Composition d'Informatique
Mécanismes de la programmation orientée-objet (INF371)
Promotion 2021

27 juin 2022

Les parties sont indépendantes entres elles et les exercices indépendants entre eux.

1 Exercices

Question 1 Donnez trois instructions XVM qui peuvent causer un débordement de pile (*overflow*). ◇

Solution. PUSH, FETCH(0), GSB, PRX, ALLOC, RFR...

Question 2 L'exécution de l'instruction POP peut-elle causer une erreur ? Et alors quand ? ◇

Solution. Oui,si la pile ne contient d'élément (stack underflow). □

Question 3 Que donne l'exécution du code suivant :

```
static String f() { return ("a" + f()); }
public static void main(String[] args) {
  System.out.println(f()); }
```

Solution. Un débordement de pile (et c'est tout). □

Question 4 Que se passe-t-il quand on compile et exécute le code suivant :

```
try {    System.out.print("Hello " + 1 / 0);   }
catch (ArithmeticException e)
   {  System.out.println("World ! ");   }
```

Solution. Ca compile bien. L'évaluation de l'argument du premier print déclenche une exception avant qu'on ait affiché quelque chose. Donc on voit juste `World !` à l'écran. □

2 Compilation des `switch`

Pour simplifier, on considèrera un seul exemple de type énuméré, à savoir les jours de la semaine :

```
enum Jour = { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
```

On se rappelle qu'on utilise souvent, avec les types énumérés, la construction `switch` :

```
switch (e) {
  case LUNDI : p0;
     break;
  case MARDI : p1;
     break;
     ...
  case DIMANCHE : p6;
     break;
   }
```

On s'intéresse à comment compiler le code source ci-dessus, sachant que :
— `e` est une expression de type `Jour`,
— `p0`, `p1`, `p2`,. . .,`p6` sont des instructions ou séquences d'instructions.

Un premier collègue, A, propose d'utiliser simplement le schéma de compilation habituel, en compilant le code ci-dessus comme on compilerait :

```
if (e == LUNDI) { p0; }
else if (e == MARDI) { p1; }
else if (e == MERCREDI) { p2; }
...
else { p6; }
```

Un autre collègue, B, propose de se ramener plutôt à ce code source qui utilise une variable locale :

```
{ Jour j = e;
    if (j == LUNDI) { p0; }
    else if (j == MARDI) { p1; }
    else if (j == MERCREDI) { p1; }
    ...
    else if (j == SAMEDI) { p5; }
    else { p6; } }
```

où `j` est un nom de variable qui n'est pas utilisé ailleurs.

Question 5 Que pouvez-vous dire de la différence entre ces deux approches ? Laquelle est meilleure ? Sont-elles toutes les deux correctes ? ◇

Solution. La solution de B est correcte. Celle de A est (1) moins efficace car la valeur de `e` est calculée sept fois et (2) incorrecte lorsque ce calcul a des effets de bord (modifie la mémoire) □

Un collègue C trouve la solution précédente pas assez efficace et peu élégante. Il regrette qu'il faille jusqu'à 6 tests pour "trouver" le code exécuté, et que certaines branches (comme `LUNDI`) sont trouvées avant d'autres (`DIMANCHE`).

On veut donc exploiter la représentation des valeurs de type `Jour` pour proposer des schémas de compilation plus efficaces. On sait donc que `LUNDI` est représenté par 0 dans le code machine, `MARDI` par 1, jusqu'à `DIMANCHE` qui est représenté par 6.

Dans un premier temps, C propose d'exploiter le fait que les représentations machine des `Jour` sont ordonnées, pour que le code du switch fasse au plus 3 comparaisons (en fait fasse au plus $\log_2(n)$ comparaisons pour un type énuméré à n valeurs).

Question 6 Donnez le début de cette manière de compiler le switch (disons les six à huit premières instructions machines - il s'agit de montrer que vous avez compris, sans se perdre dans des détails bureaucratiques). ◇

Solution. On fait des tests d'inégalité dichotomiques sur la valeur de `e`. Ca commence, par exemple, par :

$[\![e]\!]$
FETCH(0)
PUSH(3)
LEQ
GTZ(α)
FETCH(0)
PUSH(5)
LEQ
GTZ(β)

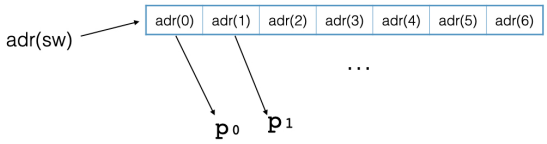


FIGURE 1 – table pour la compilation du `switch` (question 7)

Un autre collègue propose une autre approche qui, après discussion, convainc tout le monde :
— Les branches `p0`, `p1`... sont compilées et les codes machines correspondants sont placés respectivement aux adresses adr(0), adr(1), ... adr(6).
— Ces adresses sont elles-mêmes rangées dans un bloc mémoire placé en adr(*sw*). C'est-à-dire que adr(*sw*) contient la valeur adr(0), adr(*sw* + 1) contient adr(1), *etc.* ... Voir figure 1.
— Le switch lui-même sera alors compilé en utilisant un adressage indirect, similaire à ce qui est fait pour l'appel des méthodes.

Question 7 Proposez un code machine pour le code source ci-dessus à partir de là. ◇

Solution.

PUSH(adr(*sw*))
⟦e⟧
CREAD
GTO

On remarque qu'il faut ajouter un GTO à la fin de chaque bloc `p0`, `p1`... pour continuer l'exécution au bon endroit. □

3 Trouver les vélos les plus proches

Dans cette partie, une grande importance sera accordée à la clarté du code. Vous essayerez également de le rendre aussi concis que possible.

On rappelle en appendice un certain nombre de fonctions ou de classes de bibliothèques Java qui peuvent être utilisées dans ce problème.

Une municipalité ouvre un service de vélos en libre service. Les vélos peuvent être déposés et empruntés n'importe où (pas de bornes).

Les points de la carte sont désignés par deux coordonnées de type `double`.

On voudra afficher tous les vélos disponibles sur un rectangle (correspondant à un écran). Pour cela on va utiliser la distance entre deux positions (x, y) et (x', y') définie par $\max(|x-x'|, |y-y'|)$ (voir figure 2).

Question 8 Définissez une classe `Position` composée des champs `x` et `y` qui corresponde à un point sur la carte. Munissez-la d'une méthode `double distance(Position a)` qui calcule la distance avec la position `a`. ◇

Solution.

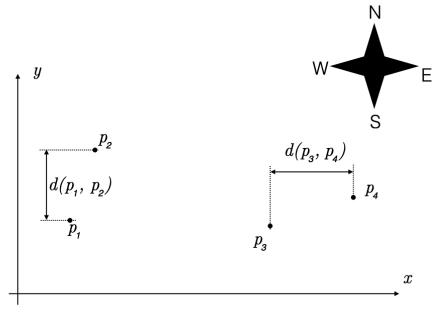


Figure 2 - Distances entre deux positions

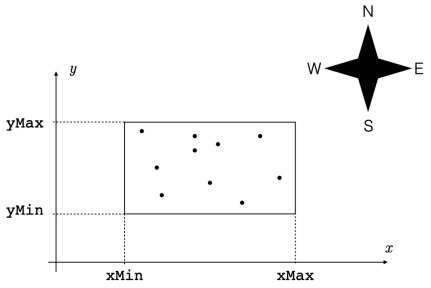


Figure 3 - Visualisation des champs de la carte

```
class Position {
    double x;
    double y;

    Position(double vx, double vy) { x = vx; y = vy; }

    double distance(Position p) {
        return(Math.max(Math.abs(x-p.x), Math.abs(y-p.y)));
    }
}
```

Question 9 Définissez une classe `Velo` qui étend la classe `Position` et comporte en plus un champ `public int iD` qui correspond au numéro d'identification du vélo. Munissez la classe d'un constructeur. ◇

Solution.
```
class Velo extends Position {
    final int iD;
    Velo (int i, double x, double y) { super(x,y); iD = i; }
}
```

Question 10 On veut être sûr qu'un numéro d'identification n'est attribué qu'une seule fois. Comment procédez-vous pour cela ?

Modifiez le constructeur pour qu'il refuse d'utiliser un `iD` qui a déjà été attribué (il pourra lancer une `Error` dans ce cas). ◇

Solution. On va garder un `HashSet<Integer>` des immatriculations déjà attribuées :

```
static HashSet<Integer> immat = new HashSet<Integer>();
```

et on fait la vérification et la mise à jour dans le constructeur :

```
Velo (int i, double x, double y) {
        super(x,y);
        if (immat.contains(i)) throw new Error("immat");
        immat.add(i);
         iD = i; }
}
```

Une `Carte` va être une structure contenant tous les vélos d'un rectangle. On donne pour cela la classe abstraite suivante :

```
abstract class Carte {
    double xMax, xMin, yMax, yMin;
    abstract LinkedList<Velo> closeBy(Position a, double distance);
    abstract void add(Velo r);

    protected int card;
    public int card() { return card; }
}
```

Une carte va donc avoir comme champs les coordonnées délimitant ce rectangle (voir figure 3). De plus, on veut avoir les opérations suivantes sur une carte :

— Pouvoir ajouter un vélo sur la carte (méthode `add`).
— Donner la liste de tous les vélos de la carte situés à une distance inférieure ou égale à `d` d'une position `a` avec la méthode `closeBy`.
— Enfin le champ `card` contiendra le nombre de vélos présents dans le rectangle.

Question 11 Ajoutez à la classe abstraite `Carte` une méthode concrète `double distance(Position a)` qui donne la distance entre la position `a` et le rectangle de la carte[1]. ◇

Solution. On utilise le fait qu'on peut ajouter une méthode concrète dans une classe abstraite.

```
 double distance(Position a) {
        return(Math.max(Math.max(Math.max(xMin - a.x, 0), Math.max(a.x - xMax, 0)),
                        Math.max( Math.max(yMin - a.y, 0) , Math.max(a.y - yMax, 0))));
    }
```

Question 12 Donnez une implémentation simple de la classe abstraite `Carte`. On appellera cette classe `Base`. Cette classe pourra, par exemple, contenir la liste des vélos présents dans le rectangle.

Le constructeur `Base(double x1, double x2, double y1, double y2)` devra construire une carte vide, sans vélos. ◇

Solution.

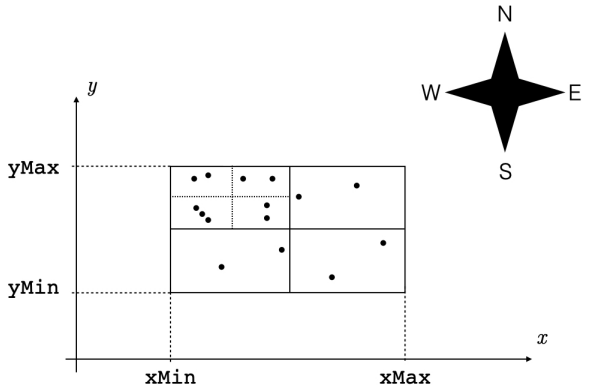
```
class Base extends Carte {
    private LinkedList<Velo> contenu;

    Base(double x1, double x2, double y1, double y2)
        {this.xMax = x2; xMin = x1; yMin = y1; yMax = y2;
            card = 0;
            contenu = new LinkedList<Velo>();
          }

    LinkedList<Velo> closeBy(Position a, Double distance) {
        LinkedList<Restaurant> l = new LinkedList<Velo>();
        for (Restaurant r : contenu) if (r.distance(a) <= distance) l.add(r);
        return(l);
      }

    void add(Velo r) {
      card++;
      contenu.add(r);
    }
}
```

1. C'est-à-dire entre `a` et les points du rectangle qui sont les plus proches de `a`.



Les quatre secteurs de la partition de la carte. Ici le secteur nord-ouest comporte beaucoup de vélos et peut être lui-même partitionné en quatre sous-parties.

Figure 4

Question 13 Quelles sont les complexités des méthodes `add` et `closeBy` de votre implémentation ? ◇

Solution. Ici, respectivement $O(1)$ et $O(n)$, où n est le nombre de vélos présents dans le rectangle.□

On veut rendre plus efficace l'opération de recherche de vélos (méthode `closeBy`). Pour cela, on va :

1. se donner une borne supérieure sur le nombre de vélos qui peuvent être présents dans une instance de `Base`. Appelons `nMax` cette borne supérieure.
2. Lorsqu'une carte comporte plus de `nMax` vélos, on découpe en quatre rectangles de surfaces égales le rectangle de la carte. Ces quatre rectangles correspondront donc aux secteurs nord-est, nord-ouest, sud-est et sud-ouest. Pour chacun de ces rectangles, on aura une carte contenant exactement les vélos appartenant à ces rectangles.
3. Pour chacune de ces cartes, si le rectangle comporte moins de `nMax` vélos, elle sera représentée par une instance de `Base`. Sinon la carte sera elle-même composée de 4 cartes, et ainsi de suite.

Ces cartes correspondent donc à des arbres dont les nœuds ont quatre fils et les feuilles sont des `Base`. On les appelle des *quadtrees*.

La figure 4 illustre la partition en quatre parties de la carte.

Voici un début de définition de la classe `QuadTree` :

```
class QuadTree extends Carte {

    static final int nMax = 10;
    private Carte[] cartes;
```

On comprend donc que le tableau `cartes` contiendra toujours 4 éléments.

Question 14 Commencez par compléter cette définition avec le constructeur `QuadTree(double xMax, double xMin, double yMax, double yMin)` qui construira une carte dont les quatre secteurs seront des `Base` vides. ◇

Solution.

```
public QuadTree( double xMax,double xMin,double yMax,double yMin) {
      card = 0;
      this.xMin = xMin;
      this.xMax = xMax;
      this.yMin = yMin;
      this.yMax = yMax;
      cartes = new Carte[4];
      cartes[0] = new Base(xMax, (xMin+xMax / 2), yMax, (yMax+yMin) / 2);
      cartes[1] = new Base(xMax, (xMin+xMax / 2), (yMax+yMin) / 2, yMin);
      cartes[2] = new Base((xMin+xMax / 2), xMin, (yMax+yMin) / 2, yMin);
      cartes[3] = new Base((xMin+xMax / 2), xMin, yMax, (yMax+yMin) / 2);
    }
```

Question 15 Ecrivez maintenant la méthode `closeBy` de `QuadTree`. ◇

Solution.

```
public LinkedList<Velo> closeBy(Position a, double d) {
    LinkedList<Velo> r = new LinkedList<Velo>();
    for (int i = 0; i < cartes.length; i++)
        if (cartes[i].distance(a) <= d)
            r.addAll(cartes[i].closeBy(a, d));
    return r;
}
```

Question 16 Sans justifier précisément, et sans rentrer dans le détail, quelle complexité pourra-t-on espérer pour `closeBy` si la distance de recherche n'est pas trop grande et que les vélos sont uniformément distribués sur la ville ? ◇

Solution. Dans certains cas, on va avoir quatre appels récursifs (si la position est proche du milieu de la carte). Mais en général on peut espérer éliminer trois ou deux parmi les quatre sous-cartes. La complexité pratique devrait donc être logarithmique par rapport au nombre de vélos.

Elle devrait également être proportionnelle à `nMax`. □

Question 17 Ecrivez une fonction `static Carte smartAdd(Carte c, Velo r)` qui rende la Carte équivalente à `c` à laquelle on aura ajouté le vélo `r`. Le point important est que si `c` est de la classe `Base` et que le nombre de vélo dépasse `nMax`, alors on devra créer un QuadTree. ◇

Solution.

```
static Carte smartAdd(Carte c, Velo v) {
     c.add(v);
     if (!(c instanceof Base) ||  (c.card() < nMax)) return(c);
     Carte n = new QuadTree(c.xMax, c.xMin, c.yMax, c.yMin);
     for (Velo w : ((Base)c).contenu)
         n.add(w);
     return n;
}
```

Question 18 Avec ce qui précède, terminez l'implémentation de la classe `QuadTree` en écrivant sa méthode `add`. ◇

Solution.

```
public void add(Velo v) {
      for  (int i = 0; i < cartes.length; i++)
          if (cartes[i].distance(v) <= 0) {
              cartes[i] = smartAdd(cartes[i], v);
              card++;
              return;
          }
      throw new Error("no map - should not happen");
  }
```

Question 19 Pouvez-vous dire quelque chose de la complexité de cette dernière méthode ? (en supposant que les vélo sont distribués uniformément à travers la ville) ◇

Solution. La taille des `Base` étant bornée par la constante `nMax`, le temps pour les `split` est aussi borné par une constante. La complexité est donc logarithmique par rapport au nombre de vélos. □

Question 20 Ajoutez aux classes `Base` et `QuadTree` une méthode `boolean remove(Velo v)` qui :
- lorsque le vélo est présent dans la carte le retire et renvoie `true`,
- lorsqu'il n'est pas présent laisse la carte inchangée et renvoie `false`.

On pourra supposer que le test d'égalité structurelle pour `Velo` a été bien implémenté et qu'un vélo est présent au plus une fois dans la carte. ◇

Solution. Dans `Base` :

```
      public boolean remove(Velo v) {
       if (contenu.remove(v)) {
           card --;
           return true;
       }
       return false;
  }
```

dans `QuadTree` :

```
public boolean remove(Velo v) {
      for (Carte c : cartes)
          if (c.distance(v) == 0 && c.remove(v)) {
              card --;
              return true;
          }
      return false;
  }
```

Question 21 Supposons que l'on veuille changer le type de la méthode `closeBy` pour qu'elle prenne aussi en argument un niveau de charge de la batterie du vélo. Comment pourrait-on organiser les données pour rendre la recherche encore plus efficace ? ◇

Solution. Ajouter la charge dans la recherche revient à ajouter une dimension à l'espace dans lequel on cherche les vélos. On pourrait donc imaginer des cartes qui sont bornées non seulement par des x et des y minimaux et maximaux, mais également des charges minimales et maximales. Dans ce cas, une carte non basique serait découpé en huit sous-cartes, et non pas en quatre.

Remarquons que c'est une question théorique dont l'intérêt pratique est discutable !

Les complexités asymptotiques resteraient inchangées. □

Appendice

Fonctions mathematiques Java

On dispose en particulier des fonctions statiques suivantes :

`double Math.abs(double x)`	valeur absolue
`double Math.max(double x, double y)`	maximum de deux nombres
`double Math.min(double x, double y)`	minimum de deux nombres

La classe `LinkedList<E>`

Elle est munie des méthodes suivantes :

`add(E e)`	qui ajoute l'élément `e`
`E remove()`	qui enlève et renvoie le premier élément de la liste
`addAll(Collection<E> l)`	qui ajoute à la liste tous les éléments de `l`
`boolean remove(Object o)`	qui enlève l'objet `o` de la liste et renvoie `true`, si `o` fait partie de la liste, et renvoie `false` et laisse la liste inchangée sinon.

Pour cette dernière méthode, on pourra supposer qu'elle vérifie l'égalité structurelle sur les objets.

On rappelle que `LinkedList` est une implémentation de `Collection` donc l'argument de `addAll` peut être une `LinkedList`.

On rappelle aussi qu'on peut itérer sur les éléments d'une `LinkedList` avec la notation `for (E e : l) ...`.

La classe `HashSet<E>`

Elle est munie des methodes suivantes :

`add(E e)` qui ajoute l'élément `e` à l'ensemble
`boolean contains(E e)` qui vérifie si `e` appartient à l'ensemble.