{-19, -15, -11, -8, -7, -6, -2, -1, 1, 2, 6, 7, 8, 11, 12, 13, 15, 17, 19, 20, 23, 27, 35, 42\}.$$

Dans la suite, on note n le cardinal de S et on suppose $n < 32$. On suppose qu'on se donne S sous la forme d'un tableau contenu dans un champ `S` de la classe dans laquelle on travaille :

```
class Numbers {
  private final int[] S;
  Numbers(int[] S) { this.S = S; }
  ...
}
```

Pour résoudre le problème, on va le généraliser et chercher $T(U)$ pour tout sous-ensemble U de S. Un sous-ensemble U de S sera représenté par un entier `u`. Le bit i de `u`, pour $0 \leq i < n$, indique la présence de l'élément `S[`i`]` dans U. (C'est la même idée que celle utilisée dans le poly et l'amphi 3 pour résoudre le problème des N reines.) Ainsi pour $n = 6$ et le tableau `S={2,3,7,10,25,100}`, l'entier 42, dont l'écriture en binaire est 101010_2, représente le sous-ensemble $\{3, 10, 100\}$, car les bits à 1 sont les bits de positions 1, 3 et 5 (correspondant donc à `S[1]`, `S[3]` et `S[5]`). On commence par quelques méthodes élémentaires pour manipuler cette représentation.

Question 1 Écrire une méthode `boolean isSingleton(int u, int i)` qui renvoie `true` si et seulement si le sous-ensemble de S représenté par l'entier `u` est le singleton {`S[i]`}. On pourra supposer $0 \leq$ `i` $< n$.

Correction :

```
return u == (1 << i);
```

Question 2 Écrire une méthode `boolean isIncluded(int u1, int u2)` qui renvoie `true` si et seulement si le sous-ensemble de S représenté par l'entier `u1` est inclus dans le sous-ensemble de S représenté par `u2`.

Correction :

```
return (u1 & u2) == u1;
```

Question 3 Écrire une méthode `int setDifference(int u1, int u2)` qui renvoie l'entier représentant la différence ensembliste `u1\u2`, pour deux sous-ensembles de S représentés par les entiers `u1` et `u2`.

Correction :

```
return u1 & ~u2;
```

Une idée pour résoudre le problème donné consiste à écrire une méthode récursive

```
TreeSet<Integer> computeAll(int u)
```

qui prend en argument un entier `u` représentant un sous-ensemble U de S et renvoie l'ensemble des entiers que l'on peut construire à partir de U. Pour représenter l'ensemble renvoyé, on utilise la classe `TreeSet` de la bibliothèque Java (un arbre binaire de recherche équilibré). Pour ce qui suit, il suffit de savoir que `new TreeSet<Integer>()` construit un nouvel ensemble (vide), que `s.add(x)` ajoute l'entier `x` à l'ensemble `s` et que la notation `for(int x: s)` peut être utilisée pour parcourir tous les éléments d'un ensemble `s`. On propose la structure suivante pour la méthode `computeAll` :

```
TreeSet<Integer> computeAll(int u) {
  assert (u != 0);
  TreeSet<Integer> res = new TreeSet<Integer>();
  // cas 1 : u est un singleton
  for (int i = 0; i < this.S.length; i++)
    if (isSingleton(u, i)) {
      ...
      return res;
    }
  // cas 2 : u est l'union disjointe de deux sous-ensembles non vides
  for (int left = 1; left < u; left++)
    if (isIncluded(left, u)) {
      ...
    }
  return res;
}
```

Comme l'indique la première ligne du code, on suppose que le sous-ensemble U représenté par l'entier `u` est non vide. On distingue alors deux cas : le cas où U est un singleton et le cas où il contient au moins deux éléments. Dans ce second cas, on examine toutes les décompositions $U = L \uplus R$ possibles, avec L et R non vides, le sous-ensemble L étant représenté par l'entier `left` dans le code ci-dessus.

Question 4 Écrire le code correspondant au premier cas (c'est-à-dire à la première occurrence de ... dans le code ci-dessus).

Correction :

```
res.add(this.S[i]);
```

Question 5 Expliquer pourquoi les valeurs de `left` considérées sont comprises entre 1 (inclus) et `u` (exclus).

Correction : `left` ne peut être plus grand que `u` car sinon `left` contiendrait des éléments qui ne sont pas dans `u`. Par ailleurs, 0 et `u` sont exclus car on cherche un sous-ensemble strict de `u`.

Question 6 Écrire le code correspondant au second cas (c'est-à-dire à la seconde occurrence de ... dans le code ci-dessus).

Correction :

```
int right = setDifference(u, left);
if (left < right) continue; // OPTIM
TreeSet<Integer> sleft = computeAll(left);
TreeSet<Integer> sright = computeAll(right);
for (int x: sleft) {
  for (int y: sright) {
    res.add(x + y);
    res.add(x - y);
    res.add(y - x);
    res.add(x * y);
  }
}
```

Question 7 Écrire enfin une méthode `TreeSet<Integer> computeAll()` qui répond au problème initial, c'est-à-dire qui renvoie l'ensemble des valeurs que l'on peut construire à partir de S.

Correction :

```
return allMemo((1 << S.length) - 1);
```

Question 8 Expliquer comment modifier le programme ci-dessus pour que chaque élément de S soit utilisé *au plus une fois* (et non pas *exactement* une fois). Note : un nombre de S au moins doit être utilisé.

Correction : Il suffit d'ajouter tous les éléments de `computeAll(left)` à `res`, c'est-à-dire ajouter la ligne

```
res.add(x);
```

juste après la ligne `for(int x: sleft)`.

Question 9 Peut-on utiliser la technique de mémoïsation pour améliorer l'efficacité en temps de ce programme ? Si oui, expliquez comment.

Correction : On va être amené à calculer plusieurs fois `computeAll(u)` pour la même valeur de `u`. Par exemple, dans le cas $n = 3$, on aura trois décompositions de $S = \{a, b, c\}$, à savoir $\{a\} \cup \{b, c\}$, $\{b\} \cup \{a, c\}$ et $\{c\} \cup \{a, b\}$, et on appellera donc plusieurs fois `computeAll` sur les singletons $\{a\}$, $\{b\}$, et $\{c\}$.
Pour mémoïser `computeAll`, il suffit d'introduire une table de hachage globale

```
private HashMap<Integer, TreeSet<Integer>> memo =
  new HashMap<Integer, TreeSet<Integer>>();
```

de la consulter au début de `computeAll` et de la remplir à la fin.

Question 10 Comment faire pour conserver, pour chaque entier qui peut être construit avec S, une séquence d'opérations permettant de le calculer ? (S'il en existe plusieurs, on choisit arbitrairement.) On ne demande pas d'écrire le code Java, mais seulement d'indiquer une méthode.

Correction : Plutôt qu'un ensemble d'entiers, on construit un ensemble de

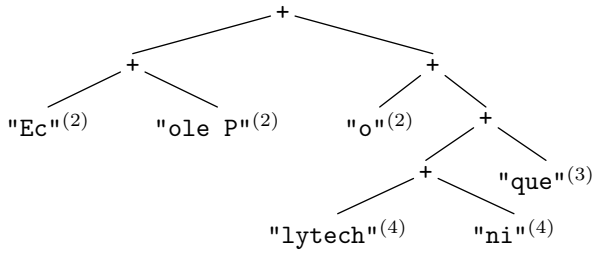
```
class Sol implements Comparable<Sol> {
  final int value;
  final char op;
  final Sol left, right;
  ...
}
```

c'est-à-dire une valeur entière (champ `value`) et la façon dont elle a été construite (champs `op`, `left` et `right`).

---

2 Des cordes équilibrées

On rappelle qu'une corde est une structure de données pour représenter des chaînes de caractères selon le principe suivant : une corde est un arbre binaire, où chaque nœud interne représente une concaténation et chaque feuille contient une chaîne de caractères usuelle. La chaîne représentée par une corde est donc la concaténation de toutes les feuilles, considérées dans l'ordre infixe. Ainsi la corde



représente la chaîne `"Ecole Polytechnique"`. La profondeur de chaque feuille est sa distance à la racine ; elle est indiquée dans l'exemple ci-dessus entres parenthèses. D'une manière générale, une corde est formée de k feuilles $s_1, \ldots, s_k$, aux profondeurs respectives $p_1, \ldots, p_k$. On définit alors sa longueur, notée $L(c)$, comme son nombre total de caractères, c'est-à-dire

$$L(c) = \sum_{i=1}^{k} |s_i|$$

où $|s_i|$ désigne la longueur de la chaîne s_i. On définit la hauteur de la corde c, notée $H(c)$, comme la profondeur maximale de ses feuilles, c'est-à-dire

$$H(c) = \max_{1 \leq i \leq k} p_i.$$

Enfin, on définit le *coût* d'une corde, noté $C(c)$, comme la somme de la distance de tous ses *caractères* à la racine, c'est-à-dire

$$C(c) = \sum_{i=1}^{k} p_i \times |s_i|.$$

Le but de ce problème est de réaliser un algorithme d'*équilibrage* d'une corde, pour majorer le coût en fonction de la longueur de la corde.

On rappelle qu'une corde peut être représentée en Java à l'aide d'une classe abstraite `Rope`, avec deux sous-classes `Str` et `App` représentant respectivement une feuille et un nœud interne. La structure est la suivante :

```
abstract class Rope {
  int length;
  ...
}
class Str extends Rope {
  private final String str;
  Str(String str) { this.str = str; this.length = str.length(); }
  ...
}
class App extends Rope {
  private final Rope left, right;
  App(Rope left, Rope right) {
    this.left = left; this.right = right;
    this.length = left.length + right.length;
  }
  ...
}
```

On note que le champ `length` de chaque corde c contient sa longueur $L(c)$. La corde vide, notée `empty`, est la corde `new Str("")` (supposée construite une fois pour toutes par la suite).

Question 11 Écrire une méthode `Rope append(Rope r)` qui concatène la corde `this` et la corde `r`, en prenant soin de ne pas construire de nouvelle corde lorsque $L(\texttt{this}) = 0$ ou $L(\texttt{r}) = 0$.

Correction :

```
Rope append(Rope r) {
  if (this.length == 0) return r;
  if (r.length    == 0) return this;
  return new App(this, r);
}
```

Question 12 Écrire une méthode `int cost()` dans la classe `Rope`, qui calcule le coût d'une corde, c'est-à-dire $C(\texttt{this})$. Si besoin, on introduira une méthode auxiliaire, définie différemment dans chacune de deux sous-classes `Str` et `App`.

Correction : dans la classe `Rope` :

```
int cost() { return cost(0); }
abstract int cost(int depth);
```

dans la classe `Str` :

```
int cost(int depth) { return depth * length; }
```

et enfin dans la classe `App` :

```
int cost(int depth) {
  return this.left.cost(depth + 1) + this.right.cost(depth + 1);
}
```

Autre solution : dans `Str`, `cost` renvoie 0 et dans `App`, `cost` renvoie `length + left.cost() + right.cost()`.

On propose l'algorithme suivant pour équilibrer une corde c. On considère un tableau `queue` de cordes dans lequel les feuilles $s_1, \ldots, s_k$ de c vont être insérées successivement, dans le sens des indices croissants. Les cases d'indices 0 et 1 ne sont pas utilisées. La case d'indice i contient soit la corde vide `empty`, soit une corde de hauteur $\leq i-2$ et dont la longueur est comprise dans l'intervalle $[F_i, F_{i+1}[$ où F_i désigne le i-ième terme de la suite de Fibonacci.

1. On insère successivement chaque feuille s_j dans `queue`, à partir de la case d'indice $i = 2$. L'insertion d'une feuille, et plus généralement d'une corde, à partir de la case d'indice i se fait ainsi :
   (a) On concatène, avec `append`, la corde à insérer avec la corde se trouvant dans la case i ; soit c_i le résultat. Si la longueur de c_i est comprise dans l'intervalle $[F_i, F_{i+1}[$ alors on affecte c_i à la case i et on a terminé l'insertion.
   (b) Sinon, on affecte `empty` à la case i et on retourne à l'étape (a) pour effectuer l'insertion de c_i à partir de la case d'indice $i+1$.

   On notera qu'on a l'invariant suivant : après l'insertion de la feuille s_j, la concaténation de toutes les cordes de `queue`, considérées dans le sens des indices décroissants, est égale au mot $s_1 s_2 \ldots s_j$.

2. Le résultat est alors la concaténation de toutes les cordes de `queue`, à savoir

$$\texttt{queue}[m-1].\texttt{append}(\texttt{queue}[m-2].\texttt{append}(\ldots(\texttt{queue}[3].\texttt{append}(\texttt{queue}[2]))))$$

où m désigne la taille de `queue`.

On rappelle que la suite de Fibonacci (F_n) est définie par

$$\begin{cases} F_0 &= 0, \\ F_1 &= 1, \\ F_{n+2} &= F_{n+1} + F_n \quad \text{pour } n \geq 0. \end{cases}$$

On suppose la longueur des cordes toujours inférieure à F_{44}. On limite donc la taille de `queue` à 44 cases indexées de 0 à 43 (les cases 0 et 1 n'étant pas utilisées). On introduit la constante `maxFib` et le tableau `fib` suivants

```
static final int maxFib = 44;
static final int[] fib = new int[maxFib + 1];
```

et on suppose que le tableau `fib` est rempli avec les valeurs F_i pour $0 \leq i \leq$ `maxFib`. Pour réaliser le point 1 de l'algorithme ci-dessus, on se donne les deux méthodes

```
void insertQueue(Rope[] queue, Rope r, int i) { ... }
abstract void insertLeaves(Rope[] queue);
```

dans la classe `Rope`. La méthode `insertLeaves` insère toutes les feuilles de la corde `this` dans le tableau `queue`. La méthode `insertQueue` insère la corde `r` dans `queue`, à partir de l'indice `i`.

Question 13 Que signifie le mot clé `abstract` devant la méthode `insertLeaves` ?

Correction : que la méthode `insertLeaves` n'est pas implémentée dans la classe `Rope`, mais seulement dans ses sous-classes.

Question 14 Écrire le code de la méthode `insertLeaves` dans chacune des classes `Str` et `App`.

Correction :

```
class Str { ...
  void insertLeaves(Rope[] queue) { insertQueue(queue, this, 2); }
}
class App { ...
  void insertLeaves(Rope[] queue) {
    this.left.insertLeaves(queue);
    this.right.insertLeaves(queue);
  }

}
```

Question 15 Écrire le code de la méthode `insertQueue`. On supposera $0 \leq \texttt{i} < \texttt{maxFib}$, $H(\texttt{r}) \leq \texttt{i} - 2$ et $L(\texttt{r}) \geq F_{\texttt{i}}$.

Correction :

```
void insertQueue(Rope[] queue, Rope r, int i) {
  assert (i < maxFib);
  Rope c = queue[i].append(r);
  if (c.length < fib[i + 1])
    queue[i] = c;
  else {
    queue[i] = empty;
    insertQueue(queue, c, i + 1);
  }
}
```

Question 16 Montrer que l'invariant $H(\texttt{queue}[i]) \leq i - 2$ et $L(\texttt{queue}[i]) \geq F_i$ est préservé par la fonction `insertQueue`, sous les hypothèses de la question précédente, pour toutes les cases $\texttt{queue}[i]$ du tableau `queue` non égales à `empty`, avec $2 \leq i < \texttt{maxFib}$.

Correction :

```
l'invariant de insertQueue est :

    H(c) <= i-2 et L(c) >= fib(i)

- cas n' < fib.(i+1) :
  - si queue(i) = empty, alors c' = c  et l'invariant de la file est vérifié
  - sinon, n' = L(c) + L(queue(i))
            >= F_i + F_i
            >= F_{i+1} contradiction

- cas n' >= fib.(i+1):
  on a H(c') = 1 + max(H(c),queue(i)) <= i-1 = (i+1)-2
  et n' = L(c') >= fib_{i+1}
  donc l'invariant de insertQueue est bien préservé
```

Question 17 Compléter la méthode `balance` qui réalise l'équilibrage d'une corde :

```
Rope balance() {
  Rope[] queue = new Rope[maxFib];
  for (int i = 0; i < maxFib; i++)
    queue[i] = empty;
  this.insertLeaves(queue);
  ...
}
```

Il s'agit donc de remplacer ... par du code réalisant le point 2 de l'algorithme donné plus haut.

Correction :

```
Rope r = empty;
for (int i = 2; i < maxFib; i++)
  r = queue[i].append(r);
return r;
```

Question 18 Soit c une corde non vide renvoyée par la méthode `balance` ci-dessus. Soit n sa longueur et h sa hauteur. Montrer que l'on a

$$n \geq F_{h+1}.$$

En déduire qu'il existe une constante K (indépendante de n) telle que

$$C(c) \leq n\,(\log_\phi(n) + K),$$

où $\log_\phi$ désigne le logarithme à base ϕ, ϕ étant le nombre d'or $(1+\sqrt{5})/2$. On admettra que $F_{i+1} \geq \phi^i/\sqrt{5}$ pour tout $i \geq 0$.

Correction :

```
n >= F_{h+1} :

dans la boucle de balance, l'invariant est le suivant :
si r <> empty alors H(r) <= i-2 et L(r) >= F_{H(c)+1}

initialement : r = empty et l'invariant est donc trivialement vrai

préservation : soit r' = queue(i).append(r) ; supposons r' <> empty

- H(r') <= 1 + max(H(queue(i)), H(r))
        <= 1 + max(i-2, i-2)
         = (i+1)-2

- si r = empty, alors r' = queue(i) et
    L(r') >= F_i >= F_{i-1} >= F_{H(r')+1}
    car H(r') = H(queue(i)) <= i-2

  si r <> empty, alors
    L(r') = L(queue(i)) + L(r)
          >= F_i + F_{H(r)+1}              (1)

    -- si H(r) >= H(queue(i))
       alors H(r') = 1 + H(r)
       comme i >= H(r)+2 alors (1) donne
          L(r') >= F_{H(r)+2} + F_{H(r)+1}
                >= F_{H(r)+1+1}
                 = F_{H(r')+1}
```

```
      -- si H(r) < H(queue(i))
         alors H(r') = 1 + H(queue(i))
         comme i >= H(queue(i))+2 alors (1) donne
            L(r') >= F_{H(queue(i))+2} + F_{H(r)+1}
                  >= F_{H(queue(i))+1+1}
                   = F_{H(r')+1}

  on déduit immédiatement le résultat n >= F_{h+1} de cet invariant
  car on a supposé r non vide à la sortie

* C(c) <= ... :

  on a C(c) <= n * h

  or n >= F_{h+1} >= 1/sqrt{5} * phi^h

  donc h <= log_phi(n) + K
```

---

* *
*