# UE J1MI2013 : Algorithmes et Programmes

Alain Griffault
Université Bordeaux

19 mai 2015

# Index

# Table des matières

# Liste des Algorithmes

# Table des figures

# Informations générales

## 2014-2015 - Cours 1

### Pages accessibles sur la toile

- Les pages pour les TDs, TPs et annales
- Le programme et les modalités officielles
- Mes pages pour l'UE

### Modalités

| Type épreuve | Durée | Nombre | Coefficient |
|---|---|---|---|
| Tests | 15 minutes | 3 | 0.15 |
| Devoir surveillé | 1 heure 30 | 1 | 0.25 |
| TP noté | 1 heure 30 | 1 | 0.20 |
| Devoir surveillé terminal | 2 heures | 1 | 0.40 |

### Équipe pédagogique

**Cours** Alain Griffault. 2011-2012, 2012-2013, 2013-2014, 2014-2015.

**Travaux dirigés**

- Olfa Ben-Ahmed. 2014-2015.
- Olivier Baudon. 2014-2015.
- Giuliana Bianchi. 2011-2012, 2012-2013, 2013-2014, 2014-2015.
- Laetitia Bourgeade. 2013-2014.
- Marie-Christine Counilh. 2011-2012, 2012-2013.
- Philippe Duchon. 2012-2013, 2013-2014, 2014-2015.
- Irène Durand. 2011-2012, 2012-2013, 2013-2014.
- Julien Ferte. 2012-2013.
- Cyril Gavoille. 2014-2015.
- Noël Gillet. 2014-2015.
- Charles Huet. 2011-2012.
- Jérôme Kirman. 2013-2014.
- Pauline Mouawad. 2014-2015.
- Thomas Place. 2013-2014, 2014-2015.
- Jean-Claude Ville. 2011-2012.
- Marc Zeitoun. 2014-2015.
- Anna Zukhova. 2012-2013.

# Chapitre 1

# Rappels UE J1MI1003 du semestre 1

## 1.1 Fonctionnement (simplifié) d'un ordinateur

*Je parle de processeur avec des registres, de mémoire avec des mots mémoires et d'instructions qui transfèrent de la mémoire vers les registres, calculent avec les registres, transfèrent des registres vers la mémoire. Cela me permet de définir l'affectation, la complexité en espace et la complexité en temps. J'introduis également la notion de programmation impérative associée à cette vision changement d'état de la mémoire.*

## 1.2 Notations utilisées pour l'affectation et pour les algorithmes

**Algorithme 1.1:** NotationsAffectations

```
x ← 3;                      /* Langage algorithmique */
x := 3;                     /* Ada, Pascal */
x = 3;                      /* C, Java, Python...  */
```

**Algorithme 1.2:** NotationsAlgorithmes

**Données** : Les paramètres
**Résultat** : Le(s) résultat(s)

```
instruction-1;              /* ‘‘;’’ séparateur */
...;
instruction-n;              /* ‘‘;’’ terminateur */
retourner variable(s)
```

Je préfère l'utilisation d'un langage algorithmique “à la pascal". J'utiliserais python lorsque je ferai des démonstrations.

## 1.3 Les structures de contrôle d'un programme

### 1.3.1 La séquence

Dans un langage algorithmique, on utilise très souvent le “ ;” pour séparer deux instructions qui doivent s'exécuter en séquence. De nombreux langages de programmation utilisent également le “ ;”. Le langage `python` utilise des règles d'indentation très fortes. Le séparateur n'est pas un caractère “visible”, mais le retour à la ligne.

```python
def sequence():
    x = 2
    y = 3
    z = x+y
    x = x+1
```

| temps | x | y | z |
|---|---|---|---|
| 1 | 2 | | |
| 2 | 2 | 3 | |
| 3 | 2 | 3 | 5 |
| 4 | 3 | 3 | 5 |

### 1.3.2 Notion de prédicats

**Rappels** Un prédicat est une expression booléenne qui est soit “vraie”, soit “fausse”, soit “non définie”.

- En logique, les opérateurs sont $\neg, \vee, \wedge$
- En langage algorithmique, les opérateurs sont `not, and, or, &, |, &&, ||`

– En programmation, la valeur “non définie” interrompt l’exécution du programme et provoque un “Bug”. Les opérateurs sont `not, and, or`, &, &&, |, ||

| $P$ | Logique algorithme $\neg P$ | valuation not $P$ | valuation paresseuse not $P$ |
|---|---|---|---|
| $T$ | $F$ | $F$ | $F$ |
| $F$ | $T$ | $T$ | $T$ |
| $ND$ | $ND$ | *Bug* | *Bug* |

| $P$ | $Q$ | Logique algorithme $P \vee Q$ | Logique algorithme $P \wedge Q$ | valuation $P \vert Q$ | valuation $P\&Q$ | valuation paresseuse $P \vert\vert Q$ | valuation paresseuse $P\&\&Q$ |
|---|---|---|---|---|---|---|---|
| $T$ | $T$ | $T$ | $T$ | $T$ | $T$ | $T$ | $T$ |
| $T$ | $F$ | $T$ | $F$ | $T$ | $F$ | $T$ | $F$ |
| $T$ | $ND$ | $T$ | $ND$ | *Bug* | *Bug* | $T$ | *Bug* |
| $F$ | $T$ | $T$ | $F$ | $T$ | $T$ | $T$ | $T$ |
| $F$ | $F$ | $F$ | $F$ | $F$ | $F$ | $F$ | $F$ |
| $F$ | $ND$ | $ND$ | $F$ | *Bug* | *Bug* | *Bug* | $F$ |
| $ND$ | $T$ | $T$ | $ND$ | *Bug* | *Bug* | *Bug* | *Bug* |
| $ND$ | $F$ | $ND$ | $F$ | *Bug* | *Bug* | *Bug* | *Bug* |
| $ND$ | $ND$ | $ND$ | $ND$ | *Bug* | *Bug* | *Bug* | *Bug* |

**Attention :** Pour les langages de programmation qui utilise l’évaluation paresseuse (python, C, JAVA, ...), les opérateurs logiques ne sont pas commutatifs. Les algorithmes vus en cours peuvent nécessiter d’être adaptés pour être corrects dans ces langages.

## 2014-2015 - Cours 2

## predicats

```python
def pair(n):
    return n%2==0
```

```python
from predicats import *

print("pair(5) = %s" %pair(5))
print("pair(10) = %s" %pair(10))
```

```
pair(5) = False
pair(10) = True
```

## evaluationParesseuse

```python
def correctOr():
    P = True
    return P or Q

def correctAnd():
    P = False
    return P and Q

def bugOr():
    P = False
    return P or Q

def bugAnd():
    P = True
    return P and Q
```

```
import sys
from evaluationParesseuse import *

print("correctAnd = %s"\
        %correctAnd())
print("correctOr = %s"\
        %correctOr())
try:
    print("bugAnd = %s" %bugAnd())
except:
    print("Unexpected error:")
    print("\t", sys.exc_info()[0])
    print("\t", sys.exc_info()[1])
try:
    print("bugOr = %s" %bugOr())
except:
    print("Unexpected error:")
    print("\t", sys.exc_info()[0])
    print("\t", sys.exc_info()[1])
```

```
correctAnd = False
correctOr = True
Unexpected error:
        <class 'NameError'>
        global name 'Q' is not defined
Unexpected error:
        <class 'NameError'>
        global name 'Q' is not defined
```

### 1.3.3 Le branchement

**Objectif :** Permettre que l'exécution d'un programme ne soit pas toujours la même séquence d'instructions.

**Algorithme 1.3:** BranchementsSemantique(P1,P2,P3)

**Données** : Une liste de prédicats
**Résultat** : La branche exécutée

**si** *P1* **alors** `/* Exécutée ssi P1 */`
  branche ← 1;
**sinon si** *P2* **alors** `/* Exécutée ssi ¬ P1 ∧ P2 */`
  branche ← 2;
**sinon si** *P3* **alors** `/* Exécutée ssi ¬ P1 ∧ ¬ P2 ∧ P3 */`
  branche ← 3;
**sinon** `/* Exécutée ssi ¬ P1 ∧ ¬ P2 ∧ ¬ P3 */`
  branche ← 4;
**retourner** *branche*

**Exemples :** Quelques algorithmes avec branchements.

**Algorithme 1.4:** Civilite(masculin,nom,prenom)

**Données** : Les attributs d'une personne
**Résultat** : Une phrase

**si** *masculin* **alors**
  ecrire('Bonjour Monsieur ',prenom,' ',nom);
**sinon**
  ecrire('Bonjour Madame ',prenom,' ',nom);

**Algorithme 1.5:** Minimum2V1(X,Y)

**Données** : Deux nombres
**Résultat** : Le plus petit

**si** $X \leq Y$ **alors**
  min ← X;
**sinon**
  min ← Y;
**retourner** min

**Algorithme 1.6:** Minimum2V2(X,Y)

**Données** : Deux nombres
**Résultat** : Le plus petit

**si** $X \leq Y$ **alors**
  **retourner** X
**sinon**
  **retourner** Y

**Algorithme 1.7:** Minimum2V3(X,Y)

**Données** : Deux nombres
**Résultat** : Le plus petit

**si** $X \leq Y$ **alors**
  **retourner** X
**retourner** Y

**Algorithme 1.8:** Minimum3V1(X,Y,Z)

**Données** : Trois nombres
**Résultat** : Le plus petit

**si** $X \leq Y$ **alors**
  **si** $X \leq Z$ **alors**
    **retourner** X
  **sinon**
    **retourner** Z
**sinon**
  **si** $Y \leq Z$ **alors**
    **retourner** Y
  **sinon**
    **retourner** Z

**Algorithme 1.9:** Minimum3V2(X,Y,Z)

**Données** : Trois nombres
**Résultat** : Le plus petit

**si** $X \leq Y \ ET \ X \leq Z$ **alors**
  **retourner** X
**sinon si** $Y \leq Z$ **alors**
  **retourner** Y
**sinon**
  **retourner** Z

**Les programmes en annexe**

- Une version python

### 1.3.4 La boucle “pour”

**Objectif :** Éviter l’écriture de nombreuses suites d’instructions similaires. Pour cela, un ensemble ou bien une liste contient les “variables” pour lesquelles il faut “itérer” la suite d’instructions. La liste ou l’ensemble est connu au début de l'instruction, et donc le nombre de "passages" aussi.

**Algorithme 1.10:** BouclesPour(L,E,min,max)

**Données** : Une liste L ou bien un ensemble E, ou bien un intervalle
**Résultat** :

**pour tous les** *x de L* **faire**
<ListeInstructions1>;

**pour chaque** *y dans E* **faire**
<ListeInstructions2>;

**pour** *i=min* **à** *max* **faire**
<ListeInstructions2>;

**bouclesPour**

```
def bouclesPour(L,E,min,max):
    for x in L:
        print(x)
    for y in E:
        print(y)
    for i in range(min,max+1):
        print(i)
```

```
from bouclesPour import *

bouclesPour(['Merkel','Hollande'],{'Madame','Monsieur'},5,7)
```

```
Merkel
Hollande
Madame
Monsieur
5
6
7
```

### 1.3.5 La boucle "tant que"

**Objectif :** Éviter l'écriture de nombreuses suites d'instructions similaires, mais sans connaître a priori le nombre de "passages" dans l'itération. Un prédicat est utilisé pour "stopper" l'itération.

**Algorithme 1.11:** BouclesTantQue(P)

**Données** : Un predicat
**Résultat** :

**tant que** *P* **faire**
<ListeInstructions>;

**bouclesTantQue**

```
def bouclesTantQue(smin):
    s = 0;
    i = 0;
    while s<smin:
        s = s+i
        i = i+1
    return i
```

```
from bouclesTantQue import *

print("bouclesTantQue(567) = %s"\
          %bouclesTantQue(567))
```

```
bouclesTantQue(567) = 35
```

### 1.3.6 La boucle “répéter”

**Objectif :** Éviter l’écriture de nombreuses suites d’instructions similaires, mais sans connaître a priori le nombre de “passages” dans l’itération. Un prédicat est utilisé pour “stopper” l’itération.

**Algorithme 1.12:** BouclesRepeter(P)

**Données** : Un predicat
**Résultat** :

**répéter**
  <ListeInstructions>;
**jusqu’à** $P$;

**Équivalence entre "tant que" et "répéter"** Les structures de programme suivantes sont équivalentes.

**Algorithme 1.13:** BouclesTantQue(P)

**Données** : Un predicat
**Résultat** :

**tant que** $P$ **faire**
  <ListeInstructions>;

≡

**Algorithme 1.14:** BouclesRepeterEquivTantQue(P)

**Données** : Un prédicat
**Résultat** :

**si** $P$ **alors**
  **répéter**
    <ListeInstructions>;
  **jusqu’à** *not* $P$;

**Algorithme 1.15:** BouclesRepeter(P)

**Données** : Un predicat
**Résultat** :

**répéter**
  <ListeInstructions>;
**jusqu’à** $P$;

≡

**Algorithme 1.16:** BouclesTantQueEquivRepeter(P)

**Données** : Un prédicat
**Résultat** :

<ListeInstructions>;
**tant que** *not* $P$ **faire**
  <ListeInstructions>;

**Remarque :** pas de "répéter" en python

### 1.3.7 La composition d’algorithmes

**Objectif :** Éviter la duplication de code en utilisant un mécanisme similaire à la composition de fonctions en mathématiques. Une fonction soit “modifie les variables” du programme, soit “retourne un résultat” ; dans les deux cas, cette fonction peut être utilisée soit comme une instruction, soit comme une expression.

**composition**

```
def minimum2(X,Y):
    if X<Y:
        return X
    else:
        return Y

def minimum3(X,Y,Z):
    return minimum2(minimum2(X,Y),Z)
```

```
from composition import *

print("minimum3(13,5,8) = %s"\
        %minimum3(13,5,8))
```

```
minimum3(13,5,8) = 5
```

## 1.4 Complexité

### 1.4.1 Définitions et objectifs

Estimer le temps (ou l’espace) requis par l’exécution d’un algorithme lorsque les données en entrée deviennent très grandes.

**Définitions** Soit $T$ un algorithme et $n$ un entier qui représente la "taille" des données passées en paramètre à l'algorithme. On note $T(n)$ la fonction qui associe à la taille $n$ des données le temps (ou l'espace) requis par l'exécution de $T$.

- $T$ a une complexité dite *au pire des cas* en $\mathcal{O}(f(n))$ si $\exists n_0, \exists c$ tel que $\forall n > n_0, T(n) \leq c \times f(n)$
- $T$ a une complexité dite *au meilleur des cas* en $\Omega(f(n))$ si $\exists n_0, \exists c$ tel que $\forall n > n_0, T(n) \geq c \times f(n)$

**2014-2015 - Cours 3**

**Complexités classiques**

- Constante : $f(n) = 1$
- Logarithmique : $f(n) = ln(n)$ ou bien $f(n) = log_2(n)$ ou bien $f(n) = log_b(n)$
- Linéaire : $f(n) = n$
- Quadratique : $f(n) = n^2$
- Polynomiale : $f(n) = n^p$
- p-Exponentielle : $f(n) = p^n$
- Factorielle : $n!$

**Un ordre sur les complexités**

- Constante < Logarithmique < Linéaire < Quadratique < Polynomiale < p-Exponentielle < Factorielle.
- min(Linéaire, Quadratique) = Linéaire
- max(Linéaire, Quadratique) = Quadratique

**Propriété** $g(n) = \mathcal{O}(f(n)) \Leftrightarrow f(n) = \Omega(g(n))$

**Définition** Soit $T$ un algorithme et $n$ un entier qui représente la "taille" des données passées en paramètre à l'algorithme.

- $T$ a une complexité dite *exacte* en $\Theta(f(n))$ si $T$ est d'une part en $\mathcal{O}(f(n))$ ; d'autre part en $\Omega(f(n))$.

### 1.4.2 Complexité d'un algorithme

De façon simplifiée, un ordinateur dispose de trois types d'instructions s'exécutant chacune dans un temps constant.

- lecture (LOAD) en temps $t_{lec}$ : c'est le transfert de la mémoire vers un registre.
- calcul en temps $t_{cal}$ : c'est l'évaluation des expressions et notamment des prédicats.
- écriture (SAVE) en temps $t_{ecr}$ : c'est le transfert d'un registre vers la mémoire.

Le temps mis par l'exécution d'un algorithme est donc : $(\sharp_{lec} \times t_{lec}) + (\sharp_{cal} \times t_{cal}) + (\sharp_{ecr} \times t_{ecr})$

En général $t_{lec} \ll t_{cal} \ll t_{ecr}$, une approximation du temps est alors $(\sharp_{ecr} \times t_{ecr})$

Dans ce cours, on considérera $t_{lec} \ll \{t_{cal}, t_{ecr}\}$, une approximation du temps est alors $((\sharp_{cal} + \sharp_{ecr}) \times t_{ecr})$.

En complexité, les coefficients multiplicateurs ne sont pas retenus (ici $t_{ecr}$).

**L'analyse de complexité en temps consiste donc à évaluer $T_t(n) = (\sharp_{cal} + \sharp_{ecr})$.**

L'espace requis par l'exécution d'un algorithme est : $\sharp_{data} + \sharp_{prog} + \sharp_{aux}$

**L'analyse de complexité en espace consiste à évaluer $T_e(n) = (\sharp_{aux})$.**

**Un tableau T : type[n]** est une structure de données qui contient $n$ variables de même type, chacune accessible par un indice compris entre 0 et $n-1$. Par exemple T : entier[10] défini un tableau de 10 variables entières T[0], ..., T[9].

**Séquence**

**Algorithme 1.17:** ComplexiteSequenceInstructions()

**Complexité :** $\sharp_{cal,ecr} = 4$ *donc* $\Omega(1), \mathcal{O}(1)$ *donc* $\Theta(1)$

```
Lecture1;
Lecture2;
Calcul3;                                  /* 1 */
Ecriture4;                                /* 1 */
Lecture5;
Calcul6;                                  /* 1 */
Ecriture7;                                /* 1 */
```

**Algorithme 1.18:** ComplexiteSequence(D)

**Données** : D, des données de taille n
**Complexité :** $\Omega(max(f_1(n), f_2(n), f_3(n))), \mathcal{O}(max(g_1(n), g_2(n), g_3(n)))$

```
< S1 >;      /* Complexité de S1 : Ω(f1(n)), O(g1(n)) */
< S2 >;      /* Complexité de S2 : Ω(f2(n)), O(g2(n)) */
< S3 >;      /* Complexité de S3 : Ω(f3(n)), O(g3(n)) */
```

**Boucle "pour"**

**Algorithme 1.19:** ComplexiteBouclePourInstructions(T)

**Données** : T, un tableau de n entiers
**Complexité :** $\sharp_{cal,ecr} = 2n + 3$ *donc* $\Omega(n), \mathcal{O}(n)$ *donc* $\Theta(n)$

```
n ← longueur(T);                          /* 1 */
somme ← 0;                                /* 1 */
pour i=0 à n-1 faire                      /* 2n */
    somme ← somme+T[i];                   /* calcul + écriture : 2 */
retourner somme;                          /* 1 */
```

**Algorithme 1.20:** ComplexiteBouclePour(T)

**Données** : T, un tableau de n entiers
**Complexité :** $\Omega(n\ f(n)), \mathcal{O}(n\ g(n))$

```
n ← longueur(T);                          /* Θ(1) */
pour i=0 à n-1 faire                      /* Ω(n × f(n)), O(n × g(n)) */
    < S(T, i) > ;                         /* Complexité de S : Ω(f(n)), O(g(n)) */
```

**Boucle "tant que" et "répéter**

**Algorithme 1.21:** ComplexiteBoucleWhileInstructions(T,X)

**Données** : T, un tableau de n entiers, X un entier
**Complexité :** $\sharp_{cal,ecr} = [7..3n + 5]$ *donc* $\Omega(1), \mathcal{O}(n)$

```
n ← longueur(T);                          /* 1 */
i ← 0;                                    /* 1 */
trouve ← Faux;                            /* 1 */
tant que non trouve et i<n faire          /* [3..3n + 1] */
    trouve ← X=T[i];                      /* 1 */
    i ← i+1;                              /* 1 */
retourner trouve;                         /* 1 */
```

**Algorithme 1.22:** ComplexiteBoucleWhileSimple(T,X)

**Données** : T, un tableau de n entiers, X un entier
**Complexité :** $\Omega(f(n)), \mathcal{O}(n\ g(n))$

n ← longueur(T); /* $\Theta(1)$ */
i ← 0; /* $\Theta(1)$ */
trouve ← Faux; /* $\Theta(1)$ */
**tant que** *non trouve et i<n* **faire** /* $[1 + f(n)..1 + n \times (g(n) + 2)]$ */
  trouve ← $< S(T, i) >$ ; /* Complexité de S : $\Omega(f(n)), \mathcal{O}(g(n))$ */
  i ← i+1; /* $\Theta(1)$ */
**retourner** *trouve*; /* $\Theta(1)$ */

**Algorithme 1.23:** ComplexiteBoucleWhile(D)

**Données** : D, une donnée de taille n
**Complexité :** $\Omega(f(n)), \mathcal{O}(?)$

P ← Vrai; /* $\Theta(1)$ */
**tant que** *P* **faire**
  P ← $< S(D) >$ ; /* Complexité de S : $\Omega(f(n)), \mathcal{O}(g(n))$ */
**retourner** *P*; /* $\Theta(1)$ */

## 2014-2015 - Cours 4

**Boucles imbriquées**

Il suffit d'appliquer les formules précédentes en substituant <S> par une boucle. Cela donne naturellement :
- 2 boucles "pour" simples imbriquées : $\Theta(n\ m)$ ou bien $\Theta(n^2)$ si $n = m$. On parle de complexité "quadratique".
- k boucles "pour" simples imbriquées : $\Theta(\prod_i^k n_i)$ ou bien $\Theta(n^k)$.
- 2 boucles "tant que" simples imbriquées : $\Omega(1), \mathcal{O}(n\ m)$ ou bien $\Omega(1), \mathcal{O}(n^2)$ si $n = m$.
- ...

**Branchement**

**Algorithme 1.24:** ComplexiteBranchement(P)

**Données** : P un predicat, n taille des données
**Complexité :** $\Omega(min(f_1(n), f_2(n)), \mathcal{O}(max(g_1(n), g_2(n)))$

**si** *P* **alors**
  $< S1 >$ ; /* Complexité de S1 : $\Omega(f_1(n)), \mathcal{O}(g_1(n))$ */
**sinon**
  $< S2 >$ ; /* Complexité de S2 : $\Omega(f_2(n)), \mathcal{O}(g_2(n))$ */

**Composition** Il suffit de remarquer que les deux algorithmes suivants s'exécutent de la même façon.

**Algorithme 1.25:** Maximum3(X,Y,Z)

**Données** : 3 entiers X, Y et Z
**Résultat** : Le plus grand des trois

**retourner** Maximum2(Maximum2(X,Y)Z);

**Algorithme 1.26:** Maximum3Equiv(X,Y,Z)

**Données** : 3 entiers X, Y et Z
**Résultat** : Le plus grand des trois
**Complexité :** $\Theta(1)$

aux ← Maximum2(X,Y);
**retourner** Maximum2(aux,Z);

**Algorithme 1.27:** ComplexiteComposition(D)

**Données** : D, une donnée de taille n
**Complexité :** $\Omega(max(f2(n), f1(aux))), \mathcal{O}(max(g2(n), g1(aux)))$

aux ← algorithme2(X); /* aux de taille $\Omega(f2'(n)), \mathcal{O}(g2'(n))$ */
**retourner** algorithme1(aux);

**Algorithme 1.28:** ComplexiteComposition(D)

```
Données : D, une donnée de taille n
Complexité : Ω(max(f2(n), f1(f2'(n)))), O(max(g2(n), g1(g2'(n))))
retourner algorithme1(algorithme2(X));
```

**Algorithme** Il suffit d'appliquer pour chaque structure de contrôle les résultats de complexité, en commençant par les structures les plus imbriquées. Attention avec les boucles “tant que “ et “répéter” pour lesquelles la complexité au pire des cas n'est pas toujours connue.

### 1.4.3 Compromis entre temps et espace

**Algorithme 1.29:** ElementPlusFrequentV1(T)

```
Données : Un tableau T d'entiers naturels
Résultat : La valeur la plus fréquente, et son nombre d'occurrences
Complexité : Θ(n²) en temps, Θ(1) en espace
n ← longueur(T);
si n>0 alors                                   /* Θ(n²) */
    nbMax ← 0;
    pour i=0 à n-1 faire                       /* Θ(n²) */
        cpt ← 0;
        pour j=0 à n-1 faire                   /* Θ(n) */
            si T[i]=T[j] alors                 /* Θ(1) */
                cpt ← cpt+1;
        si cpt>nbMax alors                     /* Θ(1) */
            nbMax ← cpt;
            eltMax ← T[i];
    retourner eltMax,nbMax;
sinon                                          /* Θ(1) */
    Ecrire La liste est vide;
```

**Algorithme 1.30:** ElementPlusFrequentV2(T)

```
Données : Un tableau T d'entiers naturels
Résultat : La valeur la plus fréquente, et son nombre d'occurrences
Complexité : Θ(n²) en temps, Θ(1) en espace
n ← longueur(T);
si n>0 alors                                   /* Θ(n²) */
    nbMax ← 0;
    pour i=0 à n-1 faire                       /* Θ(n²) */
        cpt ← 1;
        pour j=i+1 à n-1 faire                 /* Θ(n − i) */
            si T[i]=T[j] alors                 /* Θ(1) */
                cpt ← cpt+1;
        si cpt>nbMax alors                     /* Θ(1) */
            nbMax ← cpt;
            eltMax ← T[i];
    retourner eltMax,nbMax;
sinon                                          /* Θ(1) */
    Ecrire La liste est vide;
```

**Algorithme 1.31:** ElementPlusFrequentV3(T,min,max)

**Données** : Un tableau T d'entiers naturels compris entre min et max
**Résultat** : La valeur la plus fréquente, et son nombre d'occurrences
**Complexité :** $\Theta(n + (max - min))$ *en temps,* $\Theta(max - min)$ *en espace*

```
n ← longueur(T);
si n>0 alors                                      /* Θ(n + (max − min)) */
    m ← max-min+1;
    Freq : entier[m];
    pour i=0 à m-1 faire                          /* Θ(m) */
        Freq[i] ← 0;
    pour i=0 à n-1 faire                          /* Θ(n) */
        Freq[T[i]-min] ← Freq[T[i]-min]+1;
    nbMax ← Freq[0];
    eltMax ← 0;
    pour i=1 à max faire                          /* Θ(m) */
        si Freq[i]>nbMax alors                    /* Θ(1) */
            nbMax ← Freq[i];
            eltMax ← i+min;
    retourner eltMax,nbMax;
sinon                                             /* Θ(1) */
    Ecrire La liste est vide;
```

**Algorithme 1.32:** ElementPlusFrequent(T)

**Données** : Un tableau T d'entiers naturels
**Résultat** : La valeur la plus fréquente, et son nombre d'occurrences
**Complexité :** $\Theta(min(n^2, n + (max - min)))$ *en temps,* $\Omega(1), \mathcal{O}(max - min)$ *en espace*

```
n ← longueur(T);
si n>0 alors                                      /* min(Θ(n + (max − min)), Θ(n²)) */
    max ← -1;
    min ← -1;
    pour i=0 à n-1 faire                          /* Θ(n) */
        si T[i]>max alors                         /* Θ(1) */
            max ← T[i];
        si T[i]<min alors                         /* Θ(1) */
            min ← T[i];
    si (max − min) > n² alors                     /* Θ(n²) */
        ElementPlusFrequentV2(T);
    sinon                                         /* Θ(n + (max − min)) */
        ElementPlusFrequentV3(T,min,max);
sinon                                             /* Θ(1) */
    Ecrire La liste est vide;
```

**2014-2015 - Cours 5**

### 1.4.4 Techniques d'évaluation de la complexité

Deux techniques principales :

- Appliquer pour chaque structure de contrôle les résultats de complexité, en commençant par les structures les plus imbriquées.
- Relations de récurrences :
  - $T(n+1) = T(n)$, complexité constante en 1.
  - $T(b \times n) = T(n) + c$, (avec $c > 0$), complexité logarithmique en $log_b(n) \equiv\ log_2(n) \equiv\ ln(n)$.
  - $T(n+1) = T(n) + c$, (avec $c > 0$), complexité linéaire en $n$.
  - $T(n+1) = T(n) + a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$, (avec $a_k > 0$), complexité polynomiale en $n^{k+1}$.
  - $T(n+1) = c \times T(n)$, (avec $c > 1$), complexité exponentielle $c^n$.

– $T(n+1) = (n+1) \times T(n)$, complexité factorielle $n!$.

**Attention** aux `Retourner` en *milieu* de boucle, qui rendent plus difficile le calcul de la complexité.

## 1.5 Pile d'exécution et passages de paramètres

### 1.5.1 Contenu et rôle d'une pile d'exécution

La pile contient la prochaine instruction à exécuter dès qu'un algorithme s'arrête. Cette instruction liste :

- Pour l'algorithme appelant :
  - le nom de l'algorithme.
  - la valeur du compteur ordinal pour indiquer le point de reprise.
  - la liste des variables locales et leur valeur pour pouvoir recharger le contexte.
  - Un espace mémoire pour que la fonction appelée puisse y mettre les valeurs retournées.
- Pour l'algorithme appelé :
  - le nom de l'algorithme.
  - la valeur du compteur ordinal correspondant à "début de programme".
  - la liste des valeurs des variables passées en paramètres.

**Exemple :** *Je fais un dessin au tableau de l'évolution d'une pile d'exécution pendant le déroulement d'un programme.*

**Algorithme 1.33:** EvolutionPileExecution()

**Données** :
**Résultat** :

v1 ← 133;
v2 ← 9;
v3 ← Algorithme(v1);
v4 ← 17;
v2 ← 4;
v4 ← Algorithme(v2);
v2 ← 11;
v3 ← Algorithme(v2);

### 1.5.2 Notion de paramètres

Terminologie trouvée dans la littérature informatique.

- types (IN, OUT, IN-OUT)
- procédures : passage par valeur (IN) / adresse ou références (IN-OUT)
- fonctions : entrée ou donnée (IN, IN-OUT) /sortie ou retour (OUT)

Pour un langage donné, ces termes sont précisés.

- python utilise un passage par valeur non modifiable.
  - paramètres (constantes) : passage par valeur (IN).
  - paramètres (variables de types simples) : passage par valeur (IN).
  - paramètres (variables conteneurs) : passage par valeur de l'adresse du conteneur (IN), ce qui permet de modifier le contenu (IN-OUT).
  - retour (constantes) : retour par valeur (OUT).
  - retour (variables de types simples) : retour par valeur (OUT).
  - retour (variables conteneurs) : retour de l'adresse du conteneur (OUT), ce qui permet de modifier le contenu (IN-OUT).

**Un exemple**

parametres

```
# Procedure --> aucun effet sur le contexte
def echangeV1(X,Y):
    aux = X
    X = Y
    Y = aux

# Fonction --> depend de son utilisation
```

```
def echangeV2(X,Y):
    return Y,X

# Procedure specialisee pour echanger 2 elements d'un tableau
def echangeV3(T,i,j):
    aux = T[i]
    T[i] = T[j]
    T[j] = aux


from parametres import *

T = [0,1,2,3,4,5]
print("T = %s" %T)
echangeV1(T[0],T[1])
T[2],T[3] = echangeV2(T[2],T[3])
echangeV3(T,4,5)
print("T = %s" %T)
T1 = [1,2]
T2 = [3,4]
print("T1 = %s, T2 = %s" %(T1,T2))
echangeV1(T1,T2)
print("T1 = %s, T2 = %s" %(T1,T2))
T1,T2 = echangeV2(T1,T2)
print("T1 = %s, T2 = %s" %(T1,T2))
```

```
T = [0, 1, 2, 3, 4, 5]
T = [0, 1, 3, 2, 5, 4]
T1 = [1, 2], T2 = [3, 4]
T1 = [1, 2], T2 = [3, 4]
T1 = [3, 4], T2 = [1, 2]
```

**2014-2015 - Cours 6**

## 1.6 Constructions algorithmiques élémentaires

### 1.6.1 Comptages, minimum et maximum

**Algorithme 1.34:** NbXverifiantP(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Le nombre de variables de T qui vérifient le prédicat P
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
nb ← 0;
pour i=0 à n-1 faire
    si P(T[i]) alors
        nb ← nb+1;
retourner nb;
```

**Algorithme 1.35:** MinX(T)

**Données** : Un tableau T d'objets comparables
**Résultat** : Le plus petit objet de T, et son indice dans T
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
si n=0 alors
    retourner "La liste est vide";
iMin ← 0;
minX ← T[iMin];
pour i=1 à n-1 faire
    si T[i]<minX alors
        iMin ← i;
        minX ← T[i];
retourner minX,iMin;
```

**Algorithme 1.36:** MaxX(T)

**Données** : Un tableau T d'objets comparables
**Résultat** : Le plus grand objet de T, et son indice dans T
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
si n=0 alors
    retourner "La liste est vide";
iMax ← 0;
maxX ← T[iMax];
pour i=1 à n-1 faire
    si T[i]>maxX alors
        iMax ← i;
        maxX ← T[i];
retourner maxX,iMax;
```

**Les programmes en annexe**

– Une version python

### 1.6.2 Quantifications existentielle et universelle

*Une première version basée sur le comptage, puis les points de vues algorithme et programme sont donnés simultanément.*

**Le point de vue algorithme**

**Algorithme 1.37:** ExisteXverifiantPNonOptimal(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi il existe une valeur qui vérifie le prédicat P
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
nb ← 0;
pour i=0 à n-1 faire
    si P(T[i]) alors
        nb ← nb+1;
retourner nb≥1;
```

**Algorithme 1.38:** ExisteXverifiantP(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi il existe une valeur qui vérifie le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

```
n ← longueur(T);
i ← 0;
auMoinsUnP ← Faux;
tant que non auMoinsUnP et i < n faire
    auMoinsUnP ← P(T[i]);
    i ← i+1;
retourner auMoinsUnP;
```

**Algorithme 1.39:** ExisteXverifiantNonP(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi il existe une valeur qui ne vérifie pas le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

n ← longueur(T);
i ← 0;
auMoinsUnNonP ← Faux;
**tant que** *non auMoinsUnNonP et i < n* **faire**
  auMoinsUnNonP ← non P(T[i]);
  i ← i+1;
**retourner** auMoinsUnNonP;

**Algorithme 1.40:** ToutXverifieP(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi toutes les valeurs de T vérifient le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

n ← longueur(T);
i ← 0;
tousP ← Vrai;
**tant que** *tousP et i < n* **faire**
  tousP ← P(T[i]);
  i ← i+1;
**retourner** tousP;

**Algorithme 1.41:** ToutXverifieNonP(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi toutes les valeurs de T ne vérifient pas le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

n ← longueur(T);
i ← 0;
tousNonP ← Vrai;
**tant que** *tousNonP et i < n* **faire**
  tousNonP ← non P(T[i]);
  i ← i+1;
**retourner** tousNonP;

**Les programmes en annexe**

- Une version python

**Le point de vue programmeur**

**Algorithme 1.42:** ExisteXverifiantPretour(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi il existe une valeur qui vérifie le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

n ← longueur(T);
**pour** *i=0* **à** *n-1* **faire**
  **si** *P(T[i])* **alors**
    **retourner** Vrai;
**retourner** Faux;

**Algorithme 1.43:** ExisteXverifiantNonPretour(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi il existe une valeur qui ne vérifie pas le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

```
n ← longueur(T);
pour i=0 à n-1 faire
    si non P(T[i]) alors
        retourner Vrai;
retourner Faux;
```

**Algorithme 1.44:** ToutXverifiePretour(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi toutes les valeurs de T vérifient le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

```
n ← longueur(T);
pour i=0 à n-1 faire
    si non P(T[i]) alors
        retourner Faux;
retourner Vrai;
```

**Algorithme 1.45:** ToutXverifieNonPretour(T)

**Données** : Un tableau T et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : Vrai ssi toutes les valeurs de T ne vérifient pas le prédicat P
**Complexité :** $\Omega(1), \mathcal{O}(n)$

```
n ← longueur(T);
pour i=0 à n-1 faire
    si P(T[i]) alors
        retourner Faux;
retourner Vrai;
```

**Les programmes en annexe**

– Une version python

### 1.6.3 Minimum, maximum avec prédicat

Ces algorithmes combinent l'algorithme de recherche d'un extremum, avec celui de la quantification existentielle. Attention, ils utilisent l'évaluation paresseuse dans le branchement.

**Algorithme 1.46:** MinXverifiantP(T)

**Données** : Un tableau T d'objets comparables et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : S'il existe, le plus petit objet de T vérifiant P et son indice
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
auMoinsUnP ← Faux;
pour i=0 à n-1 faire
    si P(T[i]) and ((non auMoinsUnP) or T[i]<minX) alors
        auMoinsUnP ← Vrai;
        iMin ← i;
        minX ← T[i];
si auMoinsUnP alors
    retourner minX,iMin;
sinon
    retourner "Le tableau ne contient pas d'élément verifiant P";
```

**Algorithme 1.47:** MaxXverifiantP(T)

**Données** : Un tableau T d'objets comparables et une fonction prédicat P de complexité $\Theta(1)$
**Résultat** : S'il existe, le plus grand objet de T vérifiant P et son indice
**Complexité :** $\Theta(n)$

```
n ← longueur(T);
auMoinsUnP ← Faux;
pour i=1 à n-1 faire
    si P(T[i]) and ((non auMoinsUnP) or T[i]>maxX) alors
        auMoinsUnP ← Vrai;
        iMax ← i;
        maxX ← T[i];
si auMoinsUnP alors
    retourner maxX,iMax;
sinon
    retourner "Le tableau ne contient pas d'élément verifiant P";
```

**Les programmes en annexe**

– Une version python

# Chapitre 2

# Algorithmes de tri et de recherche

**2014-2015 - Cours 7**

## 2.1 Le problème du tri

Trier des objets en fonction d'un ordre sur une clef. Cela revient à trier un tableau d'entiers.

**Critères** :

- Complexité en temps.
- Complexité en espace (*tri sur place*).
- tri stable (deux objets du tableau ayant des clefs identiques sont ordonnées dans le tableau résultat comme dans le tableau de départ).

## 2.2 L'approche "incrémentale"

Dans une telle approche, la résolution d'un problème va se faire par une itération de modifications élémentaires (ou incrémentales). Chacune diminuant la *distance* au résultat final. Trois étapes :

1. **Formaliser** avec des variables une situation intermédiaire dans laquelle le problème est partiellement traité.

2. **Décrire** une modification élémentaire.

3. **Trouver** d'une part les conditions initiales sur les variables ; d'autre part les valeurs *terminales* des variables qui permettront de déterminer le prédicat de *sortie* de l'itération.

## 2.3 Deux procédures utiles

### 2.3.1 L'échange de deux valeurs du tableau

**Algorithme 2.1:** Echange(T,i,j)

**Données** : Un tableau T et deux indices valides i et j
**Résultat** : Le tableau T avec T[i] et T[j] echangés
**Complexité :** $\Theta(1)$

aux $\leftarrow$ T[i];
T[i] $\leftarrow$ T[j];
T[j] $\leftarrow$ aux;

### 2.3.2 Les décalages dans un tableau

**Algorithme 2.2:** DecalageDroite(T,g,d)

**Données** : Un tableau T et deux indices valides g et d>g
**Résultat** : Le tableau T avec pour k dans ]g,d], T'[k]=T[k-1] et T'[g]=T[d]
**Complexité :** $\Theta(d-g)$

```
aux ← T[d];
pour k=d décroissant à g+1 faire
    T[k] ← T[k-1];
T[g] ← aux;
```

**Algorithme 2.3:** DecalageGauche(T,g,d)

**Données** : Un tableau T et deux indices valides g et d>g
**Résultat** : Le tableau T pour k dans [g,d[, T'[k]=T[k+1] et T'[d]=T[g]
**Complexité :** $\Theta(d-g)$

```
aux ← T[g];
pour k=g à d-1 faire
    T[k] ← T[k+1];
T[d] ← aux;
```

**Les programmes en annexe**

- Une version python

## 2.4 Les algorithmes de tri

### 2.4.1 Le tri par sélection

**Idée et exemple**

Trouver le plus petit pour l'échanger avec le premier, recommencer avec le second plus petit, et ainsi de suite.

*Je déroule un petit exemple*

**Approche incrémentale**

- Dans une situation *intermédiaire* :
  - Les variables du tableau avec un indice $< i$ sont triées et placées.
  - Les variables du tableau avec un indice $\geq i$ sont $\geq T[i-1]$.
- Trouver le plus petit avec un indice $\geq i$ pour l'échanger avec $i$ ; puis incrémenter $i$.
- Configuration initiale : $i = 0$, et terminale : $i = n - 2$

**L'algorithme**

**Algorithme 2.4:** TriSelection(T)

**Données** : Un tableau T d'entiers
**Résultat** : Le tableau T trie par ordre croissant des valeurs

```
pour i=0 à longueur(T)-2 faire                      /* Θ(n²) */
    iMin ← i;
    pour j=i+1 à longueur(T)-1 faire                /* Θ(n − i) */
        si T[j] < T [iMin] alors                    /* Θ(1) */
            iMin ← j;
    si i ≠ iMin alors                               /* Θ(1) */
        Echange(T,i,iMin);                          /* Θ(1) */
```

**Propriétés**

- Complexité en temps : $\Theta(n^2)$
- Complexité en espace : $\Theta(1)$

- Tri stable : Non **comme tous les tris qui échangent des variables dont les indices sont non consécutifs.**

**Les programmes en annexe**

- Une version python

### 2.4.2 Le tri à bulles

**Idée et exemple**

On permute tout couple de cases successives mal ordonnées. Un premier parcours *emmène* la plus grande valeur jusqu'à la dernière case du tableau, comme les grosses bulles qui remontent à la surface de l'eau plus vite que les petites. Un second parcours fait remonter la deuxième plus grande valeur, et il suffit d'arrêter la remontée à l'avant-dernière case du tableau ; et ainsi de suite.

*Je déroule un petit exemple*

**Approche incrémentale**

- Dans une situation *intermédiaire* :
  - Les variables du tableau avec un indice $> i$ sont triées et placées.
  - Les variables du tableau avec un indice $\leq i$ sont $\leq T[i+1]$.
- Inverser les couples $(i, i+1)$ "mal ordonnés" pour les indices $< i$ ; puis décrémenter $i$.
- Configuration initiale : $i = n-1$, et terminale : $i = 0$

**L'algorithme**

**Algorithme 2.5:** TriBulle(T)

**Données** : Un tableau T d'entiers
**Résultat** : Le tableau T trie par ordre croissant des valeurs

```
pour i=len(T)-1 à 1 decroissant faire              /* Θ(n²) */
    pour j=0 à i-1 faire                           /* Θ(i) */
        si T[j] >T [j+1] alors                     /* Θ(1) */
            Echange(T,j,j+1);                      /* Θ(1) */
```

**Propriétés**

- Complexité en temps : $\Theta(n^2)$
- Complexité en espace : $\Theta(1)$
- Tri stable : Oui **sauf si** $T[j] \geq T[j+1]$ **au lieu de** $T[j] > T[j+1]$

**Les programmes en annexe**

- Une version python

### 2.4.3 Le tri par insertion

**Idée et exemple**

C'est le tri des joueurs de cartes, qui font *glisser* une carte jusqu'à son emplacement dans la partie des cartes déjà *ordonnée*.

*Je déroule un petit exemple*

**2014-2015 - Cours 8**

**Approche incrémentale**

- Dans une situation *intermédiaire* :
  - Les variables du tableau avec un indice $< i$ sont triées.
  - Les variables du tableau avec un indice $\geq i$ n'ont jamais été *regardées*.
- Insérer la variable d'indice $i$ à la bonne position entre 0 et $i$ par un décalage vers la droite ; puis incrémenter $i$.
- Configuration initiale : $i = 1$, et terminale : $i = n-1$

#### L'algorithme

**Algorithme 2.6:** TriInsertion(T)

**Données** : Un tableau T d'entiers
**Résultat** : Le tableau T trié par ordre croissant des valeurs

```
pour i=1 à longueur(T)-1 faire                          /* Ω(n), O(n²) */
    j ← i-1;
    tant que j ≥ 0 and T[i] < T[j] faire                 /* Ω(1), O(i) */
        j ← j-1;
    si j ≠ i-1 alors                                     /* Ω(1), O(i − j) */
        DecalageDroite(T,j+1,i);                         /* Θ(i − j) */
```

#### Propriétés

- Complexité en temps : $\Omega(n), \mathcal{O}(n^2)$
- Complexité en espace : $\Theta(1)$
- Tri stable : Oui **sauf si** $T[i] \leq T[j]$ **au lieu de** $T[i] < T[j]$

#### Les programmes en annexe

- Une version python

### 2.4.4 Le tri par dénombrement

#### Idée et exemple

**Hypothèse :** Toutes les valeurs sont comprises entre 0 et $max$.
Un tableau des fréquences cumulées sert à copier dans un tableau résultat les valeurs triées.
*Je déroule un petit exemple*

#### Une approche par étapes successives

Plusieurs étapes (¬ approche incrémentale) :
- Comptage du nombre d'occurrences de chaque élément.
- Comptage cumulé des nombres d'occurrences des éléments inférieurs ou égaux.
- Positionnement dans un nouveau tableau des éléments en utilisant le comptage cumulé.

Algorithme peu évident à écrire, mais dont la correction est assez évidente.

#### L'algorithme

**Algorithme 2.7:** TriDenombrement(T,max)

**Données** : Un tableau T d'entiers compris entre 0 et max inclus
**Résultat** : Le tableau T trie par ordre croissant des valeurs

```
pour i=0 à max faire                                     /* Θ(max) */
    freq[i] ← 0;
pour i=0 à longueur(T)-1 faire                           /* Θ(n) */
    freq[T[i]] ← freq[T[i]] + 1;
freq[0] ← freq[0]-1;
pour i=1 à max faire                                     /* Θ(max) */
    freq[i] ← freq[i] + freq[i-1] ;
pour i=longueur(T)-1 à 0 descendant faire                /* Θ(n) */
    res[freq[T[i]]] ← T[i];
    freq[T[i]] ← freq[T[i]] - 1;
```

#### Propriétés

- Complexité en temps : $\Theta(n + max))$
- Complexité en espace : $\Theta(n + max))$
- Tri stable : Oui **sauf si dernier parcours croissant**

**Remarques**

- Compromis espace-temps.
- Hypothèse non restrictive, l'algorithme est facile à adapter avec trois paramètres `(T, min, max)` que l'on appelle avec `(T,min(T),max(T))`.
- La complexité en espace n'est pas dépendante de la *taille des objets* du tableau, mais seulement de leur nombre. `Freq` est un tableau de clefs (entiers, réels, ...), et `res` est un tableau d'adresses mémoire.

**Les programmes en annexe**

- Une version python

## 2.5 Le problème de la recherche d'un élément

Localiser un objet par sa clef dans un ensemble d'objets.
**Critères** :
- Complexité en temps.
- Complexité en espace.
- Le premier si présence de doublons.

### 2.5.1 Recherche dans un tableau

**Idée et exemple**

Il suffit de comparer un à un les éléments du tableau avec la valeur cherchée.

**Approche incrémentale**

- Dans une situation *intermédiaire* :
  - Les variables du tableau avec un indice $< i$ sont différentes de $X$.
  - Les variables du tableau avec un indice $\geq i$ n'ont pas été *regardées*.
- Comparer $X$ avec la valeur $T[i]$. Deux cas :
  - $T[m] = X$, la recherche est terminée.
  - $T[m] \neq X$ ; il faut incrémenter $i$.
- Configuration initiale : $i = 0$, et terminale : *trouve* ou bien $i = n$

**L'algorithme**

**Algorithme 2.8:** RechercheElement(T,X)

**Données** : Un tableau T d'entiers, et un entier X
**Résultat** : Le plus petit indice i tel que T[i]=X s'il existe

i ← 0;
trouve ← faux;
**tant que** *non trouve and i< longueur(T)* **faire** /* $\Omega(1), \mathcal{O}(n)$ */
  trouve ← T[i]=X;
  i ← i+1;
**si** *trouve* **alors** /* $\Theta(1)$ */
  **retourner** i-1;
**sinon**
  **retourner** "X n'est pas dans le tableau T";

**Propriétés**

- Complexité en temps : $\Omega(1), \mathcal{O}(n)$
- Complexité en espace : $\Theta(1)$
- Le premier si présence de doublons : Oui

**Les programmes en annexe**

- Une version python

### 2.5.2 Recherche dans un tableau trié

#### Idée et exemple

Analogies avec la recherche dans un dictionnaire, et avec le jeu *deviner un nombre entre 1 et 1000.*
*Je déroule un petit exemple*

#### Approche incrémentale

- Dans une situation *intermédiaire* :
  - Les variables du tableau avec un indice $< g$ sont inférieures à $X$.
  - Les variables du tableau avec un indice $> d$ sont supérieures à $X$.
  - Les variables du tableau avec un indice dans $[g, d]$ n'ont pas été *regardées* et sont comprises entre $T[g]$ et $T[d]$.
- Comparer $X$ avec la valeur d'indice médian $m$ entre $g$ et $d$. Trois cas :
  - $T[m] = X$, la recherche est terminée.
  - $T[m] < X$, l'espace de recherche est réduit à $[m+1, d]$ ; il faut affecter $m+1$ à $g$.
  - $T[m] > X$, l'espace de recherche est réduit à $[g, m-1]$ ; il faut affecter $m-1$ à $d$.
- Configuration initiale : $g = 0, d = n-1$, et terminale : *trouve* ou bien $g > d$.

#### L'algorithme

**Algorithme 2.9:** RechercheDichotomique(T,X)

**Données** : Un tableau T d'entiers trie, et un entier X
**Résultat** : Un indice i tel que T[i]=X s'il existe

```
g ← 0;
d ← longueur(T)-1;
trouve ← faux;
tant que not trouve and g ≤ d faire
    m ← (g+d) div 2;
    si T[m]=X alors
        trouve ← vrai;
    sinon si T[m]<X alors
        g ← m+1;
    sinon
        d ← m-1;
si trouve alors
    retourner m;
sinon
    retourner “X n'est pas dans le tableau T”;
```

#### Propriétés

- Complexité en temps :
  - Meilleur des cas (X = T[(n-1)div 2]) : une seule itération donc $\Omega(1)$
  - Pire des cas (X n'est pas dans T) : $T(2n) = T(n) + 4$ donc $\mathcal{O}(log_2(n))$
- Complexité en espace : $\Theta(1)$
- Le premier si présence de doublons : Non

**Rappels** sur les logarithmes.

$$\begin{aligned} log_b(n \times m) &= log_b(n) + log_b(m) \\ log_a(n) &= log_a(b) \times log_b(n) \end{aligned}$$

#### Les programmes en annexe

- Une version python

# Chapitre 3

# La Récursivité

**2014-2015 - Cours 9**

## 3.1 Récursivité, pile d'exécution et complexité

### 3.1.1 Définitions

Un algorithme, un programme, une fonction, une structure est dit récursif(ve) si il(elle) est défini(e) en faisant référence à lui(elle)-même.

**Remarque** : pas de structures récursives dans ce cours.

### 3.1.2 Exemple du calcul du PGCD de deux entiers

Le pgcd de deux entiers strictement positifs peut être défini par :

$$pgcd(a,b) \;=\; \begin{cases} 1 & \text{si } a = 1 \text{ ou bien } b = 1 \\ a & \text{si } a = b \\ pgcd(a-b,b) & \text{si } a > b \\ pgcd(a,b-a) & \text{si } a < b \end{cases}$$

#### Les algorithmes récursif et itératif

**Algorithme 3.1:** PgcdRecursif(a,b)

**Données** : Deux entiers a et b strictement positifs
**Résultat** : pgcd(a,b)

**si** *a=1 ou b=1* **alors**
  **retourner** 1;
**sinon si** *a=b* **alors**
  **retourner** a;
**sinon si** *a>b* **alors**
  **retourner** PgcdRecursif(a-b,b);
**sinon**
  **retourner** PgcdRecursif(a,b-a);

La figure 3.1 représente l'arbre des appels pour `PgcdRecursif(24,30)`.

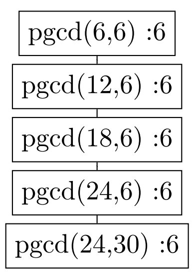



FIGURE 3.1 – Arbre des appels pour PgcdRecursif(24,30)

Le fait que le dernier appel calcule le résultat, et que tous les retours ne font que transmettre une valeur sans la modifier, permet d'écrire *facilement* une version itérative.

**Algorithme 3.2:** PgcdIteratif(a,b)

```
Données : Deux entiers a et b strictement positifs
Résultat : pgcd(a,b)
tant que Vrai faire
    si a=1 ou b=1 alors
        retourner 1;
    sinon si a=b alors
        retourner a;
    sinon si a>b alors
        a ← a-b;
    sinon
        b ← b-a;
```

#### Propriétés

- Version récursive
  - Complexité en temps : $\Omega(1), \mathcal{O}(max(a, b))$
  - Complexité en espace : $\Omega(1), \mathcal{O}(max(a, b))$
- Version itérative
  - Complexité en temps : $\Omega(1), \mathcal{O}(max(a, b))$
  - Complexité en espace : $\Theta(1)$

#### Les programmes

En annexe :
- Une version python

### 3.1.3 Pile d'exécution et complexité d'un algorithme récursif

- Complexité en temps : # appels → # changements de contexte.
- Complexité en espace : hauteur max de la pile d'exécution.

### 3.1.4 Récursivité terminale

- L'appel récursif est la dernière instruction appelée. → le résultat est un paramètre
- Transformation automatique algorithme récursif → algorithme itératif équivalent (pour la simulation), ayant même complexité en temps (sans les changements de contexte), mais une meilleure complexité espace.

## 3.2 Quelques exemples de fonctions récursives

### 3.2.1 La fonction factorielle

La fonction factorielle d'un entier naturel peut être définie par :

$$factorielle(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou bien } n = 1 \\ n \times factorielle(n-1) & \text{sinon} \end{cases}$$

#### Algorithmes itératif et récursif classiques

**Algorithme 3.3:** FactorielleIteratif(n)

```
Données : Un entier naturel n
Résultat : n!
res ← 1;
pour i=2 à n faire
    res ← n × res;
retourner res;
```

**Propriétés**

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(1)$

---

**Algorithme 3.4:** FactorielleRecursif(n)

---

**Données** : Un entier naturel n
**Résultat** : n !

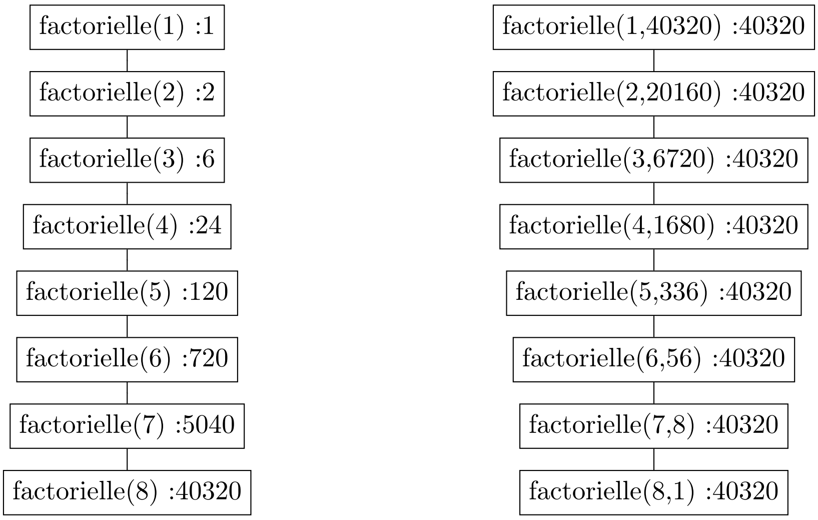
**si** $n<2$ **alors**
  **retourner** 1;
**sinon**
  **retourner** n × FactorielleRecursif(n-1);

---

**Propriétés**

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(n)$

La figure 3.2 représente l'arbre des appels pour `factorielle(8)`.




(a) Version récursive (b) Version récursive terminale

FIGURE 3.2 – Arbre des appels pour Factorielle(8)

**Algorithme récursif terminal**

---

**Algorithme 3.5:** FactorielleRecursifTerminal(n,u)

---

**Données** : Un entier naturel n, et un parametre accumulateur
**Résultat** : n ! = FactorielleRecursifTerminal(n,1)

**si** $n<2$ **alors**
  **retourner** u;
**sinon**
  **retourner** FactorielleRecursifTerminal(n-1,n*u);

---

**Algorithme 3.6:** Factorielle(n)

---

**Données** : Un entier naturel n
**Résultat** : n !

**retourner** FactorielleRecursifTerminal(n,1);

---

**Propriétés**

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(n)$

**Algorithme itératif *automatique***

**Algorithme 3.7:** FactorielleIteratifAutomatique(n)

**Données** : Un entier naturel n
**Résultat** : n !

```
u ← 1;
tant que True faire
    si n<2 alors
        retourner u;
    sinon
        u ← n*u;
        n ← n-1;
```

**Propriétés**

– Complexité en temps : $\Theta(n)$
– Complexité en espace : $\Theta(1)$

**Les programmes**

En annexe :

– Une version python

### 3.2.2 La suite de Fibonacci

La suite (ou fonction) de Fibonacci d'un entier naturel est définie par :

$$fibonacci(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & \text{sinon} \end{cases}$$

**Algorithme récursif classique**

**Algorithme 3.8:** FibonacciRecursif(n)

**Données** : Un entier naturel n
**Résultat** : fibonacci(n)

```
si n=0 alors
    retourner 0;
sinon si n=1 alors
    retourner 1;
sinon
    retourner FibonacciRecursif(n-1) + FibonacciRecursif(n-2);
```

La figure 3.3 représente l'arbre des appels pour `fibonacci(5)`. *J'ai dessiné l'arbre des appels pour fibonacci(5) pour introduire la terminologie sur les arbres (racine, branche, feuille, hauteur).*

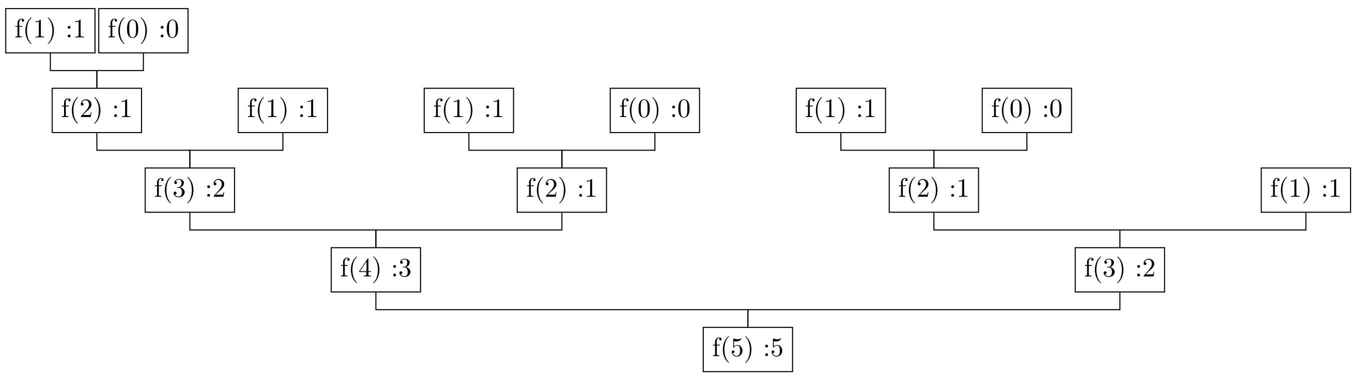



FIGURE 3.3 – Arbre des appels pour FibonacciRecursif(5)

## 2014-2015 - Cours 10

**Propriétés**

- Complexité en temps : $\Theta(Fib(n+1))$, Plus précisément :
  - $T(n+1) < 2T(n)$ donc $\mathcal{O}(2^n)$.
  - $T(n+1) > 2T(n-1)$ donc $\Omega(2^{n/2})$, soit $\Omega(\sqrt{2}^n)$.
  - En fait $\Theta(\varphi^n)$ avec $\varphi = \frac{1+\sqrt(5)}{2}$ le nombre d'or.
  - **non dit** Considérer la suite $g(n+1) = \frac{fib(n+1)}{fib(n)}$. Elle converge vers $X$ tel que $X = 1 + \frac{1}{X}$ donc vers le nombre d'or. Il est alors facile de déduire que $fib(n)$ converge vers $\varphi^n$.
- Complexité en espace : $\Theta(n)$

Pour éviter les calculs multiples, il suffit d'introduire deux paramètres, donc deux suites $(u_k, v_k)$, qui vont représenter au moment du $k^{eme}$ appel récursif (avec $k < n$), les valeurs de $(fib(k-1), fib(k))$. Les nouveaux paramètres se calculent aisément $(u_{k+1}, v_{k+1}) \leftarrow (v_k, u_k + v_k)$.

La figure 3.4 représente l'arbre des appels avec ces paramètres supplémentaires pour `fibonacci(5)`.

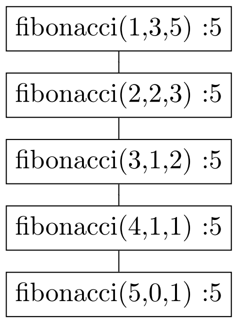



FIGURE 3.4 – Arbre des appels pour FibonacciRecursifTerminal(5,1,1)

**Algorithme récursif terminal**

**Algorithme 3.9:** FibonacciRecursifTerminal(n,u,v)

**Données** : Un entier naturel n, et deux accumulateurs u et v
**Résultat** : fibonacci(n) = FibonacciRecursifTerminal(n,0,1)

**si** *n=0* **alors**
  **retourner** u;
**sinon si** *n=1* **alors**
  **retourner** v;
**sinon**
  **retourner** FibonacciRecursifTerminal(n-1,v,u+v);

**Algorithme 3.10:** Fibonacci(n)

**Données** : Un entier naturel n
**Résultat** : fibonacci(n)

**retourner** FibonacciRecursifTerminal(n,0,1);

**Propriétés**

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(n)$

#### Algorithme itératif *automatique*

**Algorithme 3.11:** FibonacciIteratifAutomatique(n,u,v)

**Données** : Un entier naturel n
**Résultat** : fibonacci(n) = FibonacciIteratifAutomatique(n,0,1)

```
u ← 0;
v ← 1;
tant que vrai faire
    si n=0 alors
        retourner u;
    sinon si n=1 alors
        retourner v;
    sinon
        aux ← v;
        v ← u+v;
        u ← aux;
        n ← n-1;
```

#### Propriétés

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(1)$

#### Les programmes

En annexe :
- Une version python

### 3.2.3 La fonction d'Ackermann

La fonction d'Ackermann de deux entiers naturels est définie par :

$$ackermann(m,n) = \begin{cases} n+1 & \text{si } m=0 \\ ackermann(m-1,1) & \text{si } n=0 \\ ackermann(m-1, ackermann(m,n-1)) & \text{sinon} \end{cases}$$

#### Algorithme récursif classique

**Algorithme 3.12:** AckermannRecursif(m,n)

**Données** : Deux entiers m et n
**Résultat** : $2 \uparrow^{(m-2)} (n+3) - 3$

```
si m=0 alors
    retourner n+1;
sinon si n=0 alors
    retourner AckermannRecursif(m-1,1);
sinon
    retourner AckermannRecursif(m-1,AckermannRecursif(m,n-1));
```

La figure 3.5 représente l'arbre des appels pour `ack(2,2)=7`.

**Propriétés** Propriétés dites :
- $ack(n,n)$ croit beaucoup plus vite que la fonction $exp(n)$.
- Pour ackermann(4,1)=65533, RuntimeError : maximum recursion depth exceeded in comparison.
- Pas de fonction récursive terminale possible avec des simples ajoûts de paramètres.

**Propriétés non dites :**
- $ack(m,n) = 2 \uparrow^{(m-2)} (n+3) - 3$, et $ack(n,n) = 2 \uparrow^{(n-2)} (n+3) - 3$.
- Complexité en temps : #appels ?
- Complexité en espace : la branche la plus à gauche de l'avant dernier sous arbre le plus à droite à une longueur de $m + ackermann(m,n)$. La complexité est $\Theta(ackermann(m,n))$.
- Après analyse de la fonction, il est possible d'écrire une fonction plus efficace.

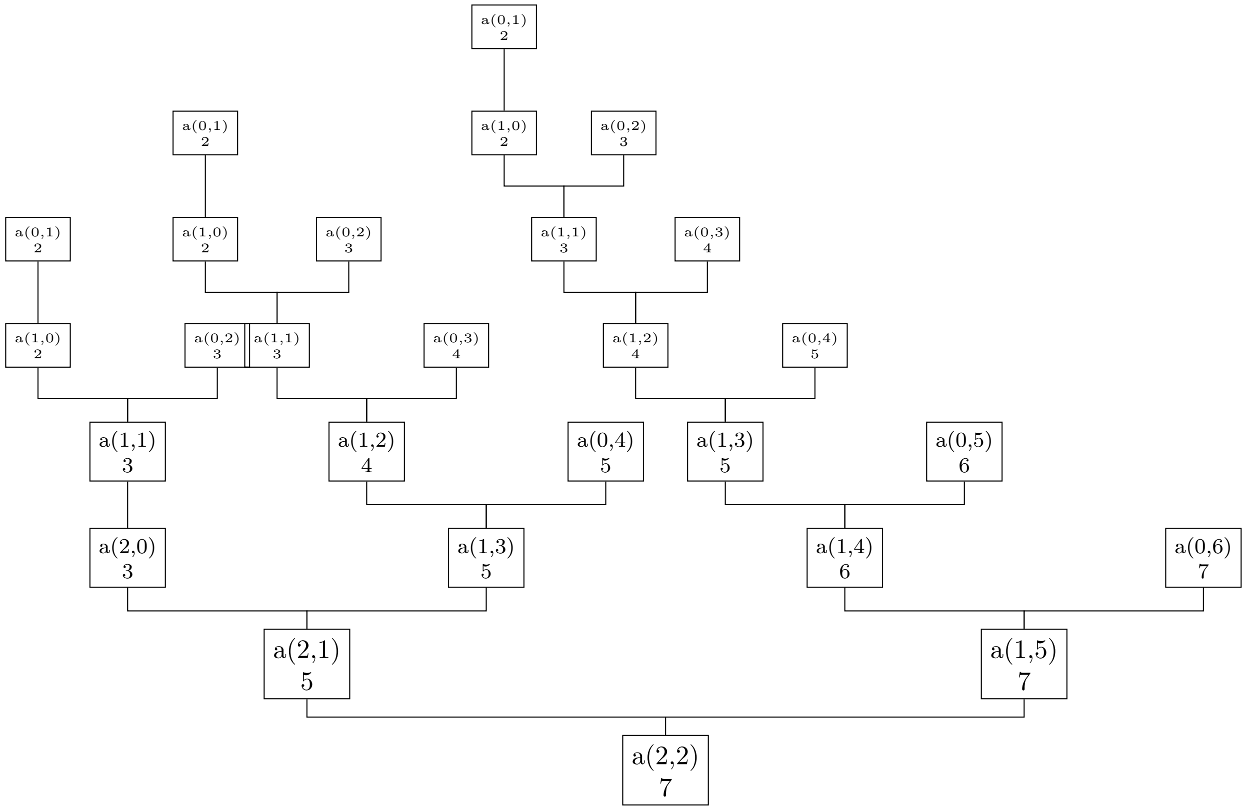



FIGURE 3.5 – Arbre des appels pour Ackermann(2,2))

#### Les programmes

En annexe :
- Une version python

### 3.2.4 La suite de Syracuse

La suite de Syracuse est définie pour un entier $n$ par :

$$suiteSyracuse(n) = \begin{cases} [\mathrm{n}] & \text{si } n \leq 1 \\ [\mathrm{n}] + suiteSyracuse(\mathrm{n}/2) & \text{si } n \text{ est pair} \\ [\mathrm{n}] + suiteSyracuse(3\mathrm{n}+1) & \text{si } n \text{ est impair} \end{cases}$$

La fonction associée à la suite de Syracuse peut être définie par :

$$syracuse(n) = \begin{cases} 1 & \text{si } n = 1 \\ syracuse(n/2) & \text{si } n \text{ est pair} \\ syracuse(3n+1) & \text{si } n \text{ est impair} \end{cases}$$

#### Algorithme récursif classique

**Algorithme 3.13:** SyracuseRecursif(n)

**Données** : Un entier n
**Résultat** : 1 si termine

**si** *n<=1* **alors**
  **retourner** 1;
**sinon si** *n mod 2 = 0* **alors**
  **retourner** SyracuseRecursif(n/2);
**sinon**
  **retourner** SyracuseRecursif(3n+1);

La figure 3.6 représente l'arbre des appels pour `syracuse(20)=1`.

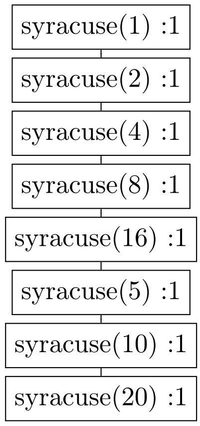



FIGURE 3.6 – Arbre des appels pour SyracuseRecursif(20))

**Propriétés**

- Complexité en temps (meilleur des cas) : $\Omega(log_2(n))$ obtenue pour les puissances de 2.
- Complexité en temps (pire des cas) : **inconnue** (problème dit ouvert).
- Complexité en espace (meilleur des cas) : $\Omega(log_2(n))$
- Complexité en espace (pire des cas) : **inconnue**
- Observation de la récursivité terminale.

**Algorithme itératif *automatique***

---

**Algorithme 3.14:** SyracuseIteratifAutomatique(n)

---

**Données** : Un entier n
**Résultat** : 1 si termine

**tant que** *Vrai* **faire**
  **si** $n<=1$ **alors**
    **retourner** 1;
  **sinon si** $n$ *mod* $2 = 0$ **alors**
    n ← n/2;
  **sinon**
    n ← 3*n+1;

---

**Propriétés**

- Complexité en temps (meilleur des cas) : $\Omega(log_2(n))$ obtenue pour les puissances de 2.
- Complexité en temps (pire des cas) : **inconnue** (problème dit ouvert).
- Complexité en espace : $\Theta(1)$

**Les programmes**

En annexe :
- Une version python

## 3.3 Conclusion

Ce qu'il faut savoir faire :
- Version récursive souvent naturelle et "facile" à prouver.
- Complexité temps difficile à calculer de manière *structurelle*, mais plus facile avec une formule de récurrence.
- Complexité espace de la pile quelquefois "critique". Les espaces *mémoire* et *pile d'exécution* s'additionnent car séparés.
- Il faut savoir écrire une version itérative d'une récursive terminale, même si le compilateur le fait automatiquement quelquefois **(facile)**. Cela garde la même complexité en temps, mais diminue la complexité en espace.

– Il faut essayer de transformer une récursive en récursive terminale en regardant les suites construites par les paramètres, et en ajoutant un (ou des) paramètre(s) accumulateur pour stocker le(s) résultat(s). **(difficile, et pas toujours possible sans utiliser d'autres artifices.)**

Les limites à connaître :

– Il existe des fonctions récursives impossibles à transformer en récursive terminale si on se limite à ajouter un nombre fini de paramètres.
– Il existe des fonction récursives terminales (donc itératives) dont on ne connaît pas la complexité. En fait on ne sait pas si le calcul termine toujours.

# Chapitre 4

# Algorithmes récursifs de tri

**2014-2015 - Cours 11**

## 4.1 L'approche "diviser pour régner"

Trois étapes :

1. **Diviser** le problème de taille $n$ en plusieurs sous-problèmes de tailles plus petites.
2. **Résoudre** les sous-problèmes (généralement de façons récursives).
3. **Combiner** les solutions aux sous-problèmes pour obtenir une solution au problème initial.

## 4.2 Le tri par fusion

### 4.2.1 L'idée

Supposons que l'on dispose d'un algorithme qui construit un tableau trié à partir de deux tableaux triés. Une solution appliquant *à la lettre* l'approche *diviser pour régner* est la suivante :

1. **Diviser** le tableau en deux tableaux de tailles identiques.
2. **Trier** récursivement les deux sous-tableaux.
3. **Combiner** les deux sous-tableaux triés en un seul tableau trié.

### 4.2.2 Les algorithmes "Fusionner" et "triFusion"

**Algorithme 4.1:** TriFusion(T,gauche,droite)

**Données** : Un tableau T d'entiers, et deux indices gauche et droite
**Résultat** : Le tableau T[gauche..droite] trie par ordre croissant des valeurs

```
si gauche<droite alors
    milieu ← (gauche+droite) div 2;
    TriFusion(T,gauche,milieu);
    TriFusion(T,milieu+1,droite);
    Fusionner(T,gauche,milieu,droite)
```

Il suffit d'appeler l'algorithme sur la totalité du tableau pour le trier.

**Algorithme 4.2:** TriFusion(T)

**Données** : Un tableau T d'entiers
**Résultat** : Le tableau T trie par ordre croissant des valeurs

```
TriFusion(T,0,longueur(T)-1);
```

Déroulement d'un exemple : Arbre des appels et fusion des résultats.

**Algorithme 4.3:** Fusionner(T,gauche,milieu,droite)

**Données** : Un tableau T, T[gauche..milieu] et T[milieu+1..droite] sont tries
**Résultat** : Le tableau T[gauche..droite] trie par ordre croissant des valeurs

```
R[0..droite-gauche];
i ← gauche;
j ← milieu+1;
k ← 0;
tant que i ≤ milieu and j ≤ droite faire
    si T[i] ≤ T[j] alors
        R[k] ← T[i];
        i ← i+1;
    sinon
        R[k] ← T[j];
        j ← j+1;
    k ← k+1;
tant que i ≤ milieu faire
    R[k] ← T[i];
    i ← i+1;
    k ← k+1;
tant que j ≤ droite faire
    R[k] ← T[j];
    j ← j+1;
    k ← k+1;
pour k=0 à droite-gauche faire
    T[gauche+k] ← R[k];
```

Déroulement d'un exemple.

### 4.2.3 Les programmes

En annexe :
- Une version python

### 4.2.4 Propriétés

Pour l'algorithme "Fusionner" :
- Complexité en temps : $\Theta(droite - gauche + 1)$ *Il faut regarder l'indice k.*
- Complexité en espace : $\Theta(droite - gauche + 1)$

Pour le tri récursif :
- Complexité en temps : $\Theta(nlog_2(n))$ *Par niveau de l'arbre des appels.*
- Complexité en espace : $\Theta(n) + \Theta(log_2(n))$ *non simplifié car le* $\Theta(n)$ *est en mémoire, et* $\Theta(log_2(n))$ *est pour la pile d'exécution.*
- Tri stable : Oui, **sauf si T[i]<T[j] au lieu de T[i]≤T[j] dans Fusionner**

**2014-2015 - Cours 12**

## 4.3 Le tri rapide

### 4.3.1 L'idée

C'est une variante de l'approche *diviser pour régner*, dans laquelle les sous-problèmes ne sont pas quelconques, afin que la combinaison des résultats des sous-problèmes devienne inutile.

Supposons que l'on sache partitionner le tableau en deux classes telles que tous les éléments de la première soient inférieurs ou égaux à ceux de la seconde.

1. **Diviser** le tableau en deux tableaux tels que $T1[i] \leq T2[j]$.
2. **Trier** récursivement les deux sous-tableaux.

### 4.3.2 Les algorithmes "partitionner" et "triRapide"

**Algorithme 4.4:** TriRapide(T,gauche,droite)

**Données** : Un tableau T d'entiers, et deux indices gauche et droite
**Résultat** : Le tableau T[gauche..droite] trie par ordre croissant des valeurs

```
si gauche<droite alors
    indicePivot ← partitionner(T,gauche,droite);
    TriRapide(T,gauche,indicePivot);
    TriRapide(T,indicePivot+1,droite);
```

Il suffit d'appeler l'algorithme sur la totalité du tableau pour le trier.

**Algorithme 4.5:** TriRapide(T)

**Données** : Un tableau T d'entiers
**Résultat** : Le tableau T trie par ordre croissant des valeurs

```
TriRapide(T,0,longueur(T)-1);
```

Déroulement d'un exemple : Arbre possible des appels.

**Algorithme 4.6:** Partitionner(T,gauche,droite)

**Données** : Un tableau T d'entiers, et deux indices gauche et droite
**Résultat** : Un indice indPivot tel que T[gauche..indPivot] ≤ T[indPivot+1..droite]

```
pivot ← T[gauche];
i ← gauche-1;
j ← droite+1;
tant que Vrai faire
    répéter
        i ← i+1;
    jusqu'à T[i] ≥ pivot;
    répéter
        j ← j-1;
    jusqu'à T[j] ≤ pivot;
    si i < j alors
        Echanger(T,i,j);
    sinon
        retourner j ;                          /* indPivot est égal à j */
```

Déroulement d'un exemple.

### 4.3.3 Les programmes

En annexe :
– Une version python

### 4.3.4 Propriétés

Pour l'algorithme "Partitionner" :
– Complexité en temps : $\Theta(droite - gauche + 1)$ *Il faut regarder la différence j-i.*
– Complexité en espace : $\Theta(1)$

Pour le tri récursif :
– Complexité en temps : $\Omega(nlog_2(n)), \mathcal{O}(n^2)$
– Complexité en espace : $\Theta(1)$
– Tri stable : Non, mais c'est possible. Une solution consiste à conserver pour chaque valeur du tableau sa place originale. Cette information sert à la fin pour ordonner correctement les valeurs égales. La complexité en espace est alors $\theta(n)$, mais ce n'est qu'un tableau d'indices.

Remarques :
– En moyenne, le plus efficace.
– Beaucoup de variantes pour le partitionnement (heuristique pour le choix du pivot, valeurs pivots correctement placées,...).

# Chapitre 5

# Introduction à la preuve d'algorithmes et de programmes

**2014-2015 - Non fait 1 (début)**

## 5.1 Objectifs et techniques

### 5.1.1 Objectifs

1. Garantir qu'un programme séquentiel :
   - Calcule le résultat attendu s'il termine.
   - Termine.
2. Trouver les hypothèses *minimales* (les moins contraignantes) pour assurer le point précédent.

Ces techniques sont utiles dans le cadre de la *certification* de programmes.

### 5.1.2 Techniques et outils

Diverses approches :
- Logique - système de preuve.
- Modèle - vérification.
- Interprétation abstraite.

Il existe des outils "plus ou moins" automatiques pour les différentes techniques.

## 5.2 Triplet de Hoare

**Définition 5.1** *Un triplet de Hoare est noté $\{P\}i\{Q\}$ ou $P$ (pré-condition) et $Q$ (post-condition) sont des assertions, et $i$ une instruction d'un algorithme ou d'un programme.*

**Définition 5.2** *Un triplet de Hoare $\{P\}i\{Q\}$ est dit valide si $M_{\uparrow i} \models P$ et $M_{i\uparrow} \models Q$, ou $M$ désigne la mémoire.*

Avec ces définitions, prouver qu'un algorithme $A$, qui prends en entrée des données vérifiant une hypothèse $H$ calcule bien le résultat $R$ consiste à prouver que le triplet de Hoare $\{H\}A\{R\}$ est valide.

## 5.3 Système de preuve : logique de Hoare

### 5.3.1 les 2 axiomes et les 5 règles classiques

L'axiome du *skip*.

$$A1 \quad : \quad \frac{True}{\{P\}\text{skip}\{P\}}$$

L'axiome de *l'affectation*.

$$A2 \quad : \quad \frac{True}{\{P_{[E/x]}\}x \leftarrow E\{P\}} \quad \text{Exemple} \quad : \quad \frac{True}{\{x+y=2\}x \leftarrow x+y+5\{x=7\}}$$

$$'' \quad : \quad \frac{True}{\{u \geq 2\}u \leftarrow u\ div\ 2\{u \geq 1\}}$$

La règle de la *composition séquentielle*.

$$R1 \quad : \quad \frac{\{P\}S1\{R\},\ \{R\}S2\{Q\}}{\{P\}S1;\ S2\{Q\}} \qquad \text{Exemple} \quad : \quad \frac{\{x{+}2y{=}2\}y{\leftarrow}2{*}y\{x{+}y{=}2\},\ \{x{+}y{=}2\}x{\leftarrow}x{+}y{+}5\{x{=}7\}}{\{x{+}2y{=}2\}y{\leftarrow}2{*}y;\ x{\leftarrow}x{+}y{+}5\{x{=}7\}}$$

$$'' \quad : \quad \frac{\{u{\geq}1\}u{\leftarrow}3{*}u{+}1\{u{\geq}2\},\ \{u{\geq}2\}u{\leftarrow}u\ div\ 2\{u{\geq}1\}}{\{u{\geq}1\}u{\leftarrow}3{*}u{+}1;\ u{\leftarrow}u\ div\ 2\{u{\geq}1\}}$$

La règle de la *conditionnelle*.

$$R2 \quad : \quad \frac{\{P\wedge B\}S1\{Q\},\ \{P\wedge\neg B\}S2\{Q\}}{\{P\}\ \text{si}\ B\ \text{alors}\ S1\ \text{sinon}\ S2\{Q\}}$$

$$\text{Exemple} \quad : \quad \frac{\{x{\geq}y\}m{\leftarrow}x\{m{=}max(x,y)\},\ \{x{<}y\}m{\leftarrow}y\{m{=}max(x,y)\}}{\{True\}\ \text{si}\ w{\geq}y\ \text{alors}\ m{\leftarrow}x\ \text{sinon}\ m{\leftarrow}y\{m{=}max(x,y)\}}$$

$$\text{``} \quad : \quad \frac{\{u{\geq}2\wedge u\ mod\ 2{=}0\}u{\leftarrow}u\ div\ 2\{u{\geq}1\},\ \{u{\geq}2\wedge u\ mod\ 2{\neq}0\}u{\leftarrow}3{*}u{+}1;\ u{\leftarrow}u\ div\ 2\{u{\geq}1\}}{\{u{\geq}2\}\ \text{si}\ u\ mod\ 2{=}0\ \text{alors}\ u{\leftarrow}u\ div\ 2\ \text{sinon}\ u{\leftarrow}3{*}u{+}1;\ u{\leftarrow}u\ div\ 2\{u{\geq}1\}}$$

La règle *faible* de *l'itération* ($P$ est dit l'invariant)

$$R3 \quad : \quad \frac{\{P\wedge B\}S\{P\}}{\{P\}\ \text{tant que}\ B\ \text{faire}\ S\{P\wedge\neg B\}}$$

$$\text{Exemple} \quad : \quad \frac{\{u{\geq}1\wedge u{>}1\}\ \text{si}\ u\ mod\ 2{=}0\ \text{alors}\ u{\leftarrow}u\ div\ 2\ \text{sinon}\ u{\leftarrow}3{*}u{+}1;\ u{\leftarrow}u\ div\ 2\{u{\geq}1\}}{\{u{\geq}1\}\ \text{tant que}\ u{>}1\ \text{faire si}\ u\ mod\ 2{=}0\ \text{alors}\ u{\leftarrow}u\ div\ 2\ \text{sinon}\ u{\leftarrow}3{*}u{+}1;\ u{\leftarrow}u\ div\ 2\{u{=}1\}}$$

La règle *forte* de *l'itération* ($P$ est dit l'invariant, et $v$ le variant)

$$R4 \quad : \quad \frac{\{P\wedge B\wedge v{=}i\wedge i{\geq}0\}S\{P\wedge v{<}i\},(v{<}0)\Rightarrow\neg B}{\{P\}\ \text{tant que}\ B\ \text{faire}\ S\{P\wedge\neg B\}}$$

Une règle de *logique* (affaiblissement des conséquences, renforcement des hypothèses).

$$R5 \quad : \quad \frac{P\Rightarrow P1,\{P1\}S\{Q1\},Q1\Rightarrow Q}{\{P\}S\{Q\}}$$

### 5.3.2 Limites

Les exemples choisis prouvent que si fa fonction de Syracuse termine, elle retourne 1.

Plus généralement, les règles précédentes sont suffisantes pour les programmes *structurés*, sans *pointeurs*. Ce système de preuve a été étendu pour d'autres constructions algorithmiques.

### 5.3.3 Utilisation

Avec ce système de preuve, la correction d'un algorithme sans itération est assez simple. Pour les itérations, il faut :

– Montrer qu'un invariant donné "manuellement" est conservé et donne le résultat attendu.
– Montrer qu'un variant donné "manuellement" donne la terminaison.

En pratique, le résultat attendu est connu, et il est donc plus *naturel* de partir du résultat et de *remonter* l'exécution du programme. C'est le sens des plus faibles pré-conditions de Dijkstra.

## 5.4 Les plus faibles pré-conditions de Dijkstra

**Définition 5.3** *Soit $i$ une instruction, soit $P$ un prédicat. $WP(i,Q)$ est définie par :*

– *$\{WP(i,Q)\}i\{Q\}$ est un triplet de Hoare valide,*
– *si $\{P\}i\{Q\}$ est un triplet valide alors $P \Rightarrow WP(i,Q)$*

### 5.4.1 WP pour les instructions sans itérations

L'axiome du *skip*.

$$A1 \quad : \quad \frac{\text{True}}{\{P\}\text{skip}\{P\}}$$

$$WP(\text{skip}, Q) = Q$$

L'axiome de *l'affectation*.

$$A2 \quad : \quad \frac{\text{True}}{\{P_{[E/x]}\}x{\leftarrow}E\{P\}}$$

$$WP(x \leftarrow E, Q) = Q_{[E/x]}$$

La règle de la *composition séquentielle.*

$$R1 \quad : \quad \frac{\{P\}S1\{R\},\{R\}S2\{Q\}}{\{P\}S1;S2\{Q\}}$$

$$WP(S1;S2,Q) \quad = \quad WP(S1,WP(S2,Q))$$

La règle de la *conditionnelle.*

$$R2 \quad : \quad \frac{\{P\wedge B\}S1\{Q\},\{P\wedge\neg B\}S2\{Q\}}{\{P\}\text{ si } B \text{ alors } S1 \text{ sinon } S2\{Q\}}$$

$$WP(\text{ si } B \text{ alors } S1 \text{ sinon } S2,Q) \quad = \quad (B \Rightarrow WP(S1,Q)) \wedge (\neg B \Rightarrow WP(S2,Q))$$

### 5.4.2 WP pour les itérations

Remarque : compliqué car définition récursive.
La règle *faible* de *l'itération* ($P$ est dit l'invariant)

$$R3 \quad : \quad \frac{\{P\wedge B\}S\{P\}}{\{P\}\text{ tant que } B \text{ faire } S\{P\wedge\neg B\}}$$

$$WP(\text{ tant que } B \text{ faire } S,Q) \quad = \quad (B \Rightarrow WP(S,WP(\text{ tant que } B \text{ faire } S,Q))) \wedge (\neg B \Rightarrow Q)$$

### 5.4.3 Utilisation

Remarque : le calcul des plus faibles pré-conditions peut être automatisé, sauf pour les itérations, ou il faut *donner* l'invariant et le variant. Certains outils actuels utilisent des *annotations* ajoutées dans les programmes pour prouver automatiquement des programmes. C'est la notion de programmes certifiés.

## 5.5 Exemple : le drapeau hollandais

### 5.5.1 L'algorithme

**Algorithme 5.1:** DrapeauDijkstra(T)

**Données** : Un tableau T de n entiers naturels
**Résultat** : Deux indices i et j tel que T[0..i-1]=0, T[i..j-1]=1 et T[j..n-1]≥2

```
i ← 0;
j ← 0;
k ← longueur(T)-1;
tant que j ≠ k+1 faire
    si T[j] = 0 alors
        Echanger(T,i,j);
        i ← i+1;
        j ← j+1;
    sinon
        si T[j] = 1 alors
            j ← j+1;
        sinon
            Echanger(T,j,k);
            k ← k-1;
retourner i,j;
```

### 5.5.2 Les programmes

En annexe :
- Une version python

### 5.5.3 L'invariant et le variant

L'invariant de boucle :

$$Inv = \forall l \in \mathbb{N}, \begin{cases} 0 \leq l < i & \Rightarrow \quad T[l] = 0 \\ i \leq l < j & \Rightarrow \quad T[l] = 1 \\ k+1 \leq l < n & \Rightarrow \quad T[l] \geq 2 \end{cases}$$

Le variant de la boucle :

$$var = k - j + 1$$

Notons également :

$$Inv11 = \forall l \in \mathbb{N}, \begin{cases} 0 \leq l < i+1 & \Rightarrow & T[l] = 0 \\ i+1 \leq l < j+1 & \Rightarrow & T[l] = 1 \\ k+1 \leq l < n & \Rightarrow & T[l] \geq 2 \end{cases}$$

$$Inv12 = \forall l \in \mathbb{N}, \begin{cases} 0 \leq l < i & \Rightarrow & T[l] = 0 \\ i \leq l < j+1 & \Rightarrow & T[l] = 1 \\ k+1 \leq l < n & \Rightarrow & T[l] \geq 2 \end{cases}$$

$$Inv21 = \forall l \in \mathbb{N}, \begin{cases} 0 \leq l < i & \Rightarrow & T[l] = 0 \\ i \leq l < j & \Rightarrow & T[l] = 1 \\ k \leq l < n & \Rightarrow & T[l] \geq 2 \end{cases}$$

### 5.5.4 La preuve de l'algorithme

**L'algorithme décoré avec des triplets de Hoare**

**Algorithme 5.2:** DrapeauDijkstraAnnote(T)

**Données** : Un tableau T de n entiers naturels
**Résultat** : Deux indices i et j tel que T[0..i-1]=0, T[i..j-1]=1 et T[j..n-1]≥2

```
assert : {Hyp : ∀l ∈ ℕ, (0 ≤ l < n − 1) ⇒ T[l] ≥ 0} ;          /* hypothèse initiale */
i ← 0;
assert : {P1 : Hyp ∧ (i = 0)} ;                                   /* axiome A2 */
j ← 0;
assert : {P2 : Hyp ∧ (i = 0 ∧ j = 0)} ;                           /* axiome A2 */
k ← longueur(T)-1;
assert : {P3 : Hyp ∧ (i = 0 ∧ j = 0 ∧ k = n − 1)} ;               /* axiome A2 */
assert : {Inv} car P3 ⇒ Inv ;                                     /* règle R5 */
tant que j ≠ k+1 faire
    assert : {Inv ∧ B1} avec B1 : j ≠ k + 1;
    si T[j] = 0 alors
        assert : {Inv ∧ B1 ∧ B2} avec B2 : T[j] = 0;
        Echanger(T,i,j);
        assert : {Inv11 ∧ B1} ;                                   /* 3 fois l'axiome A2 */
        i ← i+1;
        assert : {Inv12 ∧ B1} ;                                   /* axiome A2 */
        j ← j+1;
        assert : {Inv} ;                                          /* axiome A2 */
    sinon
        assert : {Inv ∧ B1 ∧ ¬B2};
        si T[j] = 1 alors
            assert : {Inv ∧ B1 ∧ ¬B2 ∧ B3} avec B3 : T[j] = 1;
            assert : {Inv ∧ B1 ∧ B3};
            j ← j+1;
            assert : {Inv} ;                                      /* axiome A2 */
        sinon
            assert : {Inv ∧ B1 ∧ ¬B2 ∧ ¬B3};
            assert : {Inv ∧ B1 ∧ B4} avec B4 : T[j] ≥ 2 ;         /* règle R5 */
            Echanger(T,j,k) ;                                     /* 3 fois l'axiome A2 */
            assert : {Inv21 ∧ B1};
            k ← k-1;
            assert : {Inv} ;                                      /* axiome A2 */
        assert : {Inv} ;                                          /* règle R2 */
    assert : {Inv} ;                                              /* règle R2 */
assert : {Inv ∧ ¬B1} ;                                            /* règle R4 */
retourner i,j;
```

La décoration finale montre que si le programme termine, il partitionne correctement le tableau. Pour la terminaison, il n'est pas difficile de montrer qu'à chaque itération le variant $k - j + 1$ diminue d'une unité.

**Les programmes décorés avec des triplets de Hoare**

En annexe :

– Une version python annotée

## 2014-2015 - Non fait 1 (fin)

# Chapitre 6

# Morceaux choisis

**2014-2015 - Cours 13**

## 6.1 Plus longue sous-séquence commune et programmation dynamique

### 6.1.1 Le problème PLSC (Plus longue sous-séquence commune)

Un mot est une suite de lettres $w = w_0 \dots w_n$. Les mots $u$ obtenus en retirant un nombre quelconque (entre 0 et $len(w)$) de lettres forment les sous-séquences (sous-mots) du mot $w$. Exemple : si $w = abacb$, alors

$$\begin{aligned} sousMots(w) &= \{\epsilon, a, b, c, ab, aa, ac, ba, bc, bb, cb, aba, abc, abb, aac, aab, acb, bac, bab, bcb, \\ &\quad abac, abab, abcb, aacb, bacb, abacb\} \\ card(sousMots(w)) &\leq 2^{len(w)} \end{aligned}$$

Soit $u$ et $v$ deux mots. Il est possible de calculer $sousMots(u) \cap sousMots(v)$, donc de calculer la longueur de la plus longue sous-séquence commune à ces deux mots.

Le problème PLSC consiste à trouver **une** sous-séquence commune de longueur maximale.

**Exemples :**

– Une PLSC de 'aabbccdd' et de 'abbbcccdeeeee' est le mot 'abbccd'.
– Les PLSC de 'abcabcabc' et de 'aaabbbccc' sont les mots 'aaabc', abbbc' et 'abccc'.

**Remarque :** Une solution par énumération coute $max(m, n) \times 2^{min(m,n)}$.

– Calculer $E = sousMots(min(w1, w2))$
– Pour chaque $u \in E$, tester si $u$ est un sous-mot de $max(w1, w2)$.

### 6.1.2 Un algorithme récursif

**Notations :** Soit $w = w_0 \dots w_n$ un mot. On note $w[i] = w_i$ la $(i+1)^{eme}$ lettre de $w$, et $w^i = w_0 \dots w_{i-1}$ le mot composé des $i$ premières lettres du mot $w$. Par convention $w^0$ désigne le mot vide $\epsilon$.

**Propriété :** Soient $u = u_0 \dots u_m$ et $v = v_0 \dots v_n$ deux mots. Soit $w = w^{k+1}$ une PLSC de $u^{m+1}$ et $v^{n+1}$, alors :

$$\begin{cases} \text{si } u[m] = v[n] & \text{alors } w[k] = u[m] \text{ et } w^k \text{ est une PLSC de } u^m \text{et } v^n \\ \text{si } u[m] \neq v[n] & \text{alors } w[k] \neq u[m] \Rightarrow (w^{k+1} \text{ est une PLSC de } u^m \text{ et } v^{n+1}) \\ \text{si } u[m] \neq v[n] & \text{alors } w[k] \neq v[n] \Rightarrow (w^{k+1} \text{ est une PLSC de } u^{m+1} \text{ et } v^n) \end{cases}$$

**Propriété :** De la propriété précédente, découle la suivante :

$$plsc(u^i, v^j) = \begin{cases} \epsilon & \text{si } i = 0 \text{ ou } j = 0 \\ plsc(u^{i-1}, v^{j-1}).u[i] & \text{si } i > 0, j > 0 \text{ et } u[i] = v[j] \\ max(plsc(u^{i-1}, v^j), plsc(u^i, v^{j-1})) & \text{si } i > 0, j > 0 \text{ et } u[i] \neq v[j] \end{cases}$$

Des propriétés précédentes, découle le programme récursif suivant :

**Algorithme 6.1:** PlscRecursif(u,v)

**Données** : deux mots u et v vus comme des tableaux de caracteres
**Résultat** : un mot qui est plsc de u et de v

```
si longueur(u)=0 ou longueur(v)=0 alors
    retourner ϵ;
sinon si u[0]=v[0] alors
    retourner Concatener(u[0], PlscRecursif(u[1 ... longueur(u)-1], v[1 ... longueur(v)-1]));
sinon
    p1 ← PlscRecursif(u[1 ... longueur(u)-1], v);
    p2 ← PlscRecursif(u, v[1 ... longueur(v)-1]);
    si longueur(p1) ≥ longueur(p2) alors
        retourner p1;
    sinon
        retourner p2;
```

Présentation d'un arbre d'appels récursifs pour montrer les répétitions de sous-problèmes ('abcd', et 'cda').

#### Les programmes

En annexe :
- Une version python

#### Propriétés

- Complexité en temps : $\Omega(min(m,n)), \mathcal{O}(2^{min(m,n)})$ *(u prefixe de v, aucune lettre commune)*.
- Complexité en espace : $\Omega(min(m,n)), \mathcal{O}(m+n)$ *(u prefixe de v, aucune lettre commune)*.

### 6.1.3 La programmation dite "dynamique"

Ce programme récursif génère plusieurs instances de sous-problèmes identiques. Malheureusement, il n'est pas possible comme dans le cas de la suite de Fibonacci, d'ajouter un paramètre *accumulateur* à la fonction pour la rendre récursive terminale. En effet pour ce problème, le nombre de paramètres nécéssaires dépend de la longueur des mots.

Lorsque cette situation se produit, il est en général efficace d'utiliser la programmation dite "dynamique". Cette technique consiste à utiliser des tableaux pour stocker les résultats des sous-problèmes afin d'éviter de les recalculer.

### 6.1.4 Un algorithme itératif "dynamique"

L'application de l'approche "dynamique" conduit à l'algorithme suivant :

**Algorithme 6.2:** PlscDynamique(u,v)

**Données** : deux mots u et v vus comme des tableaux de caracteres
**Résultat** : un mot qui est plsc de u et de v

```
res : mots[1+longueur(u)][1+longueur(v)];
pour i=0 à longueur(u) faire
    res[i][0] ← ϵ;
pour j=0 à longueur(v) faire
    res[0][j] ← ϵ;
pour i=1 à longueur(u) faire
    pour j=1 à longueur(v) faire
        si u[i-1]=v[j-1] alors
            res[i][j] ← concatener(res[i-1][j-1],u[i-1]);
        sinon si longueur(res[i][j-1]) ≥ longueur(res[i-1][j]) alors
            res[i][j] ← res[i][j-1];
        sinon
            res[i][j] ← res[i-1][j];
retourner res[longueur(u)][longueur(v)];
```

#### Les programmes

En annexe :
- Une version python

#### Propriétés

- Complexité en temps : $\Theta(m \times n)$
- Complexité en espace : $\Theta(min(m,n) \times m \times n)$ *(*$min(m,n)$ *pour stocker un mot).*

### 6.1.5 Un algorithme itératif "dynamique" amélioré

En fait, l'information de la longueur des "plsc" est suffisante pour pouvoir construire une "plsc", cela permet de remplacer le tableau de mots par un tableau d'entiers. Cela conduit aux deux algorithmes suivants :

**Algorithme 6.3:** PlscCodage(u,v)

**Données** : deux mots u et v vus comme des tableaux de caracteres
**Résultat** : un tableau des longueurs des plsc des prefixes de u et v

```
code : entier[1+longueur(u)][1+longueur(v)];
pour i=0 à longueur(u) faire
    code[i][0] ← 0;
pour j=0 à longueur(v) faire
    code[0][j] ← 0;
pour i=1 à longueur(u) faire
    pour j=1 à longueur(v) faire
        si u[i-1]=v[j-1] alors
            code[i][j] ← 1+code[i-1][j-1];
        sinon
            code[i][j] ← max(code[i][j-1],code[i-1][j]);
retourner code;
```

**Algorithme 6.4:** PlscDecodage(u,v,code)

**Données** : deux mots u et v et un tableau d'entiers
**Résultat** : une des plsc de u et v

```
plsc ← ε;
i ← longueur(u);
j ← longueur(v);
tant que i>0 et j>0 et code[i][j]>0 faire
    si u[i-1]=v[j-1] alors
        plsc ← concatener(u[i-1], plsc);
        i ← i-1;
        j ← j-1;
    sinon si code[i][j-1] ≥ code[i-1][j] alors
        j ← j-1;
    sinon
        i ← i-1;
retourner plsc;
```

#### Les programmes

En annexe :
- Une version python

#### Propriétés

- Complexité en temps : $\Theta(m \times n)$
- Complexité en espace : $\Theta(m \times n)$

**2013-2014 - Non fait 2 (début)**

## 6.2 L'algorithme de Bressenham de tracé d'un segment de droite

Cette section est très fortement inspirée des pages Wikipedia sur le sujet.

### 6.2.1 Le problème

Tracer un segment de droite :

1. dans un plan discret,
2. défini par deux points à coordonnées entières,
3. le plus efficacement possible,
4. sans erreur,
5. en préservant une *connexité* des points.

Applications :

- Tracer un segment sur un écran.
- Tracer une courbe avec des imprimantes matricielles, ce n'est plus vraiment d'actualité, mais l'algorithme a été découvert dans ce cadre là.

### 6.2.2 Comment aboutir à l'algorithme

### 6.2.3 Remarques

Propriétés des algorithmes précédents

1. L'algorithme de Bresenham est le plus efficace, car il n'effectue que des opérations sur les entiers.
2. Il est possible de trouver des valeurs pour lesquelles toutes les solutions qui utilisent des calculs sur les réels ne donnent pas le bon tracé. Cela est du aux erreurs d'arrondi sur les calculs, et sur les cumuls d'erreurs.
3. L'algorithme de Bresenham est toujours juste.
4. L'algorithme de Bresenham présente un *cycle* que l'on peut mémoriser pour le répéter.

En fait, la méthode peut s'appliquer aux courbes dont les dérivées permettent de calculer les orientations de segments élémentaires avec de simples opérations entières. Il existe donc des algorithmes dits de 'Bresenham' pour :

- Les courbes coniques (cercle, ellipse, arc, parabole, hyperbole).
- Les courbes de Bézier grâce aux propriétés de leur fonction polynomiale de définition.

## 6.3 Algorithmes de recherche d'un motif P dans un texte T

### 6.3.1 Le problème

### 6.3.2 L'algorithme naïf

2 boucles imbriquées en 0(T.l x P.l) et Omega(T.l)
En annexe :

- La version python

### 6.3.3 L'algorithme utilisant un automate de recherche de motif

- l'algorithme de recherche : Theta(T.l) - l'algorithme de construction de l'automate : complexite non donnée
En annexe :

- La version python

### 6.3.4 L'algorithme de Knuth, Morris et Pratt

En annexe :

- La version python

**2013-2014 - Non fait 2 (fin)**

# Chapitre 7

# Annales DST

**2014-2015 - Non fait 3 (début)**

## 7.1 Devoir Surveillé Terminal de 2011-2012

Corrigé commenté.

## 7.2 Devoir Surveillé Terminal de 2012-2013

Corrigé commenté.

**2014-2015 - Non fait 3 (fin)**

**2014-2015 - Cours 14**

## 7.3 Devoir Surveillé Terminal de 2013-2014

Corrigé commenté.

# Annexe A

# Sources python des algorithmes

## A.1 Avertissements

Les programmes qui suivent servent à illustrer un cours d'algorithmique et de programmation. Ils sont écrits avec une vision *programmation impérative* et n'utilisent donc pas du tout les aspects *objets* du langage Python.

## A.2 Rappels UE J1MI1003 du semestre 1

**predicats**

```
def pair(n):
    return n%2==0
```

```
from predicats import *

print("pair(5) = %s" %pair(5))
print("pair(10) = %s" %pair(10))
```

```
pair(5) = False
pair(10) = True
```

**evaluationParesseuse**

```
def correctOr():
    P = True
    return P or Q

def correctAnd():
    P = False
    return P and Q

def bugOr():
    P = False
    return P or Q

def bugAnd():
    P = True
    return P and Q
```

```
import sys
from evaluationParesseuse import *

print("correctAnd = %s"\
        %correctAnd())
print("correctOr = %s"\
        %correctOr())
try:
    print("bugAnd = %s" %bugAnd())
except:
    print("Unexpected error:")
    print("\t", sys.exc_info()[0])
    print("\t", sys.exc_info()[1])
try:
    print("bugOr = %s" %bugOr())
except:
    print("Unexpected error:")
    print("\t", sys.exc_info()[0])
    print("\t", sys.exc_info()[1])
```

```
correctAnd = False
correctOr = True
Unexpected error:
        <class 'NameError'>
        global name 'Q' is not defined
Unexpected error:
        <class 'NameError'>
        global name 'Q' is not defined
```

## branchements

```
def civilite(masculin,nom,prenom):
    if masculin:
        print("Bonjour Monsieur %s %s"\
                %(prenom,nom))
    else:
        print("Bonjour Madame %s %s"\
                %(prenom,nom))

def minimum2V1(X,Y):
    if X<=Y:
        min = X
    else:
        min = Y
    return min

def minimum2V2(X,Y):
    if X<=Y:
        return X
    else:
        return Y

def minimum2V3(X,Y):
    if X<=Y:
        return X
    return Y

def minimum3V1(X,Y,Z):
    if X<=Y:
        if X<=Z:
            return X
        else:
            return Z
    else:
        if Y<=Z:
            return Y
        else:
            return Z

def minimum3V2(X,Y,Z):
    if X<=Y and X<=Z:
        return X
    elif Y<=Z:
```

```
        return Y
    else:
        return Z
```

```
from branchements import *

civilite(True,'Hollande','Francois')
civilite(False,'Merkel','Angela')
print("minimum2V1(5,8) = %s"\
          %minimum2V1(5,8))
print("minimum2V2(13,8) = %s"\
          %minimum2V2(13,8))
print("minimum2V3(13,18) = %s"\
          %minimum2V3(13,18))
print("minimum3V1(5,13,8) = %s"\
          %minimum3V1(5,13,8))
print("minimum3V2(13,8,5) = %s"\
          %minimum3V2(13,8,5))
```

```
Bonjour Monsieur Francois Hollande
Bonjour Madame Angela Merkel
minimum2V1(5,8) = 5
minimum2V2(13,8) = 8
minimum2V3(13,18) = 13
minimum3V1(5,13,8) = 5
minimum3V2(13,8,5) = 5
```

## bouclesPour

```
def bouclesPour(L,E,min,max):
    for x in L:
        print(x)
    for y in E:
        print(y)
    for i in range(min,max+1):
        print(i)
```

```
from bouclesPour import *

bouclesPour(['Merkel','Hollande'],{'Madame','Monsieur'},5,7)
```

```
Merkel
Hollande
Madame
Monsieur
5
6
7
```

## bouclesTantQue

```
def bouclesTantQue(smin):
    s = 0;
    i = 0;
    while s<smin:
        s = s+i
        i = i+1
    return i
```

```
from bouclesTantQue import *

print("bouclesTantQue(567) = %s"\
          %bouclesTantQue(567))
```

```
bouclesTantQue(567) = 35
```

## composition

```
def minimum2(X,Y):
    if X<Y:
        return X
    else:
        return Y

def minimum3(X,Y,Z):
    return minimum2(minimum2(X,Y),Z)
```

```
from composition import *

print("minimum3(13,5,8) = %s"\
        %minimum3(13,5,8))
```

```
minimum3(13,5,8) = 5
```

## parametres

```
# Procedure --> aucun effet sur le contexte
def echangeV1(X,Y):
    aux = X
    X = Y
    Y = aux

# Fonction --> depend de son utilisation
def echangeV2(X,Y):
    return Y,X

# Procedure specialisee pour echanger 2 elements d'un tableau
def echangeV3(T,i,j):
    aux = T[i]
    T[i] = T[j]
    T[j] = aux
```

```
from parametres import *

T = [0,1,2,3,4,5]
print("T = %s" %T)
echangeV1(T[0],T[1])
T[2],T[3] = echangeV2(T[2],T[3])
echangeV3(T,4,5)
print("T = %s" %T)
T1 = [1,2]
T2 = [3,4]
print("T1 = %s, T2 = %s" %(T1,T2))
echangeV1(T1,T2)
print("T1 = %s, T2 = %s" %(T1,T2))
T1,T2 = echangeV2(T1,T2)
print("T1 = %s, T2 = %s" %(T1,T2))
```

```
T = [0, 1, 2, 3, 4, 5]
T = [0, 1, 3, 2, 5, 4]
T1 = [1, 2], T2 = [3, 4]
T1 = [1, 2], T2 = [3, 4]
T1 = [3, 4], T2 = [1, 2]
```

## comptages

```
import random

def P(X):
    return random.randrange(2)==0

def nbXverifiantP(T):
    nb = 0
    for i in range(len(T)):
        if P(T[i]):
            nb = nb+1
    return nb

def minX(T):
    if len(T)==0:
        return None
    iMin = 0
    minX = T[iMin]
    for i in range(1,len(T)):
        if T[i]<minX:
            iMin = i
            minX = T[i]
    return [minX,iMin]
```

```
def maxX(T):
    if len(T)==0:
        return None
    iMax = 0
    maxX = T[iMax]
    for i in range(1,len(T)):
        if T[i]>maxX:
            iMax = i
            maxX = T[i]
    return [maxX,iMax]

from comptages import *

T = [1,2,3,4]
print("T␣=%s"%T)
print("nbXverifiantP(T)␣=␣%s"\
        %nbXverifiantP(T))
print("minX(T)␣=␣%s"\
        %minX(T))
print("maxX(T)␣=␣%s"\
        %maxX(T))
```

```
T =[1, 2, 3, 4]
nbXverifiantP(T) = 2
minX(T) = [1, 0]
maxX(T) = [4, 3]
```

## quantifications

```
import random

def P(X):
    return random.randrange(2)==0

def existeXverifiantP(L):
    n = len(L)
    i = 0
    auMoinsUnP = False
    while not auMoinsUnP and i<n:
        auMoinsUnP = P(L[i])
        i = i+1
    return auMoinsUnP

def existeXverifiantNonP(L):
    n = len(L)
    i = 0
    auMoinsUnNonP = False
    while not auMoinsUnNonP and i<n:
        auMoinsUnNonP != P(L[i])
        i = i+1
    return auMoinsUnNonP

def toutXverifieP(L):
    n = len(L)
    i = 0
    tousP = True
    while tousP and i<n:
        tousP = P(L[i])
        i = i+1
    return tousP

def toutXverifieNonP(L):
    n = len(L)
    i = 0
    tousNonP = True
    while tousNonP and i<n:
        tousNonP != P(L[i])
        i = i+1
    return tousNonP
```

```
from quantifications import *

L = [1,2,3,4]
print("existeXverifiantP(L) =\n\t%s"\
        %existeXverifiantP(L))
print("existeXverifiantNonP(L) =\n\t%s"\
        %existeXverifiantNonP(L))
print("toutXverifieP(L) =\n\t%s"\
        %toutXverifieP(L))
print("toutXverifieNonP(L) =\n\t%s"\
        %toutXverifieNonP(L))
```

```
existeXverifiantP(L) =
        True
existeXverifiantNonP(L) =
        False
toutXverifieP(L) =
        False
toutXverifieNonP(L) =
        True
```

## quantificationsRetour

```
import random

def P(X):
    return random.randrange(2)==0

def existeXverifiantP(L):
    for x in L:
        if P(x):
            return True
    return False

def existeXverifiantNonP(L):
    for x in L:
        if not P(x):
            return True
    return False

def toutXverifieP(L):
    for x in L:
        if not P(x):
            return False
    return True

def toutXverifieNonP(L):
    for x in L:
        if P(x):
            return False
    return True
```

```
from quantificationsRetour import *

L = [1,2,3,4]
print("existeXverifiantP(L) =\n\t%s"\
        %existeXverifiantP(L))
print("existeXverifiantNonP(L) =\n\t%s"\
        %existeXverifiantNonP(L))
print("toutXverifieP(L) =\n\t%s"\
        %toutXverifieP(L))
print("toutXverifieNonP(L) =\n\t%s"\
        %toutXverifieNonP(L))
```

```
existeXverifiantP(L) =
        False
existeXverifiantNonP(L) =
        True
toutXverifieP(L) =
        False
toutXverifieNonP(L) =
        False
```

## extremumPredicats

```
import random

def P(X):
    return random.randrange(2)==0

def minXverifiantP(T):
    iMin = None
    minX = None
```

```
    for i in range(len(T)):
        if P(T[i]) and (minX==None or T[i]<minX):
            iMin = i
            minX = T[i]
    return [minX,iMin]

def maxXverifiantP(T):
    iMax = None
    maxX = None
    for i in range(len(T)):
        if P(T[i]) and (maxX==None or T[i]>maxX):
            iMax = i
            maxX = T[i]
    return [maxX,iMax]
```

```
from extremumPredicats import *

T = [1,2,3,4]
print("T␣=%s"%T)
print("minXverifiantP(T)␣=␣%s"\
          %minXverifiantP(T))
print("maxXverifiantP(T)␣=␣%s"\
          %maxXverifiantP(T))
```

```
T =[1, 2, 3, 4]
minXverifiantP(T) = [1, 0]
maxXverifiantP(T) = [3, 2]
```

## A.3 Algorithmes de tri et de recherche

### decalages

```
def estIndice(T,i):
    return 0<=i and i<len(T)

def echange(T,i,j):
    assert(estIndice(T,i) and estIndice(T,j))
    aux = T[i]
    T[i] = T[j]
    T[j] = aux

def decalageDroite(T,g,d):
    assert(g<=d and estIndice(T,g) and estIndice(T,d))
    aux = T[d]
    for k in range(d,g,-1):
        T[k] = T[k-1]
    T[g] = aux

def decalageGauche(T,g,d):
    assert(g<=d and estIndice(T,g) and estIndice(T,d))
    aux = T[g]
    for k in range(g,d):
        T[k] = T[k+1]
    T[d] = aux
```

```
import random
from decalages import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Tableau initial: \n\t%s"\
        %T)
echange(T,3,6)
print("Apres echange T[3] et T[6] : \n\t%s"\
        %T)
decalageDroite(T,4,7)
print("Apres decalageDroite(T,4,7) : \n\t%s"\
        %T)
decalageGauche(T,2,5)
print("Apres decalageGauche(T,2,5) : \n\t%s"\
        %T)
```

```
Tableau initial:
        [5, 6, 1, 6, 6, 3, 8, 3]
Apres echange T[3] et T[6] :
        [5, 6, 1, 8, 6, 3, 6, 3]
Apres decalageDroite(T,4,7) :
        [5, 6, 1, 8, 3, 6, 3, 6]
Apres decalageGauche(T,2,5) :
        [5, 6, 8, 3, 6, 1, 3, 6]
```

## triSelection

```
from decalages import *

def triSelection(T):
    for i in range(len(T)-1):
        iMin = i
        for j in range(i+1,len(T)):
            if T[j]<T[iMin]:
                iMin = j
        if iMin!=i:
            echange(T,i,iMin)
```

```
import random
from triSelection import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Avant tri selection : \n\t%s"\
        %T)
triSelection(T)
print("Apres tri selection : \n\t%s"\
        %T)
```

```
Avant tri selection :
        [2, 8, 8, 1, 7, 2, 1, 6]
Apres tri selection :
        [1, 1, 2, 2, 6, 7, 8, 8]
```

## triBulle

```
from decalages import *

def triBulle(T):
    for i in range(len(T)-1,0,-1):
        for j in range(i):
            if T[j]>T[j+1]:
                decalageDroite(T,j,j+1)
```

```
import random
from triBulle import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Avant tri bulle : \n\t%s"\
        %T)
triBulle(T)
print("Apres tri bulle : \n\t%s"\
        %T)
```

```
Avant tri bulle :
        [5, 8, 9, 0, 9, 8, 1, 2]
Apres tri bulle :
        [0, 1, 2, 5, 8, 8, 9, 9]
```

## triInsertion

```
from decalages import *

def triInsertion(T):
    for i in range(1,len(T)):
        j = i-1;
        while j>=0 and T[i]<T[j]:
            j = j-1
        decalageDroite(T,j+1,i)
```

```
import random
from triInsertion import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Avant tri insertion : \n\t%s"\
        %T)
triInsertion(T)
print("Apres tri insertion : \n\t%s"\
        %T)
```

```
Avant tri insertion :
        [0, 2, 7, 6, 6, 3, 5, 1]
Apres tri insertion :
        [0, 1, 2, 3, 5, 6, 6, 7]
```

## triDenombrement

```
def triDenombrement(T,minorant,majorant):
    assert(minorant<=majorant)
    for i in range(len(T)):
        assert(minorant<=T[i] and T[i]<=majorant)
    # T[i] entier entre minorant et majorant
    freq = [0]*(majorant-minorant+1)
    res = [0]*(len(T))
    for i in range(len(T)):
        freq[T[i]-minorant] = freq[T[i]-minorant]+1
    freq[0] = freq[0]-1
    for i in range(1,majorant-minorant+1):
        freq[i] = freq[i]+freq[i-1]
    for i in range(len(T)):
        res[freq[T[i]-minorant]] = T[i]
        freq[T[i]-minorant] = freq[T[i]-minorant]-1
    return res
```

```
import random
from triDenombrement import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
min = T[0]
max = T[0]
for i in range(1,len(T)):
    if T[i]<min:
        min = T[i]
    elif T[i]>max:
        max = T[i]
print("Avant tri denombrement : \n\t%s"\
        %T)
T = triDenombrement(T,min,max)
print("Apres tri denombrement : \n\t%s"\
        %T)
```

```
Avant tri denombrement :
        [5, 5, 3, 8, 1, 4, 4, 2]
Apres tri denombrement :
        [1, 2, 3, 4, 4, 5, 5, 8]
```

## rechercheElement

```
import random

def rechercheElement(T,X):
    # retourne s'il existe l'indice du premier element egal a X,
```

```
    # sinon la valeur "None"
    for i in range(len(T)):
        if T[i]==X:
            return i
    return None

from rechercheElement import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("La liste : \n\t%s"\
        %T)
print("rechercheElement(T,3) : \n\t%s"\
        %rechercheElement(T,3))
print("rechercheElement(T,5) : \n\t%s"\
        %rechercheElement(T,5))
```

```
La liste :
        [1, 7, 9, 7, 5, 8, 9, 0]
rechercheElement(T,3) :
        None
rechercheElement(T,5) :
        4
```

### rechercheDichotomique

```
import random

def estTrie(T):
    for i in range(1,len(T)):
        if T[i]<T[i-1]:
            return False
    return True

def rechercheDichotomique(T,X):
    assert(estTrie(T))
    # T est trie
    # Retourne s'il existe l'indice d'un element egal a X,
    # sinon la valeur "None"
    g = 0
    d = len(T)-1
    while g<=d:
        m = (g+d)//2
        if T[m]==X:
            return m
        elif T[m]<X:
            g = m+1
        else:
            d = m-1
    return None

from rechercheDichotomique import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
T = sorted(T)
print("La liste : \n\t%s"\
        %T)
print("rechercheDichotomique(T,3) : \n\t%s"\
        %rechercheDichotomique(T,3))
print("rechercheDichotomique(T,5) : \n\t%s"\
        %rechercheDichotomique(T,5))
```

```
La liste :
        [1, 1, 1, 2, 2, 5, 6, 7]
rechercheDichotomique(T,3) :
        None
rechercheDichotomique(T,5) :
        5
```

## A.4 La Récursivité

### pgcd

```
# Complexite :
# nb appels recursifs : entre 1 et max(a,b)//2
# - temps : omega(1), O(max(a,b))
# - espace (hauteur pile) : omega(1), O(max(a,b))
def pgcdRecursif(a,b):
    assert(a>0 and b>0)
    if (a==1)or(b==1):
        return 1
    elif a==b:
        return a
    elif a>b:
        return pgcdRecursif(a-b,b)
    else:
        return pgcdRecursif(a,b-a)

# Complexite :
# - temps : omega(1), O(max(a,b))
# - espace : theta(1)
def pgcdIteratif(a,b):
    assert(a>0 and b>0)
    while a>1 and b>1 and a!=b:
        if a>b:
            a = a-b
        else:
            b = b-a
    if a==b:
        return a
    else:
        return 1

# Complexite :
# - temps : omega(1), O(max(a,b))
# - espace : theta(1)
def pgcdIteratifRetour(a,b):
    assert(a>0 and b>0)
    while True:
        if (a==1)or(b==1):
            return 1
        elif a==b:
            return a
        elif a>b:
            a = a-b
        else:
            b = b-a

def tikzPgcdRecursif(a,b,label,la,lr):
    if a==1 or b==1:
        lr['res{0}'.format(label)]=1
        return 1,la,lr
    elif a==b:
        lr['res{0}'.format(label)]=a
        return a,la,lr
    elif a>b:
        la = la + ['child␣{{{{node␣{{{{pgcd({0},{1}):{{0[res{2}]}}}}}}}\n'.format(a-b,b,lab
        p,la,lr = tikzPgcdRecursif(a-b,b,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr
    else:
        la = la + ['child␣{{{{node␣{{{{pgcd({0},{1}):{{0[res{2}]}}}}}}}\n'.format(a,b-a,lab
        p,la,lr = tikzPgcdRecursif(a,b-a,label+'0',la,lr)
        la = la + ['}}\n']
```

```
        lr['res{0}'.format(label)]=p
        return p,la,lr

def tikzAppelsPgcdRecursif(a,b,fichier='pgcd.tex',grow='up',ld=3,sd=3.5):
    assert(a>0 and b>0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{pgcd({0},{1}):{{0[res{2}]}}}}}}[grow\'={grow}]\n'.format(a,b,label,gr
    lr = dict()
    p,la,lr = tikzPgcdRecursif(a,b,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()

from pgcd import *

print("pgcdRecursif(42,66) =\n\t%s"\
        %pgcdRecursif(42,66))
print("pgcdIteratif(42,66) =\n\t%s"\
        %pgcdIteratif(42,66))
print("pgcdIteratifRetour(42,66) =\n\t%s"\
        %pgcdIteratifRetour(42,66))
```

```
pgcdRecursif(42,66) =
        6
pgcdIteratif(42,66) =
        6
pgcdIteratifRetour(42,66) =
        6
```

```
tikzAppelsPgcdRecursif(24,30,grow='up',ld=2,sd=5)
```

## factorielle

```
# Complexite :
#  nb appels recursifs : n-1
# - temps : theta(n)
# - espace (hauteur pile) : theta(n)
def factorielleRecursif(n):
    assert(n>=0)
    if n<2:
        return 1
    else:
        return n * factorielleRecursif(n-1)

# Complexite :
# - temps : theta(n)
# - espace : theta(1)
def factorielleIteratif(n):
    assert(n>=0)
    res = 1;
    for i in range(2,n+1):
        res *= i
    return res

# Complexite :
#  nb appels recursifs : n-1
# - temps : theta(n)
# - espace (hauteur pile) : theta(n)
def factorielleRecursifTerminal(n, u=1):
    assert(n>=0)
    if n<2:
        return u
    else:
        return factorielleRecursifTerminal(n-1, n*u)
```

```
# Complexite :
# - temps : theta(n)
# - espace : theta(1)
def factorielleIteratifAutomatique(n):
    assert(n>=0)
    u=1
    while (True):
        if n<2:
            return u
        else:
            # n,u = n-1,n*u
            u = n*u
            n = n-1

def tikzFactorielleRecursif(n,label,la,lr):
    if n<2:
        lr['res{0}'.format(label)]=1
        return 1,la,lr
    else:
        la = la + ['child {{{{node {{{{factorielle({0}):{{0[res{1}]}}}}}}}}\n'.format(n-1,la
        p,la,lr = tikzFactorielleRecursif(n-1,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=n*p
        return n*p,la,lr

def tikzAppelsFactorielleRecursif(n,fichier='factorielle.tex',grow='up',ld=3,sd=3.5):
    assert(n>=0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{factorielle({0}):{{0[res{1}]}}}}}}[grow\'={grow}]\n'.format(n,label,
    lr = dict()
    p,la,lr = tikzFactorielleRecursif(n,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()

def tikzFactorielleRecursifTerm(n,u,label,la,lr):
    if n<2:
        lr['res{0}'.format(label)]=u
        return u,la,lr
    else:
        la = la + ['child {{{{node {{{{factorielle({0},{1}):{{0[res{2}]}}}}}}}}\n'.format(n-
        p,la,lr = tikzFactorielleRecursifTerm(n-1,n*u,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr

def tikzAppelsFactorielleRecursifTerm(n,u=1,fichier='factorielleTerminale.tex',grow='up',l
    assert(n>=0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{factorielle({0},{1}):{{0[res{2}]}}}}}}[grow\'={grow}]\n'.format(n,u,
    lr = dict()
```

```
    p,la,lr = tikzFactorielleRecursifTerm(n,u,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()

from factorielle import *

print("factorielleRecursif(11) =\n\t%s"\
         %factorielleRecursif(11))
print("factorielleIteratif(11) =\n\t%s"\
         %factorielleIteratif(11))
print("factorielleRecursifTerminal(11) =\n\t%s"\
         %factorielleRecursifTerminal(11))
print("factorielleIteratifAutomatique(11) =\n\t%s"\
         %factorielleIteratifAutomatique(11))

tikzAppelsFactorielleRecursif(8,grow='up',ld=2.5,sd=5)
tikzAppelsFactorielleRecursifTerm(8,1,grow='up',ld=2.5,sd=5)
```

```
factorielleRecursif(11) =
        39916800
factorielleIteratif(11) =
        39916800
factorielleRecursifTerminal(1
        39916800
factorielleIteratifAutomatiqu
        39916800
```

## fibonacci

```
# Complexite :
# nb appels : (n/2)**2 < fibonacci(n) < n**2
# - temps : theta(n**2)
# - espace (hauteur pile) : theta(n)
def fibonacciRecursif(n):
    assert(n>=0)
    if n<2:
        return n
    else:
        return fibonacciRecursif(n-1)\
            + fibonacciRecursif(n-2)

# Complexite :
# nb appels : n-1
# - temps : theta(n)
# - espace (hauteur pile) : theta(n)
def fibonacciRecursifTerminal(n, u=0, v=1):
    assert(n>=0)
    if n==0:
        return u
    elif n==1:
        return v
    else:
        return fibonacciRecursifTerminal(n-1, v, u+v)

# Complexite :
# - temps : theta(n)
# - espace : theta(1)
def fibonacciIteratifAutomatique(n):
    assert(n>=0)
    u=0
    v=1
    while True:
        if n==0:
            return u
        elif n==1:
            return v
        else:
            # parallele n, u, v = n-1, v, u+v
            # possible d'ecrire la suite : n := n-1; v := u+v; u := v-u
            aux = v
            v = u+v
```

```
            u = aux
            n = n-1

def tikzFibonacciRecursif(n,label,la,lr):
    if n<2:
        lr['res{0}'.format(label)]=n
        return n,la,lr
    else:
        la = la + ['child {{{{node {{{{f({0}):{{0[res{1}]}}}}}}}}\n'.format(n-1,label+'0')]
        p_1,la,lr = tikzFibonacciRecursif(n-1,label+'0',la,lr)
        la = la + ['}}\n']
        la = la + ['child {{{{node {{{{f({0}):{{0[res{1}]}}}}}}}}\n'.format(n-2,label+'1')]
        p_2,la,lr = tikzFibonacciRecursif(n-2,label+'1',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p_1 + p_2
        return p_1 + p_2,la,lr

def tikzAppelsFibonacciRecursif(n,fichier='fibonacci.tex',grow='up',ld=3,sd=3.5):
    assert(n>=0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    hauteur = n
    for i in range(1,hauteur):
        f.write('level {}/.style={{sibling distance={}em}},\n'.format(hauteur-i,sd*2**(i-1
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{f({0}):{{0[res{1}]}}}}}}[grow\'={grow}]\n'.format(n,label,grow=grow)
    lr = dict()
    p,la,lr = tikzFibonacciRecursif(n,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()

def tikzFibonacciRecursifTerm(n,u,v,label,la,lr):
    if n==0:
        lr['res{0}'.format(label)]=u
        return u,la,lr
    elif n==1:
        lr['res{0}'.format(label)]=v
        return v,la,lr
    else:
        la = la + ['child {{{{node {{{{fibonacci({0},{1},{2}):{{0[res{3}]}}}}}}}}\n'.format(
        p,la,lr = tikzFibonacciRecursifTerm(n-1,v,u+v,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr

def tikzAppelsFibonacciRecursifTerm(n,fichier='fibonacciTerminale.tex',grow='up',ld=3,sd=3
    assert(n>=0)
    u = 0
    v = 1
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{fibonacci({0},{1},{2}):{{0[res{3}]}}}}}}[grow\'={grow}]\n'.format(n,
    lr = dict()
```

```
    p,la,lr = tikzFibonacciRecursifTerm(n,u,v,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()

from fibonacci import *

print("fibonacciRecursif(8) =\n\t%s"\
        %fibonacciRecursif(8))
print("fibonacciRecursifTerminal(8) =\n\t%s"\
        %fibonacciRecursifTerminal(8))
print("fibonacciIteratifAutomatique(8) =\n\t%s"\
        %fibonacciIteratifAutomatique(8))

tikzAppelsFibonacciRecursif(5,grow='up',ld=3,sd=3.5)
tikzAppelsFibonacciRecursifTerm(5,grow='up',ld=2.5,sd=3.5)
```

```
fibonacciRecursif(8) =
        21
fibonacciRecursifTerminal(8) =
        21
fibonacciIteratifAutomatique(8)
        21
```

## ackermann

```
# Complexite :
# nb appels : (2***(m-2))(n + 3) - 3
# - temps : theta(n(2**m))
# - espace (hauteur pile) : ?
def ackermann(m,n):
    assert(m>=0 and n>=0)
    if m==0:
        return n+1
    elif n==0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1,ackermann(m,n-1))

def knuth(r,a,b):
    if r==1:
        return a**b
    else:
        res = 1
        for i in range(b):
            res = knuth(r-1,a,res)
        return res

def ackermannAnalyse(m,n):
    assert(m>=0 and n>=0)
    if m==0:
        return n+1
    elif m==1:
        return n+2
    elif m==2:
        return 2*n+3
    else:
        return knuth(m-2,2,n+3)-3

def tikzAckermannRecursif(m,n,label,la,lr):
    if m==0:
        lr['res{0}'.format(label)]=n+1
        return n+1,la,lr
    elif n==0:
        la = la + ['child {{{{node {{{{a({0},{1})\\\\{{0[res{2}]}}}}}}}}\n'.format(m-1,1,lab
        p,la,lr = tikzAckermannRecursif(m-1,1,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr
    else:
```

```
        la = la + ['child␣{{{{node␣{{{{a({0},{1})\\\\{{0[res{2}]}}}}}}}\n'.format(m,n-1,lab
        p_1,la,lr = tikzAckermannRecursif(m,n-1,label+'0',la,lr)
        la = la + ['}}\n']
        la = la + ['child␣{{{{node␣{{{{a({0},{1})\\\\{{0[res{2}]}}}}}}}\n'.format(m-1,p_1,l
        p_2,la,lr = tikzAckermannRecursif(m-1,p_1,label+'1',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p_2
        return p_2,la,lr

def tikzAppelsAckermannRecursif(m,n,fichier='ackermann.tex',grow='up',ld=3,sd=2,raison=2):
    assert(m>=0 and n>=0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge␣from␣parent␣fork␣{grow}\n'.format(grow=grow))
    hauteur = ackermannAnalyse(m,n)
    #for i in range(1,hauteur):
    for i in range(1,hauteur):
        if i<hauteur//3:
            f.write(',level␣{}/.style={{font=\\normalsize}}\n'.format(i))
        elif i<2*hauteur//3:
            f.write(',level␣{}/.style={{font=\\footnotesize}}\n'.format(i))
        else:
            f.write(',level␣{}/.style={{font=\\tiny}}\n'.format(i))
        #f.write(',level {}/.style={{sibling distance={}em}}\n'.format(hauteur-i,sd*raison
    f.write(',level/.style={{sibling␣distance={}em/#1}}\n'.format(sd))
    f.write(',level␣distance={}em\n'.format(ld))
    f.write(',align=center\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node␣{{{{a({0},{1})\\\\{{0[res{2}]}}}}}}}[grow\'={grow}]\n'.format(m,n,label,g
    lr = dict()
    p,la,lr = tikzAckermannRecursif(m,n,label,la,lr)
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()


import sys
from ackermann import *
from ackermannAnalyse import *

#tikzAppelsAckermannRecursif(2,2,grow='right',ld=6,sd=1.5,raison=1.8)
tikzAppelsAckermannRecursif(2,2,grow='up',ld=4,sd=27,raison=1.7)

print("ackermann(2,4)␣=\n\t%s"\
        %ackermann(2,4))
print("ackermann(3,2)␣=\n\t%s"\
        %ackermann(3,2))
try:
    print("ackermann(4,1)␣=\n\t%s"\
        %ackermann(4,1))
except:
    print("Unexpected␣error:")
    print("\t", sys.exc_info()[0])
    print("\t", sys.exc_info()[1])

print("ackermannAnalyse(2,4)␣=\n\t%s"\
        %ackermannAnalyse(2,4))
print("ackermannAnalyse(3,2)␣=\n\t%s"\
        %ackermannAnalyse(3,2))
print("ackermannAnalyse(4,1)␣=\n\t%s"\
        %ackermannAnalyse(4,1))
```

```
ackermann(2,4) =
        11
ackermann(3,2) =
        29
Unexpected error:
         <class 'Ru
         maximum rec
ackermannAnalyse(2,4
        11
ackermannAnalyse(3,2
        29
ackermannAnalyse(4,1
        65533
```

## syracuse

```
# Complexite :
# nb appels : entre log2(n) et ?
# - temps : omega(log2(n), theta inconnue
# - espace (hauteur pile) : ?
def syracuse(n):
    assert(n>0)
    if n==1:
        return [n]
    elif n%2==0:
        return [n] + syracuse(n//2)
    else:
        return [n]+ syracuse(3*n+1)

# Complexite :
# - temps : omega(log2(n), theta inconnue
# - espace : theta(1)
def syracuseIteratif(n):
    assert(n>0)
    l = [n]
    while True:
        if n==1:
            return l
        elif n%2==0:
            n = n//2
        else:
            n = 3*n+1
        l = l+[n]

def tikzSyracuseRecursif(n,label,la,lr):
    if n==1:
        lr['res{0}'.format(label)]=1
        return 1,la,lr
    elif n%2==0:
        la = la + ['child {{{{node {{{{syracuse({0}):{{0[res{1}]}}}}}}}}\n'.format(n//2,labe
        p,la,lr = tikzSyracuseRecursif(n//2,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr
    else:
        la = la + ['child {{{{node {{{{syracuse({0}):{{0[res{1}]}}}}}}}}\n'.format(3*n+1,lab
        p,la,lr = tikzSyracuseRecursif(3*n+1,label+'0',la,lr)
        la = la + ['}}\n']
        lr['res{0}'.format(label)]=p
        return p,la,lr

def tikzAppelsSyracuseRecursif(n,fichier='syracuse.tex',grow='up',ld=3,sd=3.5):
    assert(n>0)
    f = open(fichier,'w')
    f.write('\\begin{tikzpicture}\n')
    f.write('[edge from parent fork {grow},\n'.format(grow=grow))
    """
    hauteur = n
    for i in range(1,hauteur):
        f.write('level {}/.style={{sibling distance={}em}},\n'.format(hauteur-i,sd*2**(i-1
    """
    f.write('level distance={}em\n'.format(ld))
    f.write(']\n')
    label = '0'
    la = ['\\node {{{{syracuse({0}):{{0[res{1}]}}}}}}[grow\'={grow}]\n'.format(n,label,grow
    lr = dict()
    p,la,lr = tikzSyracuseRecursif(n,label,la,lr)
```

```
    f.write(''.join(la).format(lr))
    f.write(';\n')
    f.write('\\end{tikzpicture}\n')
    f.close()
```

```
from syracuse import *

print("syracuse(20)␣=\n\t%s"\
        %syracuse(20))
print("syracuseIteratif(20)␣=\n\t%s"\
        %syracuseIteratif(20))

tikzAppelsSyracuseRecursif(20,grow='up',ld=2,sd=5)
```

```
syracuse(20) =
        [20, 10, 5, 16, 8, 4, 2, 1]
syracuseIteratif(20) =
        [20, 10, 5, 16, 8, 4, 2, 1]
```

## A.5 Algorithmes récursifs de recherche et de tri

### triFusion

```
def fusionner(T,g,m,d):
    R = [0]*(d-g+1)
    i = g
    j = m+1
    k = 0
    while i<=m and j<=d:
        if T[i]<=T[j]:
            R[k] = T[i]
            i = i+1
        else:
            R[k] = T[j]
            j = j+1
        k = k+1
    while i<=m:
        R[k] = T[i]
        i = i+1
        k = k+1
    while j<=d:
        R[k] = T[j]
        j = j+1
        k = k+1
    for k in range(len(R)):
        T[g+k] = R[k]

def triFusionRec(T,g,d):
    if g<d:
        m = (g+d)//2
        triFusionRec(T,g,m)
        triFusionRec(T,m+1,d)
        fusionner(T,g,m,d)

def triFusion(T):
    triFusionRec(T,0,len(T)-1)
```

```
import random
from triFusion import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Avant tri fusion : \n\t%s"\
        %T)
triFusion(T)
print("Apres tri fusion : \n\t%s"\
        %T)
```

```
Avant tri fusion :
        [1, 5, 7, 9, 8, 8, 6, 0]
Apres tri fusion :
        [0, 1, 5, 6, 7, 8, 8, 9]
```

**triRapide**

```
def echange(T,i,j):
    aux = T[i]
    T[i] = T[j]
    T[j] = aux

def partitionner(T,g,d):
    pivot = T[g]
    i = g-1
    j = d+1
    while True:
        i = i+1
        while T[i]<pivot:
            i = i+1
        j = j-1
        while T[j]>pivot:
            j = j-1
        if i<j:
            echange(T,i,j)
            print("limite %s\t%s\t indices %s\t%s" %(g,d,i,j))
        else:
            return j

def triRapideRec(T,g,d):
    if g<d:
        m = partitionner(T,g,d)
        triRapideRec(T,g,m)
        triRapideRec(T,m+1,d)

def triRapide(T):
    triRapideRec(T,0,len(T)-1)
```

```
import random
from triRapide import *

T = []
for i in range(8):
    T = T + [random.randrange(10)]
print("Avant tri rapide : \n\t%s"\
        %T)
triRapide(T)
print("Apres tri rapide : \n\t%s"\
        %T)
```

```
Avant tri rapide :
        [0, 7, 0, 3, 6, 8, 3, 8]
limite 0        7       indices 0       2
limite 1        7       indices 1       6
limite 1        4       indices 1       3
limite 1        2       indices 1       2
limite 5        7       indices 5       7
limite 5        6       indices 5       6
Apres tri rapide :
        [0, 0, 3, 3, 6, 7, 8, 8]
```

## A.6 Introduction à la preuve d'algorithmes et de programmes

**drapeauDijkstra**

```
from decalages import *
```

```python
def drapeauDijkstra(T):
    i = 0
    j = 0
    k = len(T)-1
    while j!=k+1:
        if T[j]==0:
            echange(T,i,j)
            i = i+1
            j = j+1
        elif T[j]==1:
            j = j+1
        else:
            echange(T,j,k)
            k = k-1
    return i,j
```

```python
import random
from drapeauDijkstra import *

T = []
for i in range(10):
    T = T + [random.randrange(4)]
print("Avant partitionnement : \n\t%s"\
        %T)
drapeauDijkstra(T)
print("Apres partitionnement : \n\t%s"\
        %T)
```

```
Avant partitionnement :
        [2, 3, 1, 1, 1, 0, 1, 2, 0, 2]
Apres partitionnement :
        [0, 0, 1, 1, 1, 1, 2, 3, 2, 2]
```

## drapeauDijkstraAnnote

```python
from decalages import *

def drapeauDijkstraAnnote(T):
    assert(Hyp(T))
    i = 0
    assert(P1(T,i))
    j = 0
    assert(P2(T,i,j))
    k = len(T)-1
    assert(P3(T,i,j,k))
    assert(Inv(T,i,j,k))
    while j!=k+1:
        assert(Inv(T,i,j,k) and j!=k+1)
        if T[j]==0:
            assert(Inv(T,i,j,k) and j!=k+1 and T[j]==0)
            echange(T,i,j)
            assert(Inv11(T,i,j,k) and j!=k+1)
            i = i+1
            assert(Inv12(T,i,j,k) and j!=k+1)
            j = j+1
            assert(Inv(T,i,j,k))
        elif T[j]==1:
            assert(Inv(T,i,j,k) and j!=k+1 and T[j]==1)
            j = j+1
            assert(Inv(T,i,j,k))
        else:
            assert(Inv(T,i,j,k) and j!=k+1 and T[j]>1)
            echange(T,j,k)
            assert(Inv21(T,i,j,k) and j!=k+1)
            k = k-1
            assert(Inv(T,i,j,k))
        assert(Inv(T,i,j,k))
    assert(Inv(T,i,j,k) and j==k+1)
```

```
    return i,j

def Hyp(T):
    for l in range(len(T)):
        if T[l]<0:
            return False
    return True

def P1(T,i):
    return Hyp(T) and i==0

def P2(T,i,j):
    return Hyp(T) and i==0 and j==0

def P3(T,i,j,k):
    return Hyp(T) and i==0 and j==0 and k==len(T)-1

def Inv(T,i,j,k):
    for l in range(len(T)):
        if l<i and T[l]!=0:
            return False
        elif i<=l and l<j and T[l]!=1:
            return False
        elif k+1<=l and T[l]<2:
            return False
    return True

def Inv11(T,i,j,k):
    for l in range(len(T)):
        if l<i+1 and T[l]!=0:
            return False
        elif i+1<=l and l<j+1 and T[l]!=1:
            return False
        elif k+1<=l and T[l]<2:
            return False
    return True

def Inv12(T,i,j,k):
    for l in range(len(T)):
        if l<i and T[l]!=0:
            return False
        elif i<=l and l<j+1 and T[l]!=1:
            return False
        elif k+1<=l and T[l]<2:
            return False
    return True

def Inv21(T,i,j,k):
    for l in range(len(T)):
        if l<i and T[l]!=0:
            return False
        elif i<=l and l<j and T[l]!=1:
            return False
        elif k<=l and T[l]<2:
            return False
    return True
```

```
import random
from drapeauDijkstraAnnote import *

T = []
for i in range(10):
    T = T + [random.randrange(4)]
print("Avant␣partitionnement␣:␣\n\t%s"\
        %T)
drapeauDijkstraAnnote(T)
print("Apres␣partitionnement␣:␣\n\t%s"\
        %T)
```

```
Avant partitionnement :
        [2, 3, 3, 1, 3, 0, 3, 1, 2, 2]
Apres partitionnement :
        [0, 1, 1, 3, 3, 3, 3, 2, 2, 2]
```

## A.7 Morceaux choisis

### plscRecursif

```
def plscRecursif(u,v):
    if len(u)==0 or len(v)==0:
        return ''
    if u[0]==v[0]:
        x = u[0]
        return x+plscRecursif(u[1:],v[1:])
    else:
        p1 = plscRecursif(u[1:],v)
        p2 = plscRecursif(u,v[1:])
        if len(p1)>=len(p2):
            return p1
        else:
            return p2
```

```
from plscRecursif import *

u='aabbccdd'
v='abbbcccdeeeee'
plsc = plscRecursif(u,v)
print('plscRecursif')
print('\tu=%s,␣v=%s␣\n\tplsc=%s'%(u,v,plsc))
```

```
plscRecursif
        u=aabbccdd, v=abbbcccdeeeee
        plsc=abbccd
```

### plscDynamique

```
def plscDynamique(u,v):
    res = ['']*(len(u)+1)
    for i in range(len(u)+1):
        res[i] = ['']*(len(v)+1)
    for i in range(1,len(u)+1):
        for j in range(1,len(v)+1):
            if u[i-1]==v[j-1]:
                res[i][j] = res[i-1][j-1]+u[i-1]
            else:
                if len(res[i][j-1])>=len(res[i-1][j]):
                    res[i][j] = res[i][j-1]
                else:
                    res[i][j] = res[i-1][j]
    return res[len(u)][len(v)]
```

```
from plscDynamique import *

u='aabbccdd'
v='abbbcccdeeeee'
plsc = plscDynamique(u,v)
print('plscDynamique')
print('\tu=%s,␣v=%s␣\n\tplsc=%s'%(u,v,plsc))
```

```
plscDynamique
        u=aabbccdd, v=abbbcccdeeeee
        plsc=abbccd
```

## plsc

```
def plscCodage(u,v):
    # code = [[0]*(len(v)+1)]*(len(u)+1) ne fonctionne pas
    # car cela fabrique len(u)+1 references sur la meme liste
    # et donc des effets de bord malencontreux !
    code = [0]*(len(u)+1)
    for i in range(len(code)):
        code[i] = [0]*(len(v)+1)
    for i in range(1,len(u)+1):
        for j in range(1,len(v)+1):
            if u[i-1]==v[j-1]:
                code[i][j] = code[i-1][j-1]+1
            else:
                code[i][j] = max(code[i][j-1],code[i-1][j])
    return code

def plscDecodage(u,v,code):
    plsc = ''
    i = len(u)
    j = len(v)
    while i>0 and j>0 and code[i][j]>0:
        if u[i-1]==v[j-1]:
            plsc = u[i-1]+plsc
            i = i-1
            j = j-1
        elif code[i][j-1]>=code[i-1][j]:
            j = j-1
        else:
            i = i-1
    return plsc
```

```
from plsc import *

u='aabbccdd'
v='abbbcccdeeeee'
code = plscCodage(u,v)
for i in range(len(code)):
    print(i,code[i])
plsc = plscDecodage(u,v,code)
print('plsc : codage + decodage')
print('\tu=%s, v=%s'%(u,v))
print('\tplsc=%s'%(plsc))
```

```
0 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
2 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
3 [0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
4 [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
5 [0, 1, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4]
6 [0, 1, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5]
7 [0, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 6, 6, 6]
8 [0, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 6, 6, 6]
plsc : codage + decodage
        u=aabbccdd, v=abbbcccdeeeee
        plsc=abbccd
```

## rechercheMotifNaif

```
def rechercheMotifNaif(T,P):
    li = []
    for i in range(len(T)-len(P)):
        j = 0;
        coincide = True
        while coincide and j<len(P):
            coincide = T[i+j]==P[j]
            j = j+1
        if coincide:
            li = li + [i]
    return li
```

```
from rechercheMotifNaif import *

print("rechercheMotifNaif('dabacdbbabdcabc','ab') =\n\t%s"\
        %rechercheMotifNaif('dabacdbbabdcabc','ab'))
```

```
rechercheMotifNaif('dabacdbbab
        [1, 8, 12]
```

## rechercheMotifAutomate

```
def construireAlphabet(T):
    sigma = []
    for i in range(len(T)):
        if not T[i] in sigma:
            sigma = sigma + [T[i]]
    return sigma

def indiceLettre(sigma,x):
    for i in range(len(sigma)):
        if sigma[i]==x:
            return i
    return None

def suffixe(P,k,q,x):
    # vrai ssi Pk est un suffixe de Pq.x
    if k==0: # mot vide est toujours suffixe
        return True
    if P[k-1]!=x:
        return False
    for i in range(k-1):
        if P[k-2-i]!=P[q-1-i]:
            return False
    return True

def constructionAutomate(P,sigma):
    m = len(P)
    delta = [0]*(m+1)
    for q in range(m+1):
        delta[q] = [0]*len(sigma)
    for q in range(m+1):
        for x in range(len(sigma)):
            # calcul du plus long suffixe Pk de Pq.x
            k = min(m,q+1)
            while not suffixe(P,k,q,sigma[x]):
                k = k-1
            delta[q][x] = k
    return delta

def rechercheMotifAutomate(T,P):
    sigma = construireAlphabet(T)
    delta = constructionAutomate(P,sigma)
    li = []
    q = 0
    for i in range(len(T)):
        q = delta[q][indiceLettre(sigma,T[i])]
        if q==len(P):
            li = li + [i-len(P)+1]
    return li
```

```
from rechercheMotifAutomate import *

print("rechercheMotifAutomate('dabacdbbabdcabc','ab') =\n\t%s"\
      %rechercheMotifAutomate('dabacdbbabdcabc','ab'))
```

```
rechercheMotifAutomate('da
        [1, 8, 12]
```

## rechercheMotifKMP

```
def constructionFonctionPrefixe(P):
    # prefixe[q] = max k tel que Pk suffixe propre de P(q+1)
    # preferable de revenir a un tableau 1..n ?
    prefixe = [0]*(len(P))
    k = 0
    for q in range(1,len(P)):
```

```
        while k>0 and P[k]!=P[q]:
            k = prefixe[k-1]
        if P[k]==P[q]:
            k = k+1
        prefixe[q] = k
    return prefixe

def rechercheMotifKMP(T,P):
    prefixe = constructionFonctionPrefixe(P)
    li = []
    q = 0
    for i in range(len(T)):
        while q>0 and P[q]!=T[i]:
            q = prefixe[q-1]
        if P[q]==T[i]:
            q = q+1
        if q==len(P):
            li = li + [i-len(P)+1]
            q = prefixe[q-1]
    return li
```

```
from rechercheMotifKMP import *

print("rechercheMotifKMP('dabacdbbabdcabc','ab') =\n\t%s"\
        %rechercheMotifKMP('dabacdbbabdcabc','ab'))
print("rechercheMotifKMP('dabacdbbabdcabc','bbab') =\n\t%s"\
        %rechercheMotifKMP('dabacdbbabdcabc','bbab'))
```

```
rechercheMotifKMP('dabacdbbab
        [1, 8, 12]
rechercheMotifKMP('dabacdbbab
        [6]
```