Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Rue Royale, 67. 1000 Bruxelles
02/219.15.46 – esi@he2b.be

Algorithmique

2020

Bachelor en Informatique
DEV2

A. Paquot (APA), M. Codutti (MCD), N. Richard (NRI),
S. Drobisz (SDR), S. Rexhep (SRE) & ???

Document produit avec LaTeX.
Version du 24 décembre 2019.

Table des matières

Chapitre 1

Les traitements de rupture

Dans ce chapitre, nous allons étudier une classe de problèmes qui peuvent tous se résoudre avec un même type d'algorithme : l'algorithme de rupture.

Considérons un problème comme celui-ci :

> *« Soit une liste d'étudiants, où un étudiant est représenté par un objet reprenant son nom, son numéro, son option et son année. Écrire un algorithme qui compte le nombre d'étudiants dans chaque section et, plus précisément, dans chaque année de chaque section. »*

Nous verrons que l'algorithme de rupture sera adapté à la résolution de ce problème lorsque la liste est triée d'une certaine manière. C'est pourquoi nous allons commencer par parler du *classement complexe* des éléments.

À la fin du chapitre vous devrez être capable de :

- ▷ Détecter qu'on se trouve bien face à un problème qui peut entrer dans le cadre d'un algorithme de rupture et identifier si le tri des éléments est adéquat.
- ▷ Adapter le squelette général de l'algorithme de rupture au problème donné.

1.1 Le classement complexe

Introduction

Dans le chapitre sur les tris du cours d'algorithmique I (DEV1), vous avez abordé naturellement la notion du classement des données. Néanmoins, les données étaient « simples » : nombres ou chaines, pour lesquelles la relation d'ordre est évidente. Les algorithmes mis en œuvre peuvent facilement s'adapter pour d'autres types, par exemple des objets Date, où l'opérateur de comparaison est remplacé par la méthode « estAntérieure() ».

Mais, plus généralement, les données composées de plusieurs éléments – comme les attributs d'un objet – ne possèdent pas de relation d'ordre naturelle. C'est le cas, par exemple, des points d'un espace à deux ou trois dimensions ou encore des informations figurant sur une carte d'identité. Si on veut ordonner une série de telles données, il faudra choisir un premier critère de classement (par exemple le nom ou la date de naissance) et en cas d'égalité sur le premier critère (deux personnes peuvent avoir un même nom ou être nées le même jour), il faudra départager sur un second critère, et ainsi de suite.

Ces critères de classement sont bien entendu arbitraires, et dépendent de l'information qu'on veut retirer de l'ensemble des données. Notons aussi que l'ordre de classement peut être, pour chaque critère, croissant ou décroissant.

Exemple de classement simple

Prenons l'exemple d'un objet `Etudiant`, contenant les attributs `matricule`, `nom`, `prénom`, `dateNaissance` et `option` (G, I ou R). Pour l'exemple, considérons une liste de 6 étudiants :

matricule	nom	prénom	dateNaissance	option
29845	Durant	Kevin	20/01/94	R
30125	Dupont	Fabrice	13/06/94	G
30351	Simon	André	18/11/94	G
30597	Dupont	Charles	9/07/94	G
31857	Guilmant	Léon	17/03/96	R
31886	Durant	Sam	30/05/94	I

Cette liste est classée sur le numéro de matricule. C'est un classement simple réalisé sur un seul champ des données. Le numéro de matricule étant dans ce cas-ci un **identifiant** des données, le problème de devoir départager ne se pose pas.

Exemple de classement double

Si nous désirons à présent classer sur l'ordre alphabétique des noms, il faut décider de départager, en cas de noms identiques, sur un autre champ, de façon naturelle sur celui des prénoms. Ceci donnerait le classement double suivant, en **majeur** sur le nom et en **mineur** sur le prénom :

matricule	nom	prénom	dateNaissance	option
30597	Dupont	Charles	9/07/94	G
30125	Dupont	Fabrice	13/06/94	G
29845	Durant	Kevin	20/01/94	R
31886	Durant	Sam	30/05/94	I
31857	Guilmant	Léon	17/03/96	R
30351	Simon	André	18/11/94	G

Exemple de classement triple

Supposons enfin que nous voulions grouper les étudiants par sections, nous devons alors classer prioritairement sur l'option, départager sur les noms et ensuite sur les prénoms. C'est alors un classement triple : en **majeur** sur l'option, en **médian** sur le nom et en **mineur** sur le prénom :

matricule	nom	prénom	dateNaissance	option
30597	Dupont	Charles	9/07/94	G
30125	Dupont	Fabrice	13/06/94	G
30351	Simon	André	18/11/94	G
31886	Durant	Sam	30/05/94	I
29845	Durant	Kevin	20/01/94	R
31857	Guilmant	Léon	17/03/96	R

Remarque : un classement n'est pas forcément un classement alphabétique. Par exemple, dans le cas du classement sur l'option, toute autre permutation des lettres G, I, R serait un tri possible.

Résumé

Les exemples ci-dessus constituent des exemples de **classements complexes**. On dira que des données sont classées sur la **clé composée** champ 1 – champ 2 – … – champ i – … – champ n (où « champ i » est un attribut des données) si le classement se fait prioritairement depuis le champ 1 jusqu'au champ n. Autrement dit, si deux données ont tous leurs champs 1, 2,…, i égaux ($i < n$), le classement se fait en départageant sur le champ $i + 1$. L'indice du champ correspond au **niveau** du classement complexe.

1.2 La notion de rupture

Les algorithmes que nous allons voir peuvent s'appliquer à n'importe quel **ensemble logique** d'éléments qui peut faire l'objet d'un traitement séquentiel (les listes, les tableaux...) [1]. Les éléments peuvent être de n'importe quel **type complexe**. Nos exemples seront souvent pris sur des listes de objets.

Nous parlons de **rupture** lorsque, dans ce traitement itératif, on constate que l'information courante que l'on souhaite traiter n'appartient plus à l'ensemble (ou au sous-ensemble) des informations déjà traitées précédemment.

Lorsque les données sont triées selon une clé complexe, il est naturel de parler de *rupture sur un champ* de ces données, ou de rupture de niveau donné (voir section suivante).

Par exemple, dans le dernier classement des étudiants ci-dessus, il y a rupture sur l'option au niveau de Durant Sam et de Durant Kevin. En effet, ces deux étudiants délimitent les sous-ensembles d'étudiants partageant une même option.

Dans le 2e classement, nous pouvons parler de rupture sur les noms : l'étudiant Durant Kevin met fin au sous-ensemble des Dupont, et l'étudiant Guilmant met fin à celui des Durant.

Dans le 1er classement, qui est un classement simple sur le numéro de matricule, on peut considérer qu'il n'y a qu'un ensemble de données d'un seul tenant sans ruptures (ou avec rupture de niveau 0, voir section suivante), ou alors qu'il y a rupture à chaque étudiant, puisque chaque étudiant forme un sous-ensemble isolé par son numéro de matricule, vu qu'ils sont obligatoirement distincts (mais dans ce cas nous n'utilisons pas l'algorithme de rupture : c'est une simple itération sur la liste).

1.3 Traitement des ruptures dans une séquence ordonnée

Rupture de niveau 0

Quel que soit l'ordre de tri des données de l'ensemble parcouru séquentiellement, il est toujours possible de détecter la fin des données grâce à sa taille [2]. Cette « fin » constitue donc la rupture principale, celle signalant la fin du parcours itératif.

Sans le savoir, nous avons donc déjà traité la rupture générale d'un ensemble de données (c'est la rupture de « niveau 0 », car elle n'est pas liée à un champ des données, et est naturellement prioritaire sur ces champs). Pour illustrer cela, reprenons l'exemple de la liste d'étudiants. Le parcours de base de cet ensemble est le suivant :

```
algorithm RuptureNiveau0(etudiants: List of Etudiant)
    ∀ étudiant : étudiants // pour tout étudiant de la liste des étudiants
        // traitement de l'étudiant
    end
end
```

Un tel algorithme se réaliserait naturellement avec un boucle for. Nous allons pourtant la réaliser avec un while car cela permet de généraliser l'algorithme à des ruptures de plusieurs niveaux. Ce qui donne :

1. Et également les fichiers moyennant une petite adaptation liée au fait qu'on ne connait pas, dans ce cas, la taille de l'ensemble lorsqu'on commence à le traiter.
2. Ou via une marque spéciale de *fin de fichier* dans les cas des fichiers séquentiels.

```
algorithm RuptureNiveau0(etudiants: List of Etudiant)
    i: integer
    i = 0
    while i < etudiants.size()
        // traitement de etudiants.get(i)
        i++
    end
end
```

Si nous voulons faire des statistiques globales sur l'ensemble des étudiants (par ex. simplement les compter), le traitement de l'information consiste à incrémenter un compteur, et l'algorithme ci-dessus peut fonctionner quel que soit l'ordre de classement choisi.

Rupture de niveau 1

Passons à présent au « niveau 1 » ; c'est-à-dire un traitement de rupture correspondant à un classement complexe sur un champ. Imaginons que nous voulions savoir quel est le nombre d'étudiants dans chaque option. Une solution, sans algorithme de rupture, consisterait à avoir 3 compteurs, un par option. On peut imaginer une façon plus judicieuse de faire à partir du dernier classement, celui qui contient précisément les étudiants déjà groupés par option : à chaque fois qu'il y a rupture sur l'option, on peut alors connaitre le total d'étudiants dans l'option qui vient d'être parcourue. Ceci ne nécessite qu'un seul compteur remis à 0 à chaque fois qu'une nouvelle option est rencontrée, c'est-à-dire à chaque rupture. De plus l'algorithme fonctionnera quel que soit le nombre d'options.

En gros, ça donne :

```
algorithm RuptureNiveau1(etudiants: List of Etudiant)
    // on suppose les données classées en majeur sur l'option
    ∀ étudiant : étudiants
        cptEtudiantsOption = 0
        ∀ étudiants de la même option
            cptEtudiantsOption++
        end
        print cptEtudiantsOption et l'option
    end
end
```

Retenons

Lorsqu'on calcule une variable liée à un niveau de rupture – dans notre exemple, le nombre d'étudiants par option est lié à l'option, donc au niveau 1 – la variable :

- ▷ est initialisée juste avant d'entrer dans la boucle du niveau ;
- ▷ est mise à jour dans la boucle ;
- ▷ a sa valeur finale quand la boucle du niveau est finie.

Soyons plus précis dans notre algorithme :

```
algorithm RuptureNiveau1(etudiants: List of Etudiant)
    // on suppose les données classées en majeur sur l'option
    saveOption: string
    cptEtudiantsOption: integer
    i: integer

    i = 0                                                    // En brun ce qui est lié au niveau 0
    while i < etudiants.size()
        saveOption = etudiants.get(i).getOption()            // En orange ce qui est lié au niveau 1
        cptEtudiantsOption = 0                               // En bleu ce qui est lié à la question posée
        while  i < etudiants.size()  ET  etudiants.get(i).getOption() = saveOption
            cptEtudiantsOption++
            i++
        end
        print cptEtudiantsOption, « étudiant dans l'option », saveOption
    end
end
```

Questions de réflexion :

- ▷ pourquoi la condition i < etudiants.size() apparait-elle une 2e fois dans la boucle intérieure ?
- ▷ pourquoi est-ce que i et cpt ne sont pas initialisés au même endroit ?
- ▷ pourquoi l'incrémentation de i se fait-elle dans la boucle centrale et pas ailleurs ?
- ▷ pourquoi utilise-t-on saveOption plutôt que etudiants.get(i).option dans l'instruction d'affichage ?
- ▷ l'ordre des conditions apparaissant dans le 2e « tant que » est-il important ?

Voici le même algorithme écrit en Java :

```
1   public static void nbEtudiantsParOption(List<Etudiant> étudiants) {
2       int cptEtudiantsOption, i;
3
4       i = 0;
5       while (i < étudiants.size()) {
6           String saveOption = étudiants.get(i).getOption();
7           cptEtudiantsOption = 0;
8           while (i < étudiants.size() && étudiants.get(i).getOption().equals(saveOption)) {
9               cptEtudiantsOption++;
10              i++;
11          }
12          System.out.println("Il y a " + cptEtudiantsOption
13                  + " étudiants dans l'option " + saveOption);
14      }
15  }
```

Rupture de niveau 2

L'algorithme ci-dessus se généralise facilement si on ajoute davantage de niveaux de rupture. Pour illustrer le « niveau 2 », prenons encore l'exemple suivant : on veut connaitre pour chaque option le nombre d'étudiants nés dans les différentes années de naissance. L'algorithme correspondant s'écrit facilement et fonctionne lorsque les données sont, cette fois-ci, classées en majeur sur l'option et en mineur sur la date de naissance (ou encore classement double sur la clé composée option – dateNaissance).

En voici une première version, schématique :

```
algorithm RuptureNiveau2(etudiants: List of Etudiant)
    // on suppose les données classées en majeur sur l'option
    // et en mineur sur la date de naissance (ordre chronologique)
    ∀ étudiant : étudiants
        ∀ étudiants de la même option
            cptEtudiantsAnnéeOption = 0
            ∀ étudiants de la même année dans l'option
                cptEtudiantsAnnéeOption++
            end
            print cptEtudiantsAnnéeOption, l'année et l'option
        end
    end
end
```

Remarquez bien où la variable cptEtudiantsAnnéeOption est : initialisée, mise à jour, affichée. Voici une version plus détaillée :

```
algorithm RuptureNiveau2(etudiants: List of Etudiant)
    // on suppose les données classées en majeur sur l'option
    // et en mineur sur la date de naissance (ordre chronologique)
    saveOption: string
    saveAnnéeNaissance: integer
    cptEtudiantsAnnéeOption: integer
    i: integer
    i = 0
    while i < etudiants.size()
        saveOption = etudiants.get(i).getOption()
        while i < etudiants.size() ET etudiants.get(i).getOption() = saveOption
            saveAnnéeNaissance = etudiants.get(i).getNaissance().getAnnée()
            cptEtudiantsAnnéeOption = 0
            while i < etudiants.size() ET etudiants.get(i).getOption() = saveOption ET
                  etudiants.get(i).getNaissance().getAnnée() = saveAnnéeNaissance
                cptEtudiantsAnnéeOption++
                i++
            end
            print cptEtudiantsAnnéeOption, " étudiant dans l'option ", saveOption, " sont nés en ",
                  saveAnneeNaissance
        end
    end
end
```

Ces exemples montrent que l'algorithme de rupture et le tri des listes sont étroitement liés. La structure de l'algorithme épouse le schéma de la clé composée du classement des données, et à un classement déterminé correspondra un algorithme bien précis.

Le nombre d'étudiant par option ET par année de l'option.

Comment faire si on veux **à la fois** le nombre d'étudiants par option et le nombre d'étudiants par année (de l'option) ? Voici une solution schématique. Remarquez bien à quels endroits se font les initialisations, les mises à jour et les affichages.

```
algorithm RuptureNiveau2bis(etudiants: List of Etudiant)
    ∀ étudiant : étudiants
        cptEtudiantsOption = 0
        ∀ étudiants de la même option
            cptEtudiantsAnnéeOption = 0
            ∀ étudiants de la même année dans l'option
                cptEtudiantsAnnéeOption++
                cptEtudiantsOption++
            end
            print cptEtudiantsAnnéeOption, l'année et l'option
        end
        print cptEtudiantsOption et l'option
    end
end
```

Vous pouvez remarquer que la mise à jour se fait dans la boucle la plus interne pour les deux variables ! Dans cet exemple précis de comptage, on pourrait mettre-à-jour le nombre d'étudiants par option de façon plus efficace.

```
algorithm RuptureNiveau2bis(etudiants: List of Etudiant)
    ∀ étudiant : étudiants
        cptEtudiantsOption = 0
        ∀ étudiants de la même option
            cptEtudiantsAnnéeOption = 0
            ∀ étudiants de la même année dans l'option
                cptEtudiantsAnnéeOption++          // Plus de mise à jour de cptEtudiantsOption ici !
            end
            print cptEtudiantsAnnéeOption, l'année et l'option
            cptEtudiantsOption += cptEtudiantsAnnéeOption          // La mise à jour se fait ici
        end
        print cptEtudiantsOption et l'option
    end
end
```

1.4 Exercices

Exercice 1

La chasse au gaspi [rupture de niveau 1]

À l'ÉSI, les quantités de feuilles imprimées et photocopiées par les professeurs et les étudiants sont enregistrées à des fins de traitement. Le service technique désirant facturer les « exagérations », vous fournit une liste de toutes les impressions effectuées depuis le début de l'année. Cette liste est composée d'objets de la classe Job suivante et est ordonnée alphabétiquement **en majeur** sur le champ login :

```
class Job
    private :
        login: string
        date: date
        nombre: integer
    public :
        // Le constructeur et les getteurs
end
```

Écrire un algorithme permettant d'afficher une ligne (avec login et nombre) par utilisateur dont le nombre total de feuilles imprimées dépasse une valeur limite entrée en paramètre.

Exercice 2 Nombre d'étudiants par année et par option

Reprenons l'exemple donné pour la rupture de niveau 2 (RuptureNiveau2, page 8). Et si, à présent, on veut **à la fois** le nombre d'étudiant par option **et** le nombre d'étudiants par année (**toutes les années confondues**) ? Comment faire ?

Exercice 3 Compter les étudiants [rupture de niveau 2]

Supposons que la classe Etudiant contienne également un attribut indiquant dans quel bloc se trouve l'étudiant (1, 2 ou 3). On voudrait un algorithme qui reçoit une liste d'étudiants et calcule le nombre d'étudiants dans chaque section et, par section, dans chaque bloc.

L'affichage ressemblera à :

```
Gestion
    bloc 1 : 130 étudiants
    bloc 2 : 42 étudiants
    bloc 3 : 16 étudiants
    TOTAL  : 188 étudiants
Industriel
    bloc 1 : 32 étudiants
    bloc 2 : 14 étudiants
    bloc 3 : 8 étudiants
    TOTAL  : 54 étudiants
Réseau
    bloc 1 : 82 étudiants
    bloc 2 : 31 étudiants
    bloc 3 : 13 étudiants
    TOTAL  : 126 étudiants
```

a) Quel doit-être le tri pour pouvoir résoudre cet exercice avec un algorithme de rupture ?

b) Écrire cet algorithme.

Exercice 4 Les fanas d'info [rupture de niveau 2]

Une grande société d'informatique a organisé durant les douze derniers mois une multitude de concours ouverts aux membres de clubs d'informatique. Elle souhaiterait récompenser le club qui aura été le plus « méritant » durant cette période au point de vue de la participation des membres mineurs. Chaque résultat individuel des participants (y compris des majeurs) est repris dans une liste dont les éléments sont de type Participant.

```
class Participant
    private :
        nom: string                          // nom et prénom du participant
        âge: integer                         // âge du participant au moment du concours
        référence: string                    // référence du club auquel appartient ce participant
        numéro: integer                      // numéro du concours auquel il a participé
        résultat: integer                    // résultat obtenu lors de ce concours (sur 100)
    public :
        // Le constructeur et les getteurs
end
```

Sachant que la liste est ordonnée **en majeur sur le champ référence et en mineur sur le champ nom**, on demande d'écrire l'algorithme qui affiche les informations suivantes :

pour chaque club :

- ▷ sa référence
- ▷ pour chaque membre mineur de ce club :
  - ▷ son nom et prénom
  - ▷ la cote moyenne sur 100 des concours auquel ce membre a participé
- ▷ le nombre total de participations des membres mineurs

N.B. : un membre mineur qui s'est inscrit à un concours = une participation. Un club qui n'aura eu aucun membre mineur participant figurera quand même dans le résultat avec la mention « Pas de participation de membre mineur ». Par contre, un club dont aucun membre n'a participé au moindre concours ne sera pas affiché.

À la fin, on affichera la référence du meilleur club, à savoir celui qui a eu la plus haute cote moyenne de membres mineurs (simplifions on ne gérant pas les possibles ex-æquo).

Exercice 5 Éliminer les doublons d'une liste.

Soit une liste ordonnée d'entiers avec des possibles redondances. Écrire un algorithme qui enlève les redondances de la liste. On vous demande de créer une nouvelle liste (la liste de départ reste inchangée).

Exemple : si la liste est $(1, 3, 3, 7, 8, 8, 8)$ le résultat sera $(1, 3, 7, 8)$.

Exercice 6 Une suite logique

Voici une petite suite logique :

```
1
1 1
2 1
1 2 1 1
1 1 1 2 2 1
3 1 2 2 1 1
1 3 1 1 2 2 2 1
1 1 1 3 2 1 3 2 1 1
3 1 1 3 1 2 1 1 1 3 1 2 2 1
...
```

a) Comprendre la logique derrière cette suite et écrire la ligne suivante.

b) Écrire un algorithme qui reçoit une ligne (sous forme d'une liste d'entiers) et retourne la ligne suivante (sous forme d'une autre liste d'entiers). Votre première tâche sera probablement de comprendre ce que vient faire cet exercice dans le chapitre des ruptures puisque la liste n'est pas triée.

c) Écrire l'algorithme qui reçoit N (un entier) et affiche les N premières lignes de cette suite logique.

Exercice 7 Alternative à la rupture

Reprenons l'exemple donné pour la rupture de niveau 1 (RuptureNiveau1, page 7). Supposons que la liste ne soit **pas** triée sur l'option. Écrivez l'algorithme qui calcule le nombre d'étudiants par option en un seul parcours de la liste (vous devrez utiliser trois compteurs distincts).