INF431: Algorithmes et Programmation: du séquentiel au distribué

Promotion X2005

F. Morain J.-M. Steyaert

Gr. 1: L. Mauborgne, F. Magniez, Q. Thai
Gr. 2: P. Jacquet, P. Chassignet, B. Tomchuk
Gr. 3: A. Cohen, D. Rossin, B. Bernardo
Gr. 4: F. Pottier, J. Cervelle, S.-Y. Cho
Gr. 5: N. Sendrier, A. Ribès, A. Saurin

Plan

I. Organisation du cours.

II. Souvenirs et nouveautés de Java.

III. Code et pseudocode.

IV. Complexité élémentaire des algorithmes.

V. Utilisation des classes prédéfinies de Java.

VI. Soit E un ensemble.

I. Organisation du cours

- 18 blocs (8+1+8+1), avec amphi de 1h30 et TD ou PC de 2 heures. On trouve tout sur la Page web du cours.
- TD:
  - programmes à déposer d'une fois sur l'autre (upload), relus par les équipes pédagogiques (2h + 2h → A, B, C, D, E).
  - Deux expériences: wifi; eclipse.
- Pour les PC : exercices à rendre (2h + 2h → A, B, C, D, E).
- Délégué(e)s: un(e) ancien(ne) 321, un(e) ancien(ne) 311, un(e) EV2.

Pour mercredi prochain.

Un cours en deux parties

Partie I: (F. Morain) terminer la première phase de l'apprentissage de la programmation et de l'algorithmique.

Sanction: pale CC1 + projet (sujets donnés le 22 février; choix (binômes) pour le retour des vacances de février; à rendre pour le 29 mai; soutenances entre le 11 et le 22 juin.)

Partie II: (J.-M. Steyaert) réseaux, concurrence, logique.

Sanction: pale CC2.

Notes finales:

- Note classante finale $CC = (CC_1 + 2CC_2)/3$;
- Note (littérale) de module $= (PI + 2CC)/3$ plus note de contrôle continu ($\in \{-1, 0, 1\}$).

Emploi du temps prévisionnel

31/01 – Bloc 01 : Java I (amphi + td)
07/02 – Bloc 02 : Java II (amphi + td)
14/02 – Bloc 03 : Analyse lexicale (amphi + td)
Vacances de février
20/02 – Bloc 04 : Analyse syntaxique (amphi + td)
07/03 – Bloc 05 : Graphes I (amphi + td)
14/03 – Bloc 06 : Graphes II (amphi + PC)
21/03 – Bloc 07 : Graphes III (amphi + td)
28/03 – Bloc 08 : Graphes IV (amphi + PC)
04/04 – Bloc 09 : amphi d'ouverture (+ PC?).
Vacances de printemps
18/04 – Bloc 10 : transmission du flambeau à J.-M. Steyaert.
25/04 – Bloc 11 : etc.

La première partie du cours

La programmation est une des activités les plus complexes jamais entreprises par l'homme.

Windows XP = 50 millions lignes de code,

Linux = 30 millions lignes de code,

Génie logiciel (*Software engineering*):
Spécification → Programmation → Correction

Question primordiale: comment passe-t-on à l'échelle?

Le cursus d'apprentissage de la programmation

- Soit **`t`** un tableau;
- Soient **`l`** une liste, **`a`** un arbre;
- Soit **`g`** un graphe.

- Écrire une méthode;
- Écrire une classe;
- Écrire un ensemble de classes.

Buts: simplicité de la signature des fonctions; élégance de l'écriture en général.

Leitmotiv1: le programmeur du IIIème millénaire utilise au maximum les bibliothèques (classes prédéfinies) du langage.

Leitmotiv2: on injecte de plus en plus d'algorithmique et de complexité.

II. Souvenirs et nouveautés de Java

```
class Bonjour{
  public static void main(String[] args){
    System.out.println("Encore lui!");
  }
}
```

```
unix% javac Bonjour.java
unix% java Bonjour
```

Utiliser les possibilités de Java 1.5

Java évolue rapidement, avec pour but de faciliter la vie du programmeur.

La syntaxe tend à se simplifier, des fonctionnalités nouvelles apparaissent: auto-boxing (`Integer i = 3;`), génériques, etc.

Dans le cours, on utilisera Java 1.5 avec certaines de ses nouveautés. On peut programmer à l'ancienne ou tester la modernité. C'est affaire de goût, du moment que les programmes marchent!

Rem. on s'éloigne de plus en plus des langages préhistoriques (FORTRAN) ou ancestraux (C, etc.), on se rapproche de langages comme C++.

Entrées et sorties

En Java 1.5, on utilise de préférence un **`Scanner`** qui est facile à utiliser:

```
import java.io.*;
import java.util.*;

class EssaiScan{

  public static void main(String[] args)
                      throws IOException{
    Scanner s =
      new Scanner(new BufferedReader
                       (new FileReader(args[0])));

    while(s.hasNext())
      System.out.println(s.next());
    s.close();
  }
}
```


On peut faire mieux, en utilisant des méthodes plus précises:

```
  public static void main(String[] args)
                      throws IOException{
    Scanner s =
      new Scanner(new BufferedReader
                       (new FileReader(args[0])));

    while(s.hasNext()){
      if(s.hasNextInt())
        System.out.println(s.next());
      else
        s.next();
    }
    s.close();
  }
```


Les exceptions en Java

But: rattraper (certaines) erreurs de façon à ne pas faire planter le programme.

```
public static void main(String[] args){
  try{
    int x = Integer.parseInt(args[0]);
    System.out.println(x);
  } catch (NumberFormatException e) {
    System.err.println("Mauvais argument: "
                       + e.getMessage());
  } catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Mauvais nombre d'arguments");
  } finally {
    System.out.println("On s'en est sorti");
  }
}
```

Exceptions $\Rightarrow$ Isolement du cas anormal.
Le cas normal s'écrit indépendamment du cas anormal.

On utilise **`finally`** pour effectuer une opération finale s'il y a exception ou pas (cela évite la duplication de code).

Parenthèse: le fonctionnement des exceptions

Une exception se propage dans le bloc de la méthode où elle est levée. Si elle n'est pas récupérée dans le bloc, elle se propage à la méthode d'appel et ainsi de suite.

Si aucune méthode ne la récupère, elle arrive dans **`main`**, et c'est l'interpréteur Java qui la récupère, affiche la pile d'appels avant de sortir.

```
On s'en est sorti
Exception in thread "main" java.lang.NumberFormatException: 1
        at Excep.parseInt(Excep.java:7)
        at Excep.main(Excep.java:15)
```


III. Code et pseudocode

Les algorithmes vus jusqu'à présent étaient relativement simples, donc on pouvait les décrire directement en Java.

Certains algorithms du cours sont plus complexes, et un vrai langage à tendance à cacher la structure.

$\Rightarrow$ introduction d'un pseudocode très proche de Java, mais sans le typage.

Dans le cours: très souvent du pseudocode, accompagné parfois du programme Java. Ceux-ci sont disponibles dans le poly.

Les règles du jeu complètes sont dans le poly.

Exemple typique:

```
rechercheMinimum(A)
  imin <- 0;
  pour i <- 1 à longueur(A)-1 faire
    si A[i] < A[imin] alors
      // on a trouvé un minimum local
      imin <- i;
  retourner A[imin];
```

Rem.

- Ce sera également le langage très utilisé en PC, dans les pales.
- Attention à l'équilibre des détails: le pseudocode sera le plus proche possible du programme. À éviter:

  ```
  resolutionProbleme(A)
      retourner solution(A);
  ```

  ça ne passera pas... Vous serez très guidé(e)s et on verra beaucoup d'exemples en amphi.

IV. Complexité élémentaire des algorithmes

Ce que veut le client: combien de temps pour résoudre mon problème sur mon ordinateur?

Réponse: $T(P) = \mathscr{O} \cdot f(P)$ avec $\mathscr{O}$ qui dépend de l'ordinateur (fréquence d'horloge), et $f(P)$ qui dépend des algorithmes choisis pour résoudre P. *In fine*, c'est $T(P)$ qui nous intéresse souvent.

On remplace le temps par le nombre d'opérations élémentaires effectuées (affectations, comparaisons, $+$, $*$, etc.), puis il suffit de mesurer le temps pris par celles-ci sur une machine donnée.

Ex. Multiplication matrice vecteur: $\forall i \in [0, n-1], w_i = \sum_{k=0}^{m-1} A_{i,k} v_k$. Chaque w_i nécessite m multiplications et additions, soit nm multiplications et nm additions, dans tous les cas.

Complexité d'un algorithme: comment évolue le temps passé par un algorithme en fonction de la taille des données d'entrée? (passage à l'échelle?)

Calcul exact rare: on peut s'intéresser au cas le plus favorable, le cas le pire, ou le cas moyen.

Complémentaire à la correction d'un programme: il fait bien ce que l'on veut, mais en combien de temps?

Algorithmique: comment résoudre un problème donné ? Par quelles méthodes (algorithmes) ?

Quels critères? facilité d'implantation, taille mémoire requise, vitesse.

Ex. Nombres de Fibonacci.

Sujet de recherche: quel est le temps minimal nécessaire pour résoudre un problème donné?

Ex. Voyageur de commerce. Cf. Majeure 2.

Calculs élémentaires de complexité

Concaténation d'instruction: $T(P;Q) = T(P) + T(Q)$.

Itération: $T(\text{for(i = 0; i < n; i++) P(i);}) = \sum_{i=0}^{n-1} T(P(i))$.

Comparaisons asymptotiques: si $f, g \geqslant 0$, on écrit $f = O(g)$ ssi f/g est borné à l'infini:

$$\exists n_0, \quad \exists K, \quad \forall n \geqslant n_0, \quad 0 \leqslant f(n) \leqslant K g(n).$$

Rem. On calcule le coût de toutes les opérations; souvent on ne considère que les coûts dominants.

Exemples de complexités

$T(n) = O(1)$	:	temps constant (accès à une variable; rare !)
$T(n) = O(\log n)$	:	temps logarithmique (dichotomie)
$T(n) = O(n)$	:	temps linéaire
$T(n) = O(n \log n)$	:	tris rapides
$T(n) = O(n^2)$	:	temps quadratique (tri sélection)
$T(n) = O(n^k)$	:	temps polynomial
$T(n) = O(2^n)$	:	temps exponentiel

Ex. $n = 1000 \Rightarrow n^2 = 10^6$ faisable; 2^{64} infaisable (2^{64} s $> 5.8 \cdot 10^{11}$ années; âge de l'univers $\approx 15 \cdot 10^9$ années).

Comparaison et affectation

Pour les entiers, c'est à peu près le même coût.

Il y a beaucoup de cas où une comparaison coûte souvent plus cher qu'une affectation, par exemple la comparaison de **`String`** (min des longueurs).

De l'utilité des références:

```
String[] s = {"abcdef", "x"};
String tmp = s[0];
s[0] = s[1];
s[1] = tmp;
```

$\Rightarrow$ on comprend l'utilité de manipuler des références...

C'est le cas pour les tableaux, les matrices, etc.

Vous avez dit facile?

Recherche du minimum d'un tableau: `t` de taille n:

```
int mint;
mint = t[0]; // 1 affectation
// n-1 affectations et comparaisons
for(int i = 1; i < t.length; i++)
  if(t[i] < mint) // n-1 comparaisons
    mint = t[i]; // F(t) affectations
```

La quantité $F(t)$ dépend fortement de **`t`**:

$$0 \leqslant F(t) \leqslant n-1$$

et on peut montrer que sa valeur moyenne (sur tous les tableaux de taille n) vaut $\log n$...!

V. Utilisation des classes prédéfi nies de Java

Idée: les langages modernes (C, C++, OCaml, Java) arrivent avec de nombreuses bibliothèques regroupant des structures de données classiques. Il est inutile de tout recommencer à zéro tout le temps.

Avantages: partage de code; concision des programmes; rapidité de prototypage.

Inconvénients: dans les applications critiques, il vaut mieux reprogrammer soi-même certaines classes (listes d'entiers).

Premier exemple

Tableau de taille variable.

```
import java.util.*;

public class Essai{
  public static void main(String[] args){
    Vector<Integer> t = new Vector<Integer>(10);

    for(int i = 0; i < 10; i++)
      t.set(i, i*i);
    for(int i = 0; i < t.size(); i++)
      System.out.println(t.get(i));
  }
}
```

Second exemple: la classe **LinkedList**

```
import java.io.*;
import java.util.*;

class EssaiLL{
  public static void main(String[] args){
    LinkedList<String> l=new LinkedList<String>();

    for(int i = 0; i < 5; i++)
      l.addFirst("a"+i);
    // affichage stupide et destructif
    try{
      while(true)
        System.out.println(l.remove());
    }
    catch(NoSuchElementException e){ }
    }
}
```


En utilisant un itérateur (non destructif):

```
ListIterator li = l.listIterator(0);
while(li.hasNext())
  System.out.println(li.next());
```

ou de façon plus compacte:

```
for(ListIterator li = l.listIterator(0);
    li.hasNext(); )
  System.out.println(li.next());
```

Syntaxe 1.5 typique:

```
for(String s : l)
  System.out.println(s);
```

qui est la traduction du pseudocode:

```
pourtout s dans l faire
    écrire s;
```


Les génériques de Java

(aperçu à compléter en amphi02)

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Queue<E>
```

E est un mot clef désignant un type, prédéfini ou non:

```
class Amoi{
  int i;
  Amoi(int n){ i = n; }
}
class TestAmoi{
  public static void main(String[] args){
    LinkedList<Amoi> l = new LinkedList<Amoi>();
    l.addFirst(new Amoi(1));
    for(Amoi a : l)
      System.out.println(a.i);
  }
}
```


Utilisation des listes

- Stocker un nombre arbitraire d'éléments.
- Liste doublement chaînée avec tête et queue.
- Permet d'implanter les piles (LIFO), les files (FIFO): chaque opération se fera en $O(1)$.
- Cf. la doc de Java.

VI. Soit E un ensemble

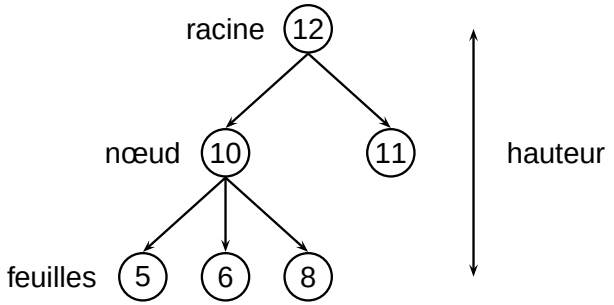
Dans la suite du cours, nous aurons à utiliser des ensembles avec diverses propriétés.

- Si $|E|$ est petit, un tableau de taille fixe peut suffire.
- Si l'ordre n'a pas d'importance: liste (en Java, **`LinkedList`** qui implante des listes doublement chaînées, *double ended queues* – dequeue).
- Si on doit tester l'appartenance souvent:
  - $O(\log|E|)$ s'il existe un ordre et si les éléments sont triés (avec un coût initial $O(|E|\log|E|)$ en utilisant des arbres.
  - $O(1)$ par hachage. En Java, **`HashSet`** ou **`Hashtable`**.

A) Exemple de révision: arbres

Un *arbre* est défini comme étant $\emptyset$ ou bien une structure contenant :

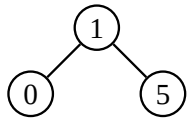
- un élément *racine* ;
- un ensemble d'arbres $(S_1, S_2, \ldots, S_n)$ attachés à la racine r.



Symboliquement, on peut écrire : $\mathscr{A} = (r, S_1, S_2, \ldots, S_n)$.

Les différents parcours d'arbre

But : on désire examiner tous les nœuds d'un arbre $A = (r, S_1, S_2, \ldots, S_n)$ une fois et une seule.

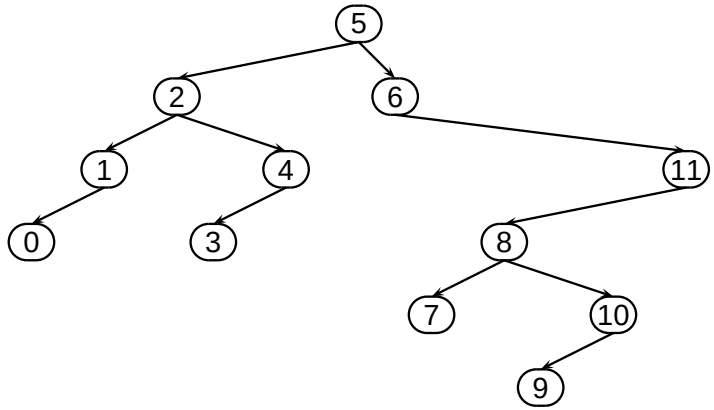


- ordre préfixe : $\pi((r, S_1, S_2, \ldots, S_n)) = (r, \pi(S_1), \pi(S_2), \ldots, \pi(S_n))$;
  Ex. : $1, 0, 5$.
- ordre suffixe : $\sigma((r, S_1, S_2, \ldots, S_n)) = (\sigma(S_1), \sigma(S_2), \ldots, \sigma(S_n), r)$;
  Ex. : $5, 0, 1$.
- ordre infixe (surtout intéressant quand $n = 2$) :
  $i((r, S_1, S_2)) = (i(S_1), r, i(S_2))$.
  Ex. : $1, 5, 0$.

Arbre binaire de recherche

Déf. un arbre binaire de recherche est tel que pour chaque nœud v, tous les nœuds u du sous-arbre gauche sont inférieurs à v et tous les nœuds w du sous-arbre droit sont supérieurs ou égaux à v.

Ex.



Applications : stockage dynamique de dictionnaires, avec opérations en $O(\log n)$.

```
public class Abr{
  int n;
  Abr filsg, filsd;

  Abr(int nn, Abr fg, Abr fd){
    n = nn; filsg = fg; filsd = fd;
  }
  static Abr inserer(Abr a, int n){
    if(a == null) return new Abr(n, null, null);
    if(n < a.n)
      a.filsg = inserer(a.filsg, n);
    else
      a.filsd = inserer(a.filsd, n);
    return a;
  }
  public static void main(String[] args){
    int[] t = {5, 2, 6, 1, 4, 3, 0, 11, 8, 7, 10, 9};
    Abr a = null;

    for(int i = 0; i < t.length; i++)
      a = inserer(a, t[i]);
  }
}
```



```
static Abr estDans(Abr a, int n){
  if(a == null) return null;
  if(a.n == n) return a;
  if(n < a.n) return estDans(a.filsg, n);
  else return estDans(a.filsd, n);
}
```

Prop. Si A est équilibré, le temps d'insertion et de recherche est en $O(\log n)$.

Exercice: montrer comment trier un tableau avec un abr.

B) File de priorité

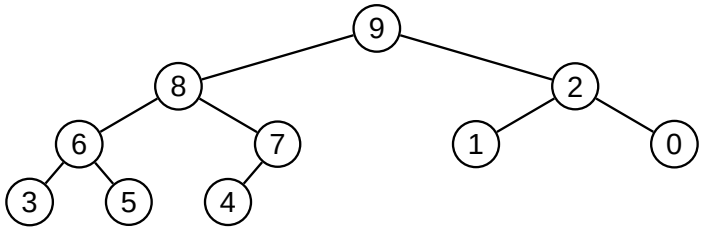
Déf. une structure permettant de savoir en $O(\log n)$ qui est le chef.

Utilisé pour les files d'attente (impression, mail, ...).

Une des implantations possibles: un tableau `t[1..n]` satisfait la propriété de tas si et seulement si pour tout $1 \leqslant i < n/2$, on a

`t[i]` $\geqslant \max($`t[2*i]`$,$`t[2*i+1]`$)$.

Dans une représentation graphique par arbre, cela équivaut à dire que le père est plus grand que ses deux fils.



En Java: `TreeSet` (avec des AVL). On en verra une utilisation plus loin.

C) comment marche le hachage

Principe: on associe à chaque objet un entier unique. On peut se servir de cet entier pour stocker l'objet par exemple dans un tableau.

Comment calculer cet entier? Par exemple une chaîne de caractères peut être transformée en entier, puis l'entier réduit si besoin.

Ex: $C(\text{abcd}) = 97 \cdot 256^3 + 98 \cdot 256^2 + 99 \cdot 256 + 100 =$
$(((97 \cdot 256) + 98)256 + 99)256 + 100 = 1633837924.$

Généralement: on travaille avec un tableau de taille fixe, typiquement un nombre premier p et on calcule $h(s) = C(s) \bmod p$.

Ex: avec $p = 10007$, on trouve $h(\text{abcd}) = 5041$.

Collision: $h(\text{bacn}) = 5041$, etc.

$\Rightarrow$ mécanisme de gestion de collisions:

- utilisation de listes contenant les collisions avec la même adresse;
- hachage ouvert: on cherche le premier indice libre à partir de l'adresse de départ.

Rem. Marche très bien si le hachage est bon et garanti des additions/tests/suppressions en $O(1)$. Cf. Knuth par exemple.

Exemple d'utilisation du hachage

Problème: on rentre une liste de chaînes et on ne veut garder qu'un seul exemplaire de chaque chaîne.

Principe du traitement: on passe en revue chaque chaîne; si elle n'est pas déjà dans la table, on la stocke. À la fin, on affiche le contenu de la table.

```
unique(dico)
t <- table_de_hachage;
pourtout s dans dico faire
    si s n'est pas dans t alors
        ajouter s dans t;
pourtout s dans t faire
    écrire s;
```

En Java: chaque objet possède une méthode **`hashCode()`**. Cette méthode est utilisée par toutes les classes qui dont du hachage (**`HashSet`**, etc.).


```
import java.io.*;
import java.util.*;

public class Chaines{
  public static void main(String[] args){
    String[] dico = {"toto", "titi", "tutu",
                     "toto", "tata"};
    HashSet<String> t = new HashSet<String>();

    for(int i = 0; i < dico.length; i++)
      if(! t.contains(dico[i]))
        t.add(dico[i]);
    for(String s : t)
      System.out.print(s+" ");
  }
}
```

```
tutu tata toto titi
```


Table clé-valeurs

Ex: On émule un dictionnaire français-anglais. On stocke un mot et sa traduction anglaise.

```
import java.io.*;
import java.util.*;

public class Traduction{
  public static void main(String[] args){
    String[] dicof={"maison","manger","Villepin"};
    String[] dicoe={"house", "eat", "Blair"};
    Hashtable<String,String> t =
        new Hashtable<String,String>();

    for(int i = 0; i < dicof.length; i++)
      t.put(dicof[i], dicoe[i]);
    for(String s : t.keySet())
      System.out.println(s+" -> "+t.get(s));
  }
}
```

Les nouveautés en TD

- **Expérience 1:** wifi pour ceux qui veulent venir avec leur portable.
- **Expérience 2:** ECLIPSE (environnement de développement tout intégré de qualité professionnelle avec éditeur, compilateur, débogueur) pour ceux qui le souhaitent.

Résumé du cours

- Organisation du cours.
- Rechargeons Java et les structures de données élémentaires en mémoire.
- Complexité.
- Classes prédéfinies.
- Rappels d'algorithmique.

Prochains rendez-vous: TD cet après-midi; amphi 02 mercredi prochain 07/02.

Rappel: noms des délégué(e)s pour le début du prochain amphi. Un délégué par groupe.