INF3105 – Structures de données et algorithmes
Examen final – Automne 2012

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 20 décembre 2012 – 13h30 à 16h30 (3 heures) – Locaux SH-2120 + SH-2580

Instructions

- Aucune documentation permise.
- Les appareils électroniques, incluant les calculatrices, sont strictement interdits.
- Répondez directement sur le questionnaire à l'intérieur des endroits appropriés.
- Pour les questions demandant l'écriture de code :
  - le fonctionnement correct, la robustesse, la clarté, l'efficacité (temps et mémoire) et la simplicité du code sont des caractéristiques à considérer ;
  - vous pouvez scinder votre solution en plusieurs fonctions à condition de donner le code pour chacune d'elles ;
  - vous pouvez supposer l'existence de fonctions raisonnables ;
  - le respect exact de la syntaxe de C++ n'est pas sujet à la correction.
- Aucune question ne sera répondue durant l'examen. Si vous croyez qu'une erreur ou qu'une ambiguïté s'est glissée dans le questionnaire, indiquez clairement la supposition que vous avez retenue pour répondre à la question.
- L'examen dure 3 heures et contient 4 questions.
- Ne détachez pas les feuilles du questionnaire, à moins de les rebrocher avant la remise.
- Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées au surveillant.

Identification

Nom : Solutionnaire

Résultat

Q1 (/6)	Q2 (/7)	Q3 (/8)	Q4 (/4)	TOTAL (/25)

1 Monceaux (*Heap*) [6 points]

```
1  template <class T> T Monceau<T>::enleverMinimum(){
2      valeurs.enlever_dernier();
3      descendre(0);
4      return valeurs[0];
5  }
6  template <class T> void Monceau<T>::descendre(int indice){
7      int suivant = indice*2;
8      if(valeurs[suivant+1]<valeurs[suivant]) suivant++;
9      if(valeurs[suivant]<valeurs[indice]){
10         valeurs[suivant] = valeurs[indice];
11         valeurs[indice] = valeurs[suivant];
12         descendre(suivant);
13     }
14 }
```

(a) Le code précédent contient des erreurs. Proposez une correction dans l'endacré ci-bas. À noter qu'il n'y a pas d'erreur au niveau du langage C++. Suggestion : commencez par noter les erreurs dans le code ci-haut. Une fois que vous êtes sûr de votre coup, transcrivez le tout au propre ci-bas. Conseil : simuler le code ligne par ligne sur un exemple devrait vous aider à identifier les erreurs. [3 points]
Il y a 7 erreurs :

```
1  template <class T> T Monceau<T>::enleverMinimum(){
2      T minimum = valeurs[0]; // Erreur 1 : il faut garder le minimum
3      valeurs[0] = valeurs[valeurs.taille()-1]; // Erreur 2 : dernier devient premier
4      valeurs.enlever_dernier();
5      descendre(0);
6      return minimum; // Erreur 3 : retourner min
7  }
8  template <class T> void Monceau<T>::descendre(int indice){
9      int suivant = indice*2+1; // Erreur 4 : il manque le +1
10     if(suivant>=valeurs.taille()) return; // Erreur 5 : cas d'arret recursion
11     //Erreur 6 : il faut tester si on peut regarder l'enfant 2
12     if(suivant+1<valeurs.taille() && valeurs[suivant+1]<valeurs[suivant]) suivant++;
13     if(valeurs[suivant]<valeurs[indice]){
14         T temp = valeurs[suivant]; // Erreur 7 : valeur temporaire pour echange
15         valeurs[suivant] = valeurs[indice];
16         valeurs[indice] = temp;
17         descendre(suivant);
18     }
19 }
```

(b) Soit le monceau `m` suivant. Insérez les éléments 5 et 13 puis enlevez le minimum. Donnez l'état du monceau après toutes les étapes intermédiaires (échanges). À noter qu'il peut y avoir plus de lignes que nécessaire. [2 points]

Étape	m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]	m[7]	m[8]	m[9]	m[10]	m[11]
#0	3	6	4	7	9	10	8	21	17	11		
#1	3	6	4	7	9	10	8	21	17	11	**5**	
#2	3	6	4	7	**5**	10	8	21	17	11	**9**	
#3	3	**5**	4	7	**6**	10	8	21	17	11	9	
#4	3	5	4	7	6	10	8	21	17	11	9	**13**
#5	–	5	4	7	6	10	8	21	17	11	9	13
#6	**13**	5	4	7	6	10	8	21	17	11	9	–
#7	**4**	5	**13**	7	6	10	8	21	17	11	9	
#8	4	5	**8**	7	6	10	**13**	21	17	11	9	

(c) Complétez le tableau suivant en indiquant la complexité temporelle des opérations dans un monceau contenant n éléments. Utilisez la notation grand O. [1 point]

Opération	Cas moyen	Pire Cas
Insertion	$O(\log n)$	$O(\log n)$
Retourner minimum sans l'enlever	$O(1)$	$O(1)$
Enlever le minimum	$O(\log n)$	$O(\log n)$

2 Table de hachage (*Hashtable*) [7 points]

Imaginez que durant le temps des fêtes, vous tentiez d'améliorer votre TP1 – Simulation de serpents virtuels. Dans le TP1, parmi les opérations coûteuses, il y a la vérification de la présence d'un obstacle et la vérification de la présence de fruits dans une case de la grille. En utilisant une liste ou un tableau non ordonné, ces opérations ont une complexité temporelle linéaire, c'est-à dire de $O(o)$ et $O(f)$, où o et f sont respectivement le nombre d'obstacles et de fruits dans la grille.

(a) Croyez-vous qu'il est **possible** d'améliorer les temps $O(o)$ et $O(f)$ en utilisant des ensembles de hachage (ex. : `std::hashset`) et/ou tables de hachage (ex. : `std::hashmap`) ? Par «possible», on sous-entend le cas moyen et on exclut le pire cas. Si cela est possible, indiquez quelle(s) complexité(s) temporelle(s) peut-on espérer pour vérifier la présence d'un obstacle et de fruits dans une case de la grille. [1 point]

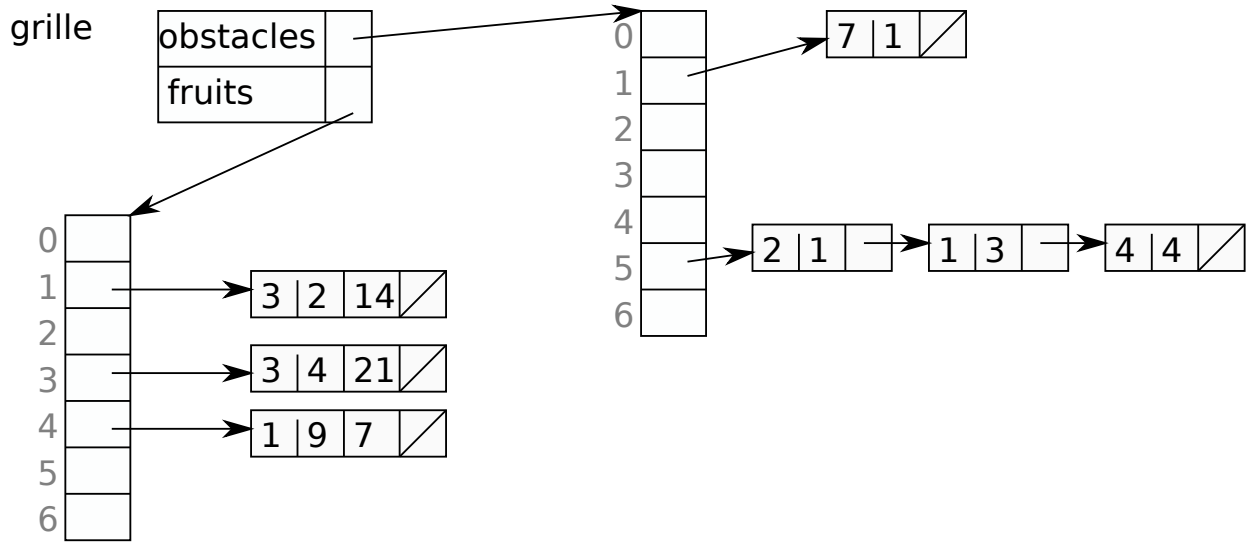
Oui, c'est possible.
$O(1)$.

Voici un extrait de la déclaration d'une classe `Grille` pour le TP1 et la définition d'une fonction `Position::valeurHachee` qui calcule et retourne une adresse dispersée.

```
1  class Grille{
2    public:
3      // retourne vrai si la case p contient un obstacle
4      bool contientObstacle(const Position& p) const;
5      // retourne et enleve le nombre d'unites d'energie a la case p
6      int  mangerFruits(const Position& p);
7      // ...
8    private:
9      EnsembleHache<Position> obstacles; // Enumeration des obstacles
10     TableHache<Position, int> fruits; // Map : position -> unites d'energie
11     // ...
12 };
13 /...
14 int Position::valeurHachee() const{
15     return 2*x+y;
16 }
```

(b) Dessinez la représentation en mémoire d'un objet `grille` de type `Grille` contenant quatre obstacles, situés aux positions $(2,1)$, $(1,3)$, $(4,4)$ et $(7,1)$, et trois fruits, situés aux positions $(3,2)$, $(3,4)$ et $(1,9)$, et ayant respectivement 14, 21 et 7 unités d'énergie. Supposez que les objets `obstacles` et `fruits` ont présentement 7 casiers (*buckets*). La gestion des collisions doit se faire avec une liste chaînée (gestion externe). [2 points]

Il y a plusieurs façons de dessiner cela. On peut faire abstraction du support du tableau.



(c) Imaginez que vous deveniez correctrice ou correcteur pour INF3105. Imaginez que l'UQAM met en place un nouveau système de rémunération, pour les correcteurs et correctrices, qui prévoit des primes inversement proportionnelles aux notes accordées. Dans ce nouveau système de rémunération, vous avez intérêt à trouver le maximum d'erreurs. Et lorsque l'efficacité des programmes est évaluée, vous avez intérêt à **tester le pire cas** !
Pour maximiser votre paie pour le TP1, vous décidez de tester un pire cas contenant 2500 obstacles. En supposant que la classe `EnsembleHache` a un facteur de chargement (*load factor*) de 0.25, vous pouvez déduire que l'objet `obstacles` contiendra 10000 casiers (*buckets*) [1]. En considérant la fonction `Position::valeurHachee()` à la page précédente, écrivez un programme qui génère et affiche (`cout <<`) les coordonnées d'un pire cas. [2 points]
Plusieurs réponses possibles. Des solutions plus simples existent. Les includes et la signature du main() ne sont pas nécessaires.

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4      // il n'est pas requis d'expliciter ces variables
5      int nbCasiers = 10000;
6      int n = 2500;
7      for(int i=0;i<n;i++)
8      {
9          int x = i;
10         int y = nbCasiers - (2*x)%nbCasiers;
11         cout << '(' << x << ',' << y << ')' << endl;
12     }
13 }
```

(d) Quelle est la complexité temporelle de la fonction `Grille::contientObstacle(const Point& p)` dans le pire cas en supposant o obstacles ? [1 point]

$O(o)$. Dans le pire cas, il faut itérer toute la liste attachée au casier (*bucket*).

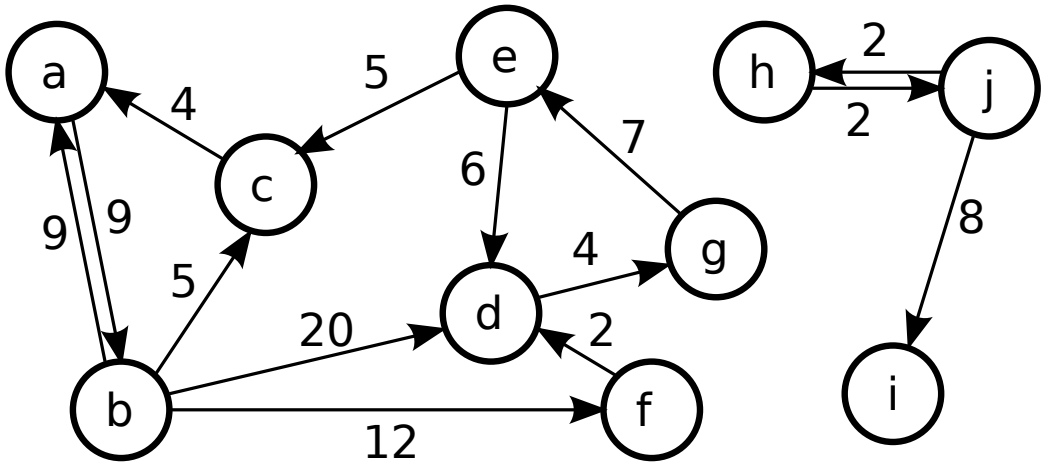
(e) Les étudiant(e)s ont été mis au courant que l'efficacité de leur TP1 sera évaluée à l'aide de deux tests de performance. Il y aura un test sera un test aléatoire et un test d'un pire cas. Expliquez comment un(e) étudiant(e) peut maximiser sa note sachant ces informations. [1 point]

Pour obtenir un bon résultat dans le test aléatoire, il faut garder les ensembles et tables de hachage. Ainsi, on peut espérer une complexité de $O(1)$ pour les fonctions de présence d'obstacle et de fruits. Pour éviter la dégénération dans un pire cas, il suffit d'utiliser un arbre binaire de recherche équilibré (AVL ou R-N) comme structure auxiliaire pour gérer les collisions. Ainsi, la complexité du pire cas passe d'un temps linéaire à un temps logarithmique !

1. Le nombre 10000 est choisi pour simplifier le problème. En pratique, on préfère des nombres premiers.

3 Graphe [8 points]

Soit le graphe orienté $G_1 = (V_1, E_1)$ suivant, où $V_1 = \{a,b,c,d,e,f,g,h,i,j\}$ et $E_1 = \{(a,b,9),(b,a,9),\cdots,(j,i,8)\}$.



(a) Donnez l'ordre de visite des sommets lors d'une recherche en **profondeur** dans G_1 à partir du sommet c. Supposez que la fonction `getAretesSourtantes()` retourne les arêtes dans l'ordre alphabétique des sommets d'arrivée. [1 point]

c,a,b,d,g,e,f

(b) Donnez l'ordre de visite des sommets lors d'une recherche en **largeur** dans G_1 à partir du sommet e. Supposez que la fonction `getAretesSourtantes()` retourne les arêtes dans l'ordre alphabétique des sommets d'arrivée. [1 point]

e,c,d,a,g,b,f

(c) Énumérez les compostantes **fortements connexes** de G_1 ? [1 point]

$\{\{a,b,c,d,e,f,g\},\{h,j\},\{i\}\}$

(d) Simulez l'algorithme Dijkstra pour trouver le plus court chemin de e à f. Indiquez clairement toutes les étapes. [2 points]

Notation : dans les colonnes de sommet, distance / parent.

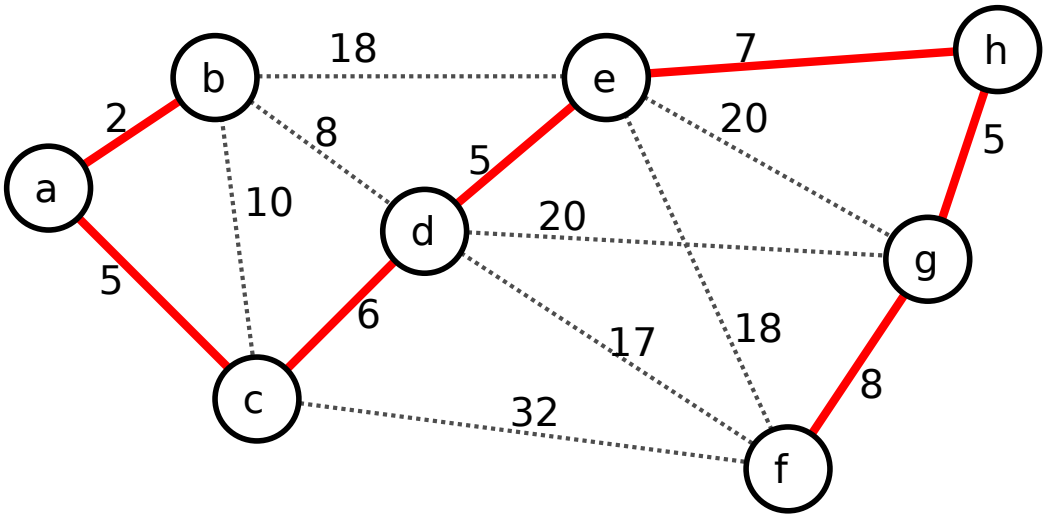
Étape	Sommet choisi	a	b	c	d	e	f	g	h	i	j	File prioritaire
#0	(initialisation)	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	e
#1	e	$+\infty$	$+\infty$	5 / e	6 /e	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	c, d
#2	c	9 /c	$+\infty$	5 / e	6 /e	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	d, a
#3	d	9 /c	$+\infty$	5 / e	6 /e	0	$+\infty$	10/d	$+\infty$	$+\infty$	$+\infty$	a, g, b
#4	a	9 /c	18 / a	5 / e	6 /e	0	$+\infty$	10/d	$+\infty$	$+\infty$	$+\infty$	g, b
#5	g	9 /c	18 / a	5 / e	6 /e	0	$+\infty$	10/d	$+\infty$	$+\infty$	$+\infty$	b
#6	b	9 /c	18 / a	5 / e	6 /e	0	30 / b	10/d	$+\infty$	$+\infty$	$+\infty$	f
#7	f	9 /c	18 / a	5 / e	6 /e	0	30 / b	10/d	$+\infty$	$+\infty$	$+\infty$	

Chemin : $\langle e,c,a,b,f\rangle$. Longueur : 30.

Il y a plusieurs façons présenter la solution et de simuler l'algorithme.

Initialement, on peut aussi mettre tous les sommets dans la file, ou juste mettre le sommet d'origine.

Soit le graphe non orienté $G_2 = (V_2, E_2)$ suivant, où $V_2 = \{a,b,c,d,e,f,g,h,i\}$ et $E_2 = \{(a,b,8),(a,c,5),\cdots,(g,h,5)\}$.



(e) Simulez l'algorithme Prim-Jarnik pour retrouver un arbre de recouvrement minimal pour le graphe G_2. [2 points]

Étape	Choix	File prioritaire
#1	Choix arbitraire sommet a	$(a,b,2),(a,c,5)$
#2	Choix de $(a,b,2)$ pour relier b	$(a,c,5),(b,d,8),(b,c,10),(b,e,18)$
#3	Choix de $(a,c,5)$ pour relier c	$(c,d,6),(b,d,8),(b,c,10),(b,e,18),(c,f,32)$
#4	Choix de $(c,d,6)$ pour relier d	$(d,e,5),(b,d,8),(b,c,10),(d,f,17),(b,e,18),(d,g,20),(c,f,32)$
	Optionnel : élimination d'arcs	$(d,e,5),(d,f,17),(b,e,18),(d,g,20),(c,f,32)$
#5	Choix de $(d,e,5)$ pour relier e	$(e,h,7),(d,f,17),(e,f,18),(b,e,18),(e,g,20),(d,g,20),(c,f,32)$
	Optionnel : élimination d'arcs	$(e,h,7),(d,f,17),(e,f,18),(e,g,20),(d,g,20),(c,f,32)$
#6	Choix de $(e,h,7)$ pour relier h	$(h,g,5),(d,f,17),(e,f,18),(e,g,20),(d,g,20),(c,f,32)$
#7	Choix de $(h,g,5)$ pour relier g	$(g,f,8),(d,f,17),(e,f,18),(e,g,20),(d,g,20),(c,f,32)$
	Optionnel : élimination d'arcs	$(g,f,8),(d,f,17),(e,f,18),(c,f,32)$
#8	Choix de $(g,f,8)$ pour relier f	$(d,f,17),(e,f,18),(c,f,32)$
#9	Fin	

(f) L'arbre de recouvrement minimal trouvé en (e) est-il unique ? Justifiez votre réponse. [1 point]

Oui, la solution est unique. En simulant l'algorithme de Kruskal, lorsqu'on a le choix entre plusieurs arêtes, toutes ces arêtes se retrouvent dans la solution. Étapes : #1 choix unique $(a,b,2)$; #2,#3, #4 $(a,c,5)$, $(d,e,5)$, $(d,e,5)$ (peut importe l'ordre, on doit tous les choisir) ; #5 choix unique $(c,d,6)$; #6 choix unique $(e,h,7)$; #7 choix unique $(g,f,8)$. Il n'arrive aucun cas où qu'une arête candidate est non choisie.
L'explication peut être plus simple.

4 Résolution d'un problème [4 points]

Dans un logiciel de type chiffrier ou tableur (ex. : Microsoft Excel, Lotus 1-2-3, LibreOffice Calc, etc.), on peut écrire des formules dans les cellules. Une formule peut référer à d'autres cellules. Par exemple, la cellule `A3` est égale à `A1+A2`, soit la valeur 3. Les formules des cellules doivent être évaluées afin d'afficher les résultats à l'écran.
Il peut arriver que des formules soient **incalculables** à cause de dépendances cycliques. Dans l'exemple ci-dessous, la cellule `A5` est incalculable, car elle fait référence à elle-même. Il va de même pour les cellules `H3` et `H4` qui se réfèrent mutuellement. Les cellules `E2`, `E3`, `E4` et `E5` sont aussi incalculables en raison d'un cycle. Une cellule dont la formule réfère une à cellule incalculable est aussi incalculable (ex. : `H5`).

	A	B	C	D	E	F	G	H	I	J	K	...
1	1	"chaine"	1									
2	2		=C1+1	=C2*2	=D2+E5							
3	=A1+A2		=C2+1	=C3*2	=E2			=H4				
4			=C3+1	=C4*2	=E3			=H3				
5	=A5		=C4+1	=C5*2	=E4			=moyenne(H3 :H4)				

Dans un logiciel de type chiffrier, la détection des cellules incalculables est essentielle. Cela peut être fait en détectant les cycles dans le graphe orienté $G = (V, E)$, où les sommets V sont les cellules de la feuille de calcul, et les arêtes E marquent les relations de dépendance. Vous devez implémenter cette fonctionnalité dans le logiciel basé sur le code suivant.

```
1  class Feuille{ // type 'Feuille de Calcul'
2  public:
3    struct Coordonnee{ // Coordonnee de cellule
4      char colonne; // modele simple avec 26 colones A...Z.
5      int ligne;
6      bool operator < (const Coordonnee& autre) const;
7    };
8    set<Coordonnee> getCellulesIncalculables() const;
9    // ...
10 private:
11   struct Cellule{
12     list<Coordonnee> getDependences(); //retourne cellules mentionnees dans formule
13     string           formule;
14     // ...
15   };
16   map<Coordonnee, Cellule> cellules;
17 };
```

Sur page suivante, vous devrez implémenter la fonction `Feuille::getCellulesIncalculables()`. Si vous jugez que des modifications à la classe `Feuille` sont nécessaires, énumérez-les ici. Un changement peut être l'ajout d'une variable ou d'une fonction, le changement d'un type de retour, etc. Comme il y a plusieurs réponses possibles, il pourrait n'y avoir aucun changement. Vous ne pouvez ni supposer, ni utiliser une classe `Graphe`.

Version simple : On itére sur toutes les cellules. Pour chaque cellule, on vérifie sa calculabilité avec une recherche en profondeur. Dans la recherche en profondeur, on marque les cellules (sommets du graphe) en cours de visite. Dès qu'on visite une cellule qui est déjà en cours de visite, une dépendance cyclique est détectée. Tout ce qui est sur la pile d'exécution (en cours de visite) est incalculable.
Dans la déclaration, on ajoute :
- `mutable bool envisite;` dans `Cellule`.
- `bool estCalculable(const Coordonne&) const;` dans `Feuille`.

Pour un maximum de 3 points sur 4, vous pouvez expliquer dans vos propres mots, plutôt que d'écrire du code, comment vous réaliseriez cette fonction.

```
1  set<Coordonnee> Feuille::getCellulesIncalculables() const{
2      set<Coordonnee> result;
3      map<Coordonnee,Cellule>::const_iterator iter=cellules.begin();
4      for(;iter!=cellules.end();++iter)
5          if(!estCalculable(iter->first))
6              result.insert(iter->first);
7      return result;
8  }
9  bool Feuille::estCalculable(const Coordonne& coor) const{
10     const Cellule& cellule = cellules[coor];
11     if(cellule.envisite) return false;
12     cellule.envisite=true;
13     bool calculable = true;
14     list<Coordonnee> deps=cellule.getDependences();
15     for(list<Coordonnee>::iterator i=deps.begin();i!=deps.end();++i)
16         calculable &= eslCalculable(*i);
17     cellule.envisite=false;  // tres important
18     return calculable;
19 }
```

Version efficace : La version simple fait des calculs redondants. Il est possible de les éviter en marquant les cellules calculables et incalculables dès que leur état de calculabilité est connue. Dans la déclaration, on ajoute :

— `enum {INCONNU, ENCOURS, CALCULABLE, INCALCULABLE} etat;` dans `Cellule`.
— `bool estCalculable(const Coordonne&) const;` dans `Feuille` (privée).

```
1  set<Coordonnee> Feuille::getCellulesIncalculables() const{
2      map<Coordonnee,Cellule>::const_iterator iter=cellules.begin();
3      for(;iter!=cellules.end();++iter)
4          iter->second.etat=INCONNU;
5      set<Coordonnee> result;
6      for(iter=cellules.begin();iter!=cellules.end();++iter)
7          if(!estCalculable(iter->first))
8              result.insert(iter->first);
9      return result;
10 }
11 bool Feuille::estCalculable(const Coordonne& coor) const{
12     const Cellule& cellule = cellules[coor];
13     if(cellule.etat==CALCULABLE) return true;
14     if(cellule.etat==ENCOURS) cellule.etat=INCALCULABLE;
15     if(cellule.etat==INCALCULABLE) return false;
16     assert(cellule.etat==INCONNU);
17     cellule.etat==ENCOURS;
18     bool calc = true;
19     list<Coordonnee> deps=cellule.getDependences();
20     for(list<Coordonnee>::iterator i=deps.begin();calc&&i!=deps.end();++i)
21         calc &= eslCalculable(*i);
22     cellule.etat = calc ? CALCULABLE : INCALCULABLE;
23     return calc;
24 }
```

Correction : 2 points pour le principe ; 1 pour fonctionnement correct ; 1 point pour l'efficacité.

/********** Fin de l'examen ! **********/