Algorithmique avancée et programmation C
Exercices de TD
3.3.4
avec corrections

N. Delestre

Table des matières

Avant propos

Évaluation par attendus d'apprentissages disciplinaires

Depuis l'année universitaire 2018-2019, la validation du cours ≪ Algorithique avancée et programmation C ≫ utilise une évaluation par attendus d'apprentissages disciplinaires (AAD). Le référentiel des AAD est disponible sur le site Moodle de l'INSA Rouen Normandie : `https://moodle.insa-rouen.fr/course/view.php?id=60&section=0`.

Les exercices de ce document vous permettent de travailler ces AAD.

Quelque soit l'exercice les AAD suivants sont évalués :

- AN001 : Désigner les choses (identifiant significatif)
- AN002 : Être précis quant aux types de données utilisés
- AN003 : Connaître le rôle de l'analyse
- CP001 : Comprendre le paradigme de programmation impératif
- CP002 : Comprendre le paradigme de programmation structuré
- CP006 : Comprendre le rôle de la conception préliminaire
- CD004 : Écrire des algos avec le pseudo code utilisé à l'INSA
- CD005 : Écrire un pseudo code lisible (indentation, identifiant significatif)
- CD006 : Choisir la bonne itération
- CD007 : Utiliser les bonnes catégories de paramètres effectifs pour un passage de paramètre donnée
- CD009 : Écrire un algorithme qui résout le problème
- CD010 : Connaître le rôle de la conception détaillée

Le tableau ci dessous croise les exercices de ce livret avec les autres compétences :

Croisement AAD - exercices

AAD	Exercices
AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple	3.4, 7, 12, 13
AN101 : Identifier les entrées et sorties d'un problème	1.3, 2.4, 4, 5
AN102 : Décomposer logiquement un problème	2.4, 4
AN103 : Généraliser un problème	4
AN104 : Savoir si un problème doit être décomposé	2.4
AN201 : Identifier les dépendances d'un TAD	6, 8, 12
AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier	6, 8, 12

AAD	Exercices
AN204 : Formaliser des opérations d'un TAD	6, 12
AN205 : Formaliser les préconditions d'une opération d'un TAD	6, 8
AN206 : Formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD	6, 12
AN301 : Lister les collections usuelles	8
CP003 : Choisir entre une fonction et une procédure	1.3, 4, 5, 6, 8, 12
CP004 : Concevoir une signature (préconditions incluses)	1.1, 1.2,1.3, 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 3.5, 4, 5, 6, 12
CP005 : Choisir un passage de paramètre (E, S, E/S)	2.2, 5, 6, 12
CD001 : Dissocier les deux rôles du développeur : concepteur et utilisateur	6
CD002 : En tant qu'utilisateur, respecter une signature	1.1, 1.2
CD003 : Utiliser le principe d'encapsulation	6, 8
CD101 : Estimer la taille d'un problème (n)	1.4, 4
CD102 : Calculer une complexité dans le pire et le meilleur des cas	1.4, 4, 7
CD104 : Écrire un algorithme d'une complexité donnée	2.3, 3.2, 3.3
CD201 : Identifier et résoudre le problème des cas non récursifs	3.1, 3.2, 3.3, 3.4, 3.5, 7, 8, 10, 12
CD202 : Identifier et résoudre le problème des cas récursifs	3.1, 3.2, 3.3, 3.4, 3.5, 7, 8, 10, 12
CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique	3.1, 3.2, 3.3, 3.4, 3.5
CD301 : Identifier un problème qui se résout à l'aide d'un algorithme dichotomique	2.3
CD302 : Définir l'espace de recherche d'un algorithmique dichotomique	1.4, 2.3
CD303 : Diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)	1.4, 2.3
CD403 : Concevoir et utiliser des arbres (binaires, n-aires)	10
CD501 : Comprendre les algorithmes des différents tris et leurs complexités	7
CD601 : Concevoir des collections à l'aide de SDD	10
CD602 : Comprendre les algorithmes d'insertion et de suppression (naïfs et AVL) dans un arbre binaire de recherche	10
CD701 : Définir la programmation dynamique	13
CD702 : Appliquer la programmation dynamique pour des cas simples	13
CD801 : Concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)	12

AAD	Exercices
CD804 : Comprendre des algorithmes de recherche du plus court chemin : Dijkstra et A*	12
CD901 : Concevoir un type de données adapté à la situation en terme d'espace mémoire et d'efficacité	9, 10

Pseudo code

Vous écrirez vos algorithmes avec le pseudo code utilisé dans la plupart des cours d'algorithmique de l'INSA Rouen Normandie. Voici la syntaxe des instructions disponibles :

Type de données

Les types de base sont : **Entier**, **Naturel**, **NaturelNonNul**, **Reel**, **ReelPositif**, **ReelPositifNonNul**, **ReelNegatif**, **ReelNegatifNonNul**, **Booleen**, **Caractere**, **Chaine de caracteres**.

On définit un nouveau type de la façon suivante :

Type Identifiant_nouveau_type **=** Identifiant_type_existant

On déclare un tableau de la façon suivante :

— Tableau à une dimension : **Tableau[**borne_de_début...borne_de_fin**] de** type_des_éléments

— Tableau à deux dimensions : **Tableau[**borne_de_début...borne_de_fin**][**borne_de_début...borne_de_fin**] de** type_des_éléments

— ...

On définit une structure de la façon suivante :

Type Identifiant **= Structure**
    identifiant_attribut_1 : Type_1
    ...
finstructure

Affectation

Le symbole d'affectation est ←.

Conditionnelles

Il y a trois instructions conditionnelles :

si condition **alors**
    instruction(s)
finsi

si condition **alors**
    instruction(s)
sinon
    instruction(s)
finsi

cas où identifiant_variable **vaut**
    valeur_1:
        instruction(s)_1
    ...
    autre :
        instruction(s)
fincas

Itérations

L'instruction de base pour les itérations déterministes est le **pour** :

pour identifiant ←borne_de_début **à** borne_de_fin **faire**
    instruction(s)

finpour

On peut itérer sur les éléments d'une liste, d'une liste ordonnée ou d'un ensemble grâce à l'instruction **pour chaque** :

pour chaque élément **de** collection
  instruction(s)
finpour

Pour les itérations indéterministes nous avons deux instructions :

tant que condition **faire**
  instruction(s)
fintantque

repeter
  instruction(s)
jusqu'a ce que condition

Sous-programmes

Les fonctions permettent de calculer un résultat (composé d'une ou plusieurs valeurs) de manière déterministe :

fonction identifiant **(**paramètre(s)_formel(s)**) :** Type(s) de retour
  ⌊**précondition(s)** expression(s) booléenne(s)
  Déclaration variable(s) locale(s)
debut
  instruction(s) avec au moins une fois l'instruction **retourner**
fin

Les procédures permettent de créer de nouvelles instructions :

procédure identifiant **(**paramètre(s)_formel(s)_avec_passage_de_paramètres**)**
  ⌊**précondition(s)** expression(s) booléenne(s)
  Déclaration variable(s) locale(s)
debut
  instruction(s)
fin

Les passages de paramètre sont : entrée (**E**), sortie (**S**) et entrée/sortie (**E/S**).

Chapitre 1

Rappels : chaîne de caractères, itérations, conditionnelles

Pour certains de ces exercices on considère que l'on possède les fonctions suivantes :

— **fonction** longueur (uneChaine : **Chaine de caracteres) : Naturel**

— **fonction** iemeCaractere (uneChaine : **Chaine de caracteres**, iemePlace : **Naturel) : Caractere**

⌊**précondition(s)** $0 < iemePlace$ et $iemePlace \leq longueur(uneChaine)$

1.1 estUnPrefixe

Attendus d'apprentissages disciplinaires évalués

— CP004 : Concevoir une signature (préconditions incluses)

— CD002 : En tant qu'utilisateur, respecter une signature

— CD006 : Choisir la bonne itération

Proposez la fonction `estUnPrefixe` qui permet de savoir si une première chaîne de caractères est préfixe d'une deuxième chaîne de caractères (par exemple ≪ pré ≫ est un préfixe de ≪ prédire ≫ et de ≪ pré ≫).

Correction proposée:

fonction estUnPrefixe (lePrefixePotentiel,uneChaine : **Chaine de caracteres) : Booleen**

Déclaration i : **NaturelNonNul**
resultat : **Booleen**

debut
si longueur(lePrefixePotentiel)>longueur(uneChaine) **alors**
retourner FAUX
sinon
i ← 1
resultat ← VRAI
tant que resultat et i≤longueur(lePrefixePotentiel) **faire**
si iemeCaractere(uneChaine,i)=iemeCaractere(lePrefixePotentiel,i) **alors**
i ← i+1
sinon
resultat ← FAUX
finsi

```
        fintantque
        retourner resultat
    finsi
fin
```

1.2 Palindrome

Attendus d'apprentissages disciplinaires évalués

- CP004 : Concevoir une signature (préconditions incluses)
- CD002 : En tant qu'utilisateur, respecter une signature
- CD006 : Choisir la bonne itération

Une chaîne de caractères est un palindrome si la lecture de gauche à droite et de droite à gauche est identique. Par exemple "radar", "été", "rotor", etc. La chaîne de caractères vide est considérée comme étant un palindrome

Écrire une fonction qui permet de savoir si une chaîne est un palindrome.

Correction proposée:

```
fonction estUnPalindrome (ch : Chaine de caracteres) : Booleen

    Déclaration g,d : NaturelNonNul
                resultat : Booleen

debut
    si longueur(ch)=0 alors
        retourner VRAI
    sinon
        resultat ← VRAI
        g ← 1
        d ← longueur(ch)
        tant que resultat et g<d faire
            si iemeCaractere(ch,g) = iemeCaractere(ch,d) alors
                g ← g+1
                d ← d-1
            sinon
                resultat ← FAUX
            finsi
        fintantque
        retourner resultat
    finsi
fin
```

1.3 Position d'une sous-chaîne

Attendus d'apprentissages disciplinaires évalués

— AN101 : Identifier les entrées et sorties d'un problème
— CP003 : Choisir entre une fonction et une procédure
— CP004 : Concevoir une signature (préconditions incluses)

Soit l'analyse descendante présentée par la figure 1.1 qui permet de rechercher la position d'une chaîne de caractères dans une autre chaîne indépendemment de la casse (d'où le suffixe IC à l'opération `positionSousChaineIC`), c'est-à-dire que l'on ne fait pas de distinction entre majuscule et minuscule.

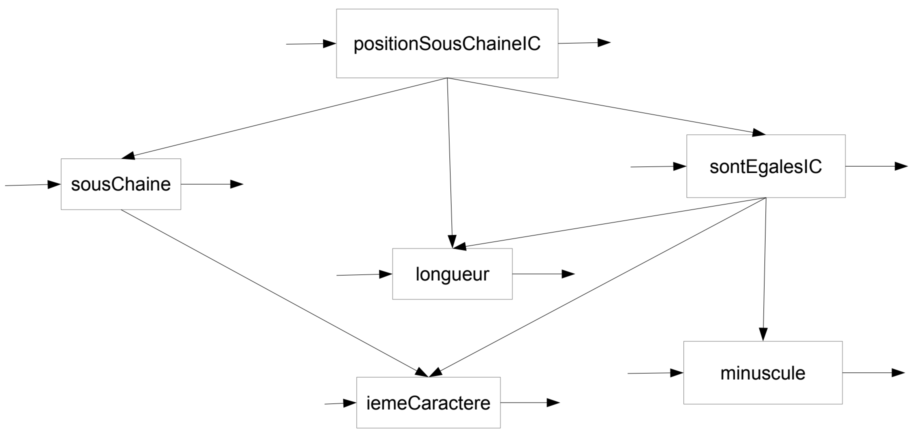


FIGURE 1.1 – Une analyse descendante

Pour résoudre ce problème il faut pouvoir :

— obtenir la longueur d'une chaîne de caractères ;
— obtenir la sous-chaîne d'une chaîne en précisant l'indice de départ de cette sous-chaîne et sa longueur (le premier caractère d'une sous-chaîne à l'indice 1) ;
— savoir si deux chaînes de caractères sont égales indépendemment de la casse.

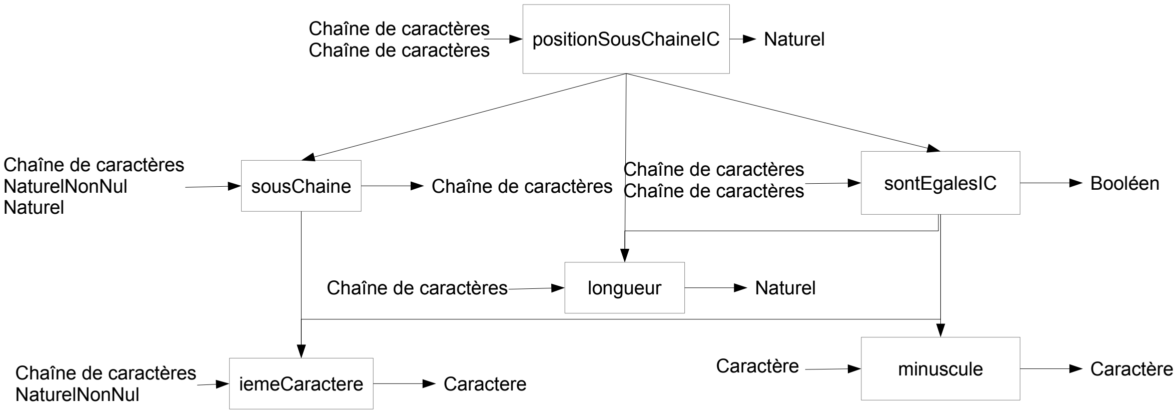
L'opération `positionSousChaineIC` retournera la première position de la chaîne recherchée dans la chaîne si cette première est présente, 0 sinon.

Par exemple :

— `positionSousChaineIC("AbCdEfGh","cDE")` retournera la valeur 3 ;
— `positionSousChaineIC("AbCdEfGh","abc")` retournera la valeur 1 ;
— `positionSousChaineIC("AbCdEfGh","xyz")` retournera la valeur 0.

1. Complétez l'analyse descendante en précisant les types de données en entrée et en sortie.
2. Donnez les signatures complètes (avec préconditions si nécessaire) des sous-programmes (fonctions ou procédures) correspondant aux opérations de l'analyse descendante.
3. Donnez l'algorithme du sous-programme correspondant à l'opération `positionSousChaineIC` et `sousChaine`

Correction proposée:



Note : minuscule est sur les caractères et non chaîne de caractères sinon il y aurait une autre sous boite…

fonction positionSousChaineIC (chaine, chaineARechercher : **Chaine de caracteres) : Naturel**

⌊**précondition(s)** longueur(chaineARechercher)>0
longueur(chaineARechercher)≤longueur(chaine)

fonction longueur (chaine : **Chaine de caracteres) : Naturel**

fonction sousChaine (chaine : **Chaine de caracteres**, pos : **NaturelNonNul**, long : **Naturel) : Chaine de caracteres**

⌊**précondition(s)** long≤longueur(chaine)-position+1

fonction sontEgalesIC (chaine1, chaine2 : **Chaine de caracteres) : Booleen**

fonction minuscule (c : **Caractere) : Caractere**

fonction positionSousChaineIC (chaine, chaineARechercher : **Chaine de caracteres) : Naturel**

⌊**précondition(s)** longueur(chaineARechercher)>0
longueur(chaineARechercher)≤longueur(chaine)

Déclaration i : **Naturel**

debut
i ← 1
tant que i+longueur(chaineARechercher)-1≤longueur(chaine) et non sontEgalesIC(sousChaine(chaine,i, longueur(chaineARechercher)),chaineARechercher) **faire**
i ← i+1
fintantque
si i+longueur(chaineARechercher)>longueur(chaine)+1 **alors**
i ← 0
finsi
retourner i
fin

fonction sousChaine (chaine : **Chaine de caracteres**, pos : **NaturelNonNul**, long : **Naturel) : Chaine de caracteres**

⌊**précondition(s)** long≤longueur(chaine)-pos+1

Déclaration resultat : **Chaine de caracteres**, i : **Naturel**

debut
resultat ← ””
pour i ←0 **à** long-1 **faire**
resultat ← resultat + iemeCaractere(chaine,pos+i)

finpour
retourner resultat
fin

1.4 Racine carrée d'un nombre : recherche par dichotomie

Attendus d'apprentissages disciplinaires évalués

— CD302 : Définir l'espace de recherche d'un algorithmique dichotomique
— CD303 : Diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)
— CD101 : Estimer la taille d'un problème (n)
— CD102 : Calculer une complexité dans le pire et le meilleur des cas

L'objectif de cet exercice est de rechercher une valeur approchée de la racine carrée d'un nombre réel positif x ($x \geq 1$) à ϵ près à l'aide d'un algorithme dichotomique.

Pour rappel :

> ≪ La dichotomie ("couper en deux" en grec) est, en algorithmique, un processus itératif [..] de recherche où, à chaque étape, on coupe en deux parties (pas forcément égales) un espace de recherche qui devient restreint à l'une de ces deux parties.
>
> On suppose bien sûr qu'il existe un test relativement simple permettant à chaque étape de déterminer l'une des deux parties dans laquelle se trouve une solution. Pour optimiser le nombre d'itérations nécessaires, on s'arrangera pour choisir à chaque étape deux parties sensiblement de la même "taille" (pour un concept de "taille" approprié au problème), le nombre total d'itérations nécessaires à la complétion de l'algorithme étant alors logarithmique en la taille totale du problème initial. ≫ (wikipédia).

1. Définir ≪ l'espace de recherche ≫ pour le problème de la recherche d'une racine carrée.
2. Quelle condition booléenne permet de savoir si il doit y avoir une nouvelle itération ?
3. Quel test va vous permettre de savoir dans laquelle des deux parties se trouve la solution ?
4. Proposez l'algorithme de la fonction suivante (on suppose que x et $epsilon$ sont positifs et que x est supérieur ou égal à 1) :
   — **fonction** racineCarree (x,epsilon : **ReelPositif) : ReelPositif**
5. Quelle est la complexité de votre algorithme ?

Correction proposée:

1. La taille de l'espace de recherche est : $(d - g)/\epsilon$.
2. $d - g > \epsilon$
3. m^2 plus petit ou plus grand que x
4.

fonction racineCarree (x,ϵ : **ReelPositif) : ReelPositif**
  Déclaration g,d,m : **ReelPositif**
debut
  g ← 0
  d ← x

tant que d-g> ϵ **faire**
  m ← (g+d)/2
  si m*m<x **alors**
    g ← m
  sinon
    d ← m
  finsi
fintantque
retourner g
fin

5. La taille du problème est définie par la valeur $(d-g)/\epsilon$. Le nombre d'itérations est donc de $log_2((d-g)/\epsilon)$.
   La représentation des flottants utilise un nombre fixe de bits (souvent la norme IEEE 754), Il y a donc une borne MAX. De plus chaque opération sur les flottants (comparaison, multiplication, division par 2) est dans ce cas supposée en temps constant, cet algorithme est $O(log_2((d-g)/\epsilon)$.

Chapitre 2

Rappels : les tableaux

Dans certains exercices qui vont suivre, le tableau d'entiers t est défini par [1..MAX] et il contient n éléments significatifs ($n \leq$ MAX).

2.1 Plus petit élément

Attendus d'apprentissages disciplinaires évalués
— CP004 : Concevoir une signature (préconditions incluses)

Écrire une fonction, `minTableau`, qui à partir d'un tableau d'entiers t non trié de n éléments significatifs retourne le plus petit élément du tableau.

Correction proposée:

fonction minTableau (t : **Tableau[**1..MAX**] d'Entier**, n : **NaturelNonNul) : Entier**

⌊**précondition(s)** n≤MAX

Déclaration i : **Naturel**,
min : **Entier**

debut
min ← t[1]
pour i ←2 **à** n **faire**
si t[i]<min **alors**
min ← t[i]
finsi
finpour
retourner min
fin

2.2 Sous-séquences croissantes

Attendus d'apprentissages disciplinaires évalués

- CP003 : Choisir entre une fonction et une procédure
- CP004 : Concevoir une signature (préconditions incluses)
- CP005 : Choisir un passage de paramètre (E, S, E/S)
- CD005 : Écrire un pseudo code lisible (indentation, identifiant significatif)

Écrire un sous-programme `sousSequencesCroissantes`, qui à partir d'un tableau d'entiers t de n éléments, fournit le nombre de sous-séquences strictement croissantes de ce tableau, ainsi que les indices de début et de fin de la plus grande sous-séquence. Exemple : t un tableau de 15 éléments : $1, 2, 5, 3, 12, 25, 13$ $, 8, 4, 7, 24, 28, 32, 11, 14$. Les séquences strictement croissantes sont : $< 1, 2, 5 >, < 3, 12, 25 >, < 13 >, < 8 >, < 4, 7, 24, 28, 32 >, < 11, 14 >$. Le nombre de sous-séquences est : 6 et la plus grande sous-séquence est : $< 4, 7, 24, 28, 32 >$. Donc dans ce cas les trois valeurs calculées seraient 6, 9 et 13.

Correction proposée:

```
fonction sousSequencesCroissantes (t :Tableau[1..MAX] d'Entier, n : NaturelNonNul) : NaturelNonNul,
NaturelNonNul, NaturelNonNul
    ⌊précondition(s)  n≤MAX
    Déclaration  i :Naturel
                 debutSequenceCourante, nbSsSequences, debutDeLaPlusGrandeSsSequence, finDeLaPlusGran-
                 deSsSequence : NaturelNonNul
debut
    si n>1 alors
        nbSsSequences ← 1
        debutDeLaPlusGrandeSsSequence ← 1
        finDeLaPlusGrandeSsSequence ← 1
        debutSequenceCourante ← 1
        pour i ←1 à n-1 faire
            si t[i]>t[i+1] alors
                nbSsSequences ← nbSsSequences+1
                si i-debutSequenceCourante>finDeLaPlusGrandeSsSequence-debutDeLaPlusGrandeSsSequence
                alors
                    debutDeLaPlusGrandeSsSequence ← debutSequenceCourante
                    finDeLaPlusGrandeSsSequence ← i
                finsi
                debutSequenceCourante ← i+1
            finsi
        finpour
        si n-debutSequenceCourante>finDeLaPlusGrandeSsSequence-debutDeLaPlusGrandeSsSequence alors

            debutDeLaPlusGrandeSsSequence ← debutSequenceCourante
            finDeLaPlusGrandeSsSequence ← n
        finsi
        retourner nbSsSequences, debutDeLaPlusGrandeSsSequence, finDeLaPlusGrandeSsSequence
    sinon
        retourner 1,1,1
```

finsi
fin

2.3 Recherche d'un élément en $O(log(n))$

Attendus d'apprentissages disciplinaires évalués

— CP004 : Concevoir une signature (préconditions incluses)
— CD104 : Écrire un algorithme d'une complexité donnée
— CD301 : Identifier un problème qui se résout à l'aide d'un algorithme dichotomique
— CD302 : Définir l'espace de recherche d'un algorithmique dichotomique
— CD303 : Diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)

Écrire une fonction, `recherche`, qui détermine le plus petit indice d'un élément, (dont on est sûr de l'existence) dans un tableau d'entiers t trié dans l'ordre croissant de n éléments en $O(log(n))$. Il peut y avoir des doubles (ou plus) dans le tableau.

Correction proposée:

fonction recherche **(**t : **Tableau[**1..MAX**] d'Entier**, n : **NaturelNonNul**, element : **Entier) : NaturelNonNul**

⌊**précondition(s)** n≤MAX
$\exists$ 1 $\leq$ i $\leq$ n tel que t[i] = element
estTrieEnOrdreCroissant(t)

Déclaration g,d,m : **Naturel**

debut
g ← 1
d ← n
tant que g $\neq$ d **faire**
m ← (g + d) div 2
si t[m] $\geq$ element **alors**
d ← m
sinon
g ← m + 1
finsi
fintantque
retourner d
fin

Quelques remarques sur les algorithmes dichotomiques sur du discret :

— On sort du tant quand les deux indices se croisent

— Il faut savoir quand ≪ garder ≫ l'élément du milieu (et donc quand l'exclure, sinon il y a un risque de boucle infinie). Ici, comme on cherche le plus petit indice de l'élément recherché, lorsque t[m] est cet élément, il faut le garder (c'est peut être lui qui est recherché).

2.4 Lissage de courbe

Attendus d'apprentissages disciplinaires évalués

- AN101 : Identifier les entrées et sorties d'un problème
- AN102 : Décomposer logiquement un problème
- AN104 : Savoir si un problème doit être décomposé

L'objectif de cet exercice est de développer un ≪ filtre non causal ≫, c'est-à-dire une fonction qui lisse un signal en utilisant une fenêtre glissante pour moyenner les valeurs (Cf. figure 2.1). Pour les premières et dernières valeurs, seules les valeurs dans la fenêtre sont prises en compte.

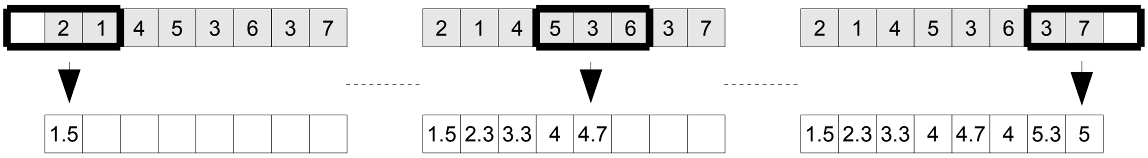


FIGURE 2.1 – Lissage d'un signal avec une fenêtre de taille 3

Soit le type `Signal` :

Type Signal **= Structure**
  donnees : **Tableau[**1..MAX**] de Reel**
  nbDonnees : **Naturel**
finstructure

Après avoir fait une analyse descendante du problème, proposez l'algorithme de la fonction `filtreNonCausal` avec la signature suivante :

- **fonction** filtreNonCausal (signalNonLisse : Signal, tailleFenetre : **NaturelNonNul) :** Signal
  ⌊**précondition(s)** impair(tailleFenetre)

Correction proposée:

Analyse descencante :

- $filtreNonCausal$: $Signal$ × **Naturel** $\nrightarrow$ $Signal$
  - min : **Naturel** × **Entier** → **Entier**
  - max : **Naturel** × **Entier** → **Entier**
  - $moyenne$: $Signal$ × **Naturel** × **Naturel** $\nrightarrow$ **Reel**
    - $somme$: $Signal$ × **Naturel** × **Naturel** $\nrightarrow$ **Reel**

Algorithmes :

fonction somme (unSignal : Signal, debut, fin : **NaturelNonNul) : Reel**
  ⌊**précondition(s)** debut≤ fin
    fin≤ unSignal.nbDonnees
    unSignal.nbDonnees≤ MAX

Déclaration resultat : **Reel**
  i : **Naturel**

debut
resultat ← 0
pour i ←debut **à** fin **faire**
resultat ← resultat+ unSignal.donnes[i]
finpour
retourner resultat
fin
fonction moyenne (unSignal : Signal, debut, fin : **NaturelNonNul) : Reel**

⌊**précondition(s)** debut≤ fin
fin≤ unSignal.nbDonnees
unSignal.nbDonnees≤ MAX

debut
retourner somme(unSignal,debut,fin)/(fin-debut+1)
fin
fonction filtreNonCausal (unSignal : Signal, tailleFenetre : **NaturelNonNul) :** Signal

⌊**précondition(s)** impaire(tailleFenetre)
unSignal.nbDonnees≤ MAX

Déclaration resultat : Signal
i : **Naturel**

debut
resultat.nbDonnees ← unSignal.nbDonnees
pour i ←1 **à** resultat.nbDonnees **faire**
resultat.donnes[i] ← moyenne(unSignal,entierEnNaturel(max(1,i-tailleFenetre div 2)), entierEnNaturel(min(unSignal.nbDonnees,i+tailleFenetre div 2)))
finpour
retourner resultat
fin

Il est noté qu'il faut explicitement utiliser la fonction de transtypage `entierEnNaturel` qui possède la signature suivante :

— **fonction** entierEnNaturel (e : **Entier) : Naturel**

⌊**précondition(s)** e≥0

Chapitre 3

Rappels : récursivité

3.1 Palindrome

Attendus d'apprentissages disciplinaires évalués
— CP004 : Concevoir une signature (préconditions incluses) — CD201 : Identifier et résoudre le problème des cas non récursifs — CD202 : Identifier et résoudre le problème des cas récursifs — CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

Écrire une fonction qui permet de savoir si une chaîne est un palindrome. Est-ce un algorithme récursif terminal ou non-terminal ?

Correction proposée:

fonction estUnPalindrome (uneChaine : **Chaine de caracteres) : Booleen**
debut
    si longueur(uneChaine)=0 ou longueur(uneChaine)=1 **alors**
        retourner VRAI
    sinon
        si iemeCaractere(uneChaine,1)$\neq$iemeCaractere(uneChaine,longueur(uneChaine)) **alors**
            retourner FAUX
        sinon
            retourner estUnPalindrome(sousChaine(uneChaine,2,longueur(uneChaine)-2))
        finsi
    finsi
fin

Le problème est que c'est algorithme est en $O(n^2)$. Pour obtenir un algorithme en $O(n)$, il faut utiliser une fonction privée prenant en paramètre le chaine et les indices :

fonction estUnPalindrome (uneChaine : **Chaine de caracteres) : Booleen**
debut
    retourner estUnPalindromeR(uneChaine,1,longueur(uneChaine)-1)
fin
fonction estUnPalindromeR (uneChaine : **Chaine de caracteres**, debut, fin : **NaturelNonNul) : Booleen**
debut
    si fin$\leq$debut **alors**
        retourner VRAI

sinon
si iemeCaractere(uneChaine,debut)≠iemeCaractere(uneChaine,fin) **alors**
retourner FAUX
sinon
retourner estUnPalindromeR(sousChaine(uneChaine,debut+1,fin-1))
finsi
finsi
fin

Il est noté que ces deux algorithmes sont des algorithmes récursif terminal.

3.2 Puissance d'un nombre

Attendus d'apprentissages disciplinaires évalués

- CP004 : Concevoir une signature (préconditions incluses)
- CD104 : Écrire un algorithme d'une complexité donnée
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs
- CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

Écrire une fonction récursive, `puissance`, qui élève un réel a à la puissance nb (naturel) en $\Omega(n)$.
Correction proposée:

fonction puissance (a : **Reel**, nb : **Naturel) : Reel**

Déclaration temp : **Reel**

debut
si nb = 0 **alors**
retourner 1
sinon
si estPair(nb) **alors**
temp ← puissance(a,nb div 2)
retourner temp*temp
sinon
retourner a*puissance(a,nb-1)
finsi
finsi
fin

Pour rappel, la taille du problème n ici est le nombre de bits qu'il faut pour représenter nb. Donc nb vaut au maximum 2^n. Dans le meilleur des cas l'algorithme divise nb par 2, le nombre d'itérations dans le meilleur des cas est donc de $log_2(nb)$ et donc la complexité de cet algorithme est en $\emptyset(n * log_2(n))$.

Il est noté que cet algorithme n'est pas une récursivité terminale.

3.3 Recherche du zéro d'une fonction en $O(n)$

Attendus d'apprentissages disciplinaires évalués

— CP004 : Concevoir une signature (préconditions incluses)
— CD104 : Écrire un algorithme d'une complexité donnée
— CD201 : Identifier et résoudre le problème des cas non récursifs
— CD202 : Identifier et résoudre le problème des cas récursifs
— CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

Écrire une fonction récursive, `zeroFonction`, qui calcule le zéro d'une fonction réelle $f(x)$ sur l'intervalle réel $[a, b]$, avec une précision ϵ. La fonction f est strictement monotone sur $[a, b]$.

Correction proposée:

fonction zeroFonction **(**a,b : **Reel**, ϵ : **ReelPositif**, f : FonctionRDansR**) : Reel**
⌊**précondition(s)** a $\leq$ b
strictementMonotone(f,a,b)

Déclaration m : **Reel**

debut
m $\leftarrow$ (a + b) / 2
si (b - a)$\leq$ ϵ **alors**
retourner m
sinon
si memeSigne(f(a),f(m)) **alors**
retourner zeroFonction(m, b, ϵ,f)
sinon
retourner zeroFonction(a, m, ϵ,f)
finsi
finsi
fin

La taille du problème est égal aux nombre de bits qu'il faut pour représenter ce $(b - a)/\epsilon$. Si on arrondit ce nombre au naturel le plus proche N, et si n représente le nombre de bits pour représenter N, N vaut au maximum $2^n - 1$. Comme le nombre d'itérations est de $log_2(N)$ (algorithmique dichotomique), la complexité de cet algorithme est en $O(n)$ et en $\Omega(1)$ (dans le cas où il n'y aucune itération).

3.4 Dessin récursif

Attendus d'apprentissages disciplinaires évalués

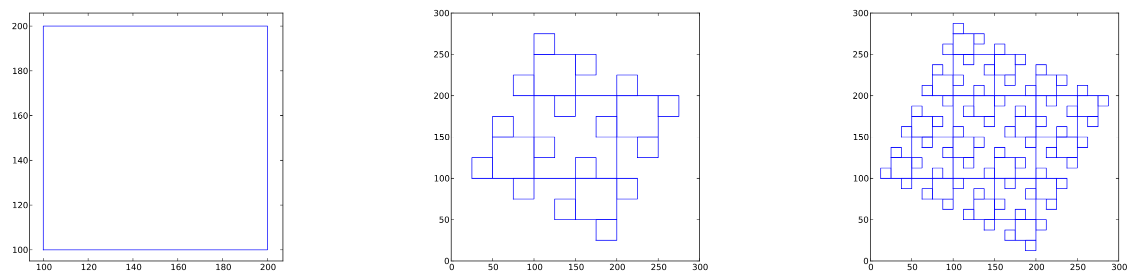
— AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple
— CD201 : Identifier et résoudre le problème des cas non récursifs
— CD202 : Identifier et résoudre le problème des cas récursifs
— CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

Supposons que la procédure suivante permette de dessiner un carré sur un graphique (variable de type `Graphique`) :

— **procédure** carre (**E/S** g : Graphique,**E** x,y,cote : **Reel)**

L'objectif est de concevoir une procédure `carres` qui permet de dessiner sur un graphique des dessins récursifs tels que présentés par la figure 3.1. La signature de cette procédure est :

— **procédure** carres (**E/S** g : Graphique,**E** x,y,cote : **Reel**, n : **NaturelNonNul)**

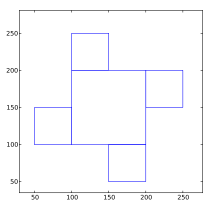


(a) $carres(g, 100, 100, 100, 1)$ (b) $carres(g, 100, 100, 100, 3)$ (c) $carres(g, 100, 100, 100, 4)$

FIGURE 3.1 – Résultats de différents appels de la procédure carres

1. Dessinez le résultat de l'exécution de $carres(g, 100, 100, 100, 2)$.
2. Donnez l'algorithme de la procédure `carres`.

Correction proposée:

1. 

2. Algorithme

   procédure carres (**E/S** g : Graphique,**E** x,y,cote : **Reel**, n : **NaturelNonNul)**
   debut
   carre(g,x,y,cote)
   si n>1 **alors**
   carres(g,x-cote/2,y,cote/2,n-1)
   carres(g,x,y+cote,cote/2,n-1)
   carres(g,x+cote,y+cote/2,cote/2,n-1)
   carres(g,x+cote/2,y-cote/2,cote/2,n-1)
   finsi
   fin

NB : Cet exercice est inspiré de `http://www-fourier.ujf-grenoble.fr/~parisse/giac/doc/fr/casrouge/casrouge018.html`.

3.5 Inversion d'un tableau

Attendus d'apprentissages disciplinaires évalués

- CP004 : Concevoir une signature (préconditions incluses)
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs
- CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

Soit un tableau d'entiers t. Écrire une procédure, `inverserTableau`, qui change de place les éléments de ce tableau de telle façon que le nouveau tableau t soit une sorte de "miroir" de l'ancien.

Exemple : 1 2 4 6 $\rightarrow$ 6 4 2 1

Correction proposée:

procédure inverserTableauR **(E/S** t : **Tableau[**1..MAX**] d'Entier**, **E** debut, fin : **Naturel)**
debut
  si debut < fin **alors**
    echanger(t[debut], t[fin])
    si debut<fin-1 **alors**
      inverserTableauR(t, debut+1, fin-1)
    finsi
  finsi
fin
procédure inverserTableau **(E/S** t : **Tableau[**1..MAX**] d'Entier**, **E** n : **Naturel)**
debut
  inverserTableauR(t,1,n)
fin

Chapitre 4

Représentation d'un naturel

Attendus d'apprentissages disciplinaires évalués

- AN101 : Identifier les entrées et sorties d'un problème
- AN102 : Décomposer logiquement un problème
- AN103 : Généraliser un problème
- AN104 : Savoir si un problème doit être décomposé
- CP003 : Choisir entre une fonction et une procédure
- CP004 : Concevoir une signature (préconditions incluses)
- CD001 : Dissocier les deux rôles du développeur : concepteur et utilisateur
- CD002 : En tant qu'utilisateur, respecter une signature

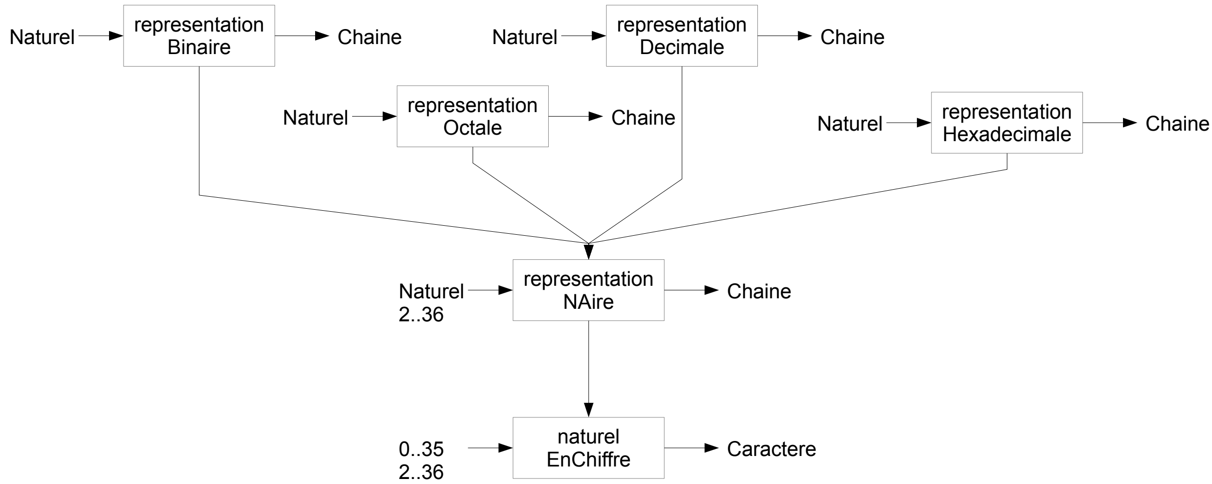
L'objectif de cet exercice est de concevoir quatre fonctions permettant de représenter un naturel en chaîne de caractères telles que la première fonction donnera une représentation binaire, la deuxième une représentation octale, la troisième une représentation décimale et la dernière une représentation hexadécimale.

4.1 Analyse

L'analyse de ce problème nous indique que ces quatre fonctions sont des cas particuliers de représentation d'un naturel en chaîne de caractères dans une base donnée. De plus pour construire la chaîne de caractères résultat, il faut être capable de concaténer des caractères représentant des chiffres pour une base donnée.

Proposez l'analyse descendante de ce problème.

Correction proposée:



4.2 Conception préliminaire

Donnez les signatures des fonctions ou procédures identifiées précédemment.

Correction proposée:

— **fonction** naturelEnChiffre (nombre : 0..35, base : 2..36) **: Caractere**
  ⌊**précondition(s)** nombre $<$ base
— **fonction** representationNAire (nombre : **Naturel**, base : 2..36) **: Chaine de caracteres**
— **fonction** representationBinaire (n : **Naturel) : Chaine de caracteres**
— **fonction** representationOctale (n : **Naturel) : Chaine de caracteres**
— **fonction** representationDecimale (n : **Naturel) : Chaine de caracteres**
— **fonction** representationHexadecimale (n : **Naturel) : Chaine de caracteres**

4.3 Conception détaillée

Donnez les algorithmes de ces fonctions ou procédures

Correction proposée:
fonction naturelEnChiffre (nombre : 0..35, base : 2..36) **: Caractere**
⌊**précondition(s)** nombre $<$ base
Déclaration chiffre : **Caractere**,
i : **Naturel**
debut
chiffre ← '0'
pour i ←1 **à** nombre **faire**

```
        si chiffre = '9' alors
            chiffre ← 'A'
        sinon
            chiffre ← succ(chiffre)
        finsi
    finpour
    retourner chiffre
fin

fonction representationNAire (nombre : Naturel, base : 2..36) : Chaine de caracteres
    Déclaration   representation : Chaine de caracteres
debut
    representation ← ""
    repeter
        representation ← naturelEnChiffre(nombre mod base, base) + representation
        nombre ← nombre div base
    jusqu'a ce que nombre = 0
    retourner representation
fin

fonction representationBinaire (n : Naturel) : Chaine de caracteres
debut
    retourner representationNAire(n,2)
fin

fonction representationOctale (n : Naturel) : Chaine de caracteres
debut
    retourner representationNAire(n,8)
fin

fonction representationDecimale (n : Naturel) : Chaine de caracteres
debut
    retourner representationNAire(n,10)
fin

fonction representationHexadecimale (n : Naturel) : Chaine de caracteres
debut
    retourner representationNAire(n,16)
fin
```

Chapitre 5

Calculatrice

Attendus d'apprentissages disciplinaires évalués

- AN101 : Identifier les entrées et sorties d'un problème
- CP003 : Choisir entre une fonction et une procédure
- CP004 : Concevoir une signature (préconditions incluses)
- CP005 : Choisir un passage de paramètre (E, S, E/S)

L'objectif de cet exercice est d'écrire un sous-programme, calculer, qui permet de calculer la valeur d'une une expression arithmétique simple (opérande gauche positive, opérateur, opérande droite positive) à partir d'une chaîne de caractères (par exemple "875+47.5"). Ce sous-programme, outre ce résultat, permettra de savoir si la chaîne est réellement une expression arithmétique (Conseil : Créer des procédures/fonctions permettant de reconnaître des opérandes et opérateurs) et si elle est logiquement valide

On considère posséder le type `Operateur` défini de la façon suivante :

- **Type** Operateur **=** $\{$Addition, Soustraction, Multiplication, Division$\}$

5.1 Analyse

Remplissez l'analyse descendante présentée par la figure 5.1 sachant que la reconnaissance d'une entité (opérateur, opérande, etc.) dans la chaîne de caractères commencent à une certaine position et que la reconnaissance peut échouer.

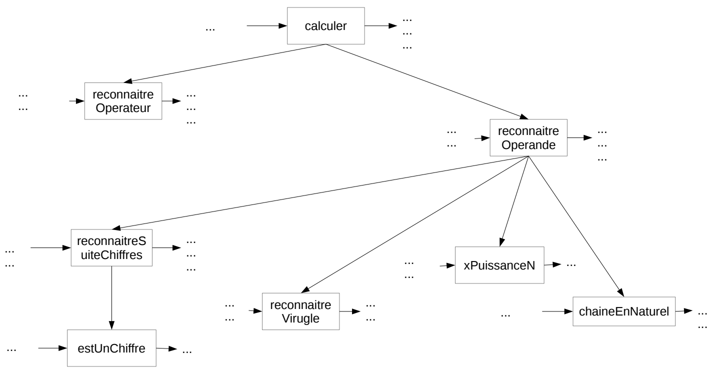
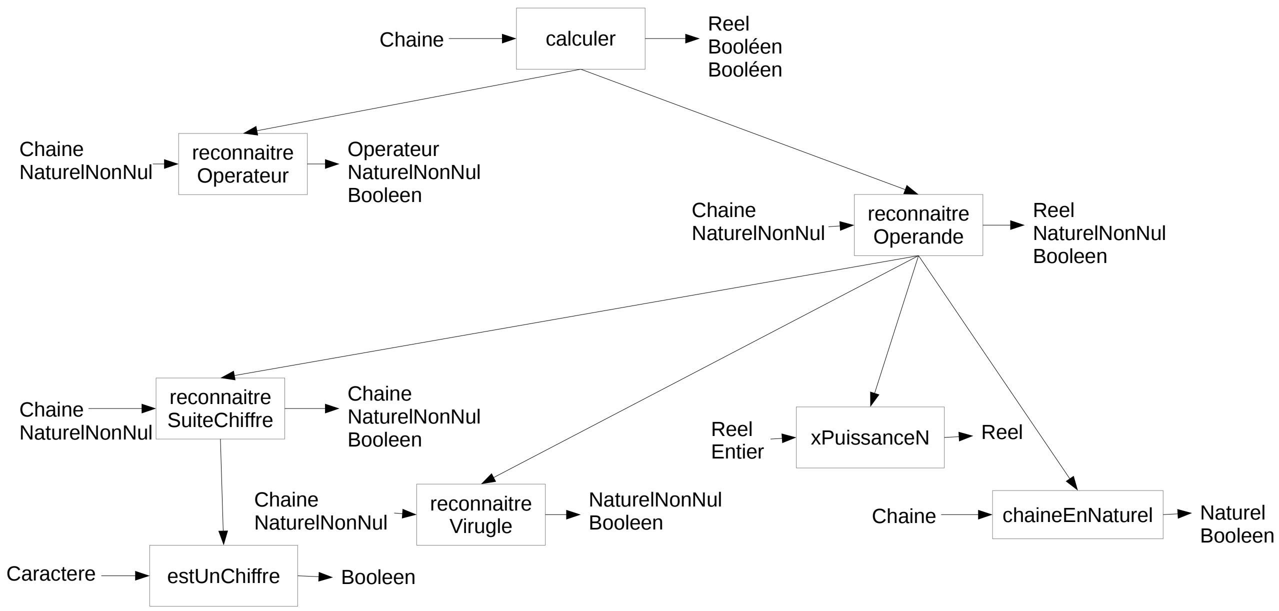


FIGURE 5.1 – Analyse descendante d'une calculatrice simple

Correction proposée:

Notes, remarques pour l'enseignant et points à vérifier

— La difficulté ici est d'avoir une analyse cohérente du problème



5.2 Conception préliminaire

Donnez les signatures des fonctions ou procédures correspondant aux opérations de l'analyse précédente.

Correction proposée:

— **fonction** calculer **(**leTexte : **Chaine de caracteres) : Reel**, **Booleen**, **Booleen**

⌊**précondition(s)** longueur(leTexte) > 0

— **procédure** reconnaitreOperateur **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** estUnOperateur : **Booleen**, lOperateur : Operateur**)**

⌊**précondition(s)** debut $<$ longueur(leTexte)

— **procédure** reconnaitreOperande **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** estUneOperande : **Booleen**, leReel : **Reel)**

⌊**précondition(s)** debut $\leq$ longueur(leTexte)

— **procédure** reconnaitreSuiteChiffres **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** suiteChiffres : **Chaine de caracteres**, estUneSuiteDeChiffres : **Booleen)**

⌊**précondition(s)** position $\leq$ longueur(leTexte)

— **procédure** reconnaitreVirgule **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** estUneVirgule : **Booleen)**

⌊**précondition(s)** position $\leq$ longueur(leTexte)

— **fonction** estUnChiffre (c : **Caractere) : Booleen**
— **fonction** XPuissanceN (x : **Reel**, n : **Entier) : Reel**
— **fonction** chaineEnNaturel (c : **Chaine de caracteres) : Naturel**, **Booleen**

5.3 Conception détaillée

Donnez les algorithmes des fonctions et procédures identifées.

Correction proposée:

Notes, remarques pour l'enseignant et points à vérifier

— Montrer qu'une fois la conception préliminaire terminée, on peut répartir la conception détaillée entre plusieurs personnes

procédure reconnaitreOperateur **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** estUnOperateur : **Booleen**, lOperateur : Operateur,)

⌊**précondition(s)** debut $\leq$ longueur(leTexte)

debut
estUnOperateur ← VRAI
position ← position+1
cas où iemeCaractere(leTexte,position) **vaut**
'+':
lOperateur ← Addition
'-':
lOperateur ← Soustraction
'*':
lOperateur ← Multiplication
'/':
lOperateur ← Division
autre :
estUnOperateur ← FAUX
position ← position-1
fincas
fin

fonction estUnChiffre **(**c : **Caractere) : Booleen**
debut
retourner c≥'0' et c≤'9'
fin

procédure reconnaitreSuiteChiffres **(E** leTexte : **Chaine de caracteres**, **E/S** position : **NaturelNonNul**, **S** suiteChiffres : **Chaine de caracteres**, estUneSuiteDeChiffres : **Booleen)**

⌊**précondition(s)** position $\leq$ longueur(leTexte)

debut
suiteChiffres ← ""
estUneSuiteDeChiffres ← FAUX
tant que position ≤ longueur(texte) et estUnChiffre(iemeCaractere (leTexte, position)) **faire**

suiteChiffres ← suiteChiffres + iemeCaractere (leTexte, position)
position ← position + 1
fintantque
si suiteChiffres≠”” **alors**
estUneSuiteDeChiffres ← VRAI
finsi
fin

procédure reconnaitreOperande **(E** leTexte : **Chaine de caracteres**, **E/S** position : **Naturel**, **S** estUneOperande : **Booleen**, leReel : **Reel**, prochainDebut : **NaturelNonNul)**

⌊**précondition(s)** debut ≤ longueur(leTexte)

Déclaration chPartieEntiere, chPartieDecimale : **Chaine de caracteres**
partieEntiere, partieDecimale : **Naturel**
ok, ilYAUneVirgule : **Booleen**

debut
reconnaitreSuiteChiffres(leTexte,position,chPartieEntiere,ok)
si ok **alors**
chaineEnNaturel(chPartieEntiere,partieEntiere,ok)
reconnaitreVirgule(leTexte,position,ilYAUneVirgule)
si ilYAUneVirgule **alors**
reconnaitreSuiteChiffres(leTexte,position,chPartieDecimale,ok)
si ok **alors**
chaineEnNaturel(chPartieDecimale,partieDecimale,ok)
leReel ← partieEntiere + partieDecimale / XPuissanceN(10,longueur(chPartieDecimale))
finsi
sinon
leReel ← naturelEnReel(partieEntiere)
finsi
finsi
estUneOperande ← ok
fin

fonction calculer (leTexte : **Chaine de caracteres) : Reel**, **Booleen**, **Booleen**

⌊**précondition(s)** longueur(leTexte) > 0

Déclaration i : **Naturel**
valeur, operandeG, operandeD : **Reel**
operateur : Operateur
toujoursValide, estUneExpressionSemantiquementCorrecte : **Booleen**

debut
valeur ← 0
i ← 1
reconnaitreOperande(leTexte,i, toujoursValide, operandeG)
si toujoursValide et i<longueur(leTexte) **alors**
reconnaitreOperateur(leTexte, i, toujoursValide, operateur)
si toujoursValide et i≤ longueur(leTexte) **alors**
reconnaitreOperande(leTexte, i, toujoursValide, operandeD)
si toujoursValide et i = longueur(leTexte) + 1 **alors**
estUneExpressionSemantiquementCorrecte ← VRAI

```
        cas où operateur vaut
          Addition:
            valeur ← operandeG + operandeD
          Soustraction:
            valeur ← operandeG - operandeD
          Multiplication:
            valeur ← operandeG * operandeD
          Division:
            si operandeD ≠ 0 alors
              valeur ← operandeG / operandeD
            sinon
              estUneExpressionSemantiquementCorrecte ← FAUX
            finsi
        fincas
        retourner valeur, VRAI, estUneExpressionSemantiquementCorrecte
      finsi
    finsi
  finsi
  retourner 0, FAUX, FAUX
fin
```

Chapitre 6

Un peu de géométrie

Correction proposée:

Notes, remarques pour l'enseignant et points à vérifier

— Manipuler les TAD
— Appliquer le principe d'encapsulation

Attendus d'apprentissages disciplinaires évalués

— AN201 : Identifier les dépendances d'un TAD
— AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
— AN204 : Formaliser des opérations d'un TAD
— AN205 : Formaliser les préconditions d'une opération d'un TAD
— AN206 : Formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD
— CP003 : Choisir entre une fonction et une procédure
— CP004 : Concevoir une signature (préconditions incluses)
— CP005 : Choisir un passage de paramètre (E, S, E/S)
— CD003 : Utiliser le principe d'encapsulation

6.1 Le TAD Point2D

Soit le TAD `Point2D` définit de la façon suivante :

Nom: Point2D
Utilise: **Reel**
Opérations: point2D: **Reel** $\times$ **Reel** $\rightarrow$ Point2D
obtenirX: Point2D $\rightarrow$ **Reel**
obtenirY: Point2D $\rightarrow$ **Reel**
distanceEuclidienne: Point2D $\times$ Point2D $\rightarrow$ **ReelPositif**
translater: Point2D $\times$ Point2D $\rightarrow$ Point2D
faireRotation: Point2D $\times$ Point2D $\times$ **Reel** $\rightarrow$ Point2D

1. Analyse : Donnez la partie axiomes pour ce TAD (sauf pour l'opération `faireRotation`)
   Correction proposée:

Axiomes:
- $obtenirX(point2D(x,y)) = x$
- $obtenirY(point2D(x,y)) = y$
- $distanceEuclidienne(point2D(x_1,y_1), point2D(x_2,y_2)) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$
- $translater(point2D(x_1,y_1), point2D(x_2,y_2)) = point2D(x_1+x_2, y_1+y_2)$

Remarque(s) :

— il ne sert à rien d'ajouter trop d'axiomes, au risque d'avoir un TAD inconsistant ou de proposer des tautologies.
Par exemple l'axiome $point2D(obtenirX(p1), obtenirY(p1)) = p1$ est une tautologie.
En effet si on remplace $p1$ par $point2D(x,y)$, on a alors :

$$point2D(obtenirX(point2D(x,y)), obtenirY(point2D(x,y))) = point2D(x,y)$$

Soit

$$point2D(x,y) = point2D(x,y)$$

qui est toujours vrai.

2. Conception préliminaire : Donnez les signatures des fonctions et procédures des opérations de ce TAD
   Correction proposée:

— **fonction** point2D **(**x,y : **Reel) :** Point2D
— **fonction** obtenirX **(**p : Point2D**) : Reel**
— **fonction** obtenirY **(**p : Point2D**) : Reel**
— **fonction** distanceEuclidienne **(**p1,p2 : Point2D**) : ReelPositif**
— **procédure** translater **(E/S** p : Point2D,**E** vecteur : Point2D**)**
— **procédure** realiserRotation **(E/S** p : Point2D,**E** centre : Point2D, angleEnDegre : **Reel)**

Remarque(s) :

— Il est important de choisir de bons identifiants pour les paramètres formels. Ici il pourrait y avoir ambiguïté sur l'unité du paramètre formel de l'angle de la rotation.

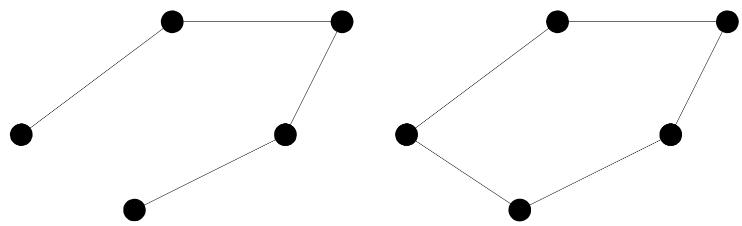
6.2 Polyligne

« Une ligne polygonale, ou ligne brisée (on utilise aussi parfois polyligne par traduction de l'anglais *polyline*) est une figure géométrique formée d'une suite de segments, la seconde extrémité de chacun d'entre eux étant la première du suivant.[...] Un polygone est une ligne polygonale fermée. » (Wikipédia)

La figure 6.1 présente deux polylignes composées de 5 points.

De cette définition nous pouvons faire les constats suivants :

— Tous les points d'une polyligne sont distincts ;
— Une polyligne est constituée d'au moins deux points ;



(a) polyligne ouverte (b) polyligne fermée

FIGURE 6.1 – Deux polylignes

- On peut obtenir le nombre de points d'une polyligne ;
- Une polyligne est ouverte ou fermée (qu'elle soit ouverte ou fermée ne change pas le nombre de points : dans le cas où elle est fermée, on considère qu'il a une ligne entre le dernier et le premier point) ;
- On peut insérer, supprimer des points à une polyligne (par exemple la figure 6.2 présente la supression du troisième point de la polyligne ouverte de la figure 6.1).
- On peut parcourir les points d'une polyligne ;
- On peut effectuer des transformations géométriques (translation, rotation, etc.) ;
- On peut calculer des propriétés d'une polyligne (par exemple sa longueur totale).

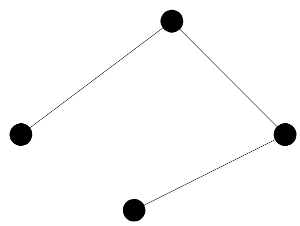


FIGURE 6.2 – Supression d'un point

6.2.1 Analyse

Proposez le TAD `Polyligne` (sans les parties Axiome et Sémantique) avec les opérations suivantes :

- créer une polyligne ouverte à partir de deux `Point2D` ;
- savoir si une polyligne est fermée ;
- ouvrir une polyligne ;
- fermer une polyligne ;
- connaitre le nombre de points d'un polyligne ;
- obtenir le ième point d'une polyligne ;
- insérer le ième point d'une polyligne ;
- supprimer le ième point d'une polyligne (on suppose qu'elle a au moins 3 points) ;
- calculer la longueur d'un polyligne ;

— translater une polyligne ;
— faire une rotation d'une polyligne.

Correction proposée:

Nom: Polyligne
Utilise: **Reel**,**Booleen**,**NaturelNonNul**,Point2D
Opérations:
polyligne: Point2D × Point2D ↛ Polyligne
estFermee: Polyligne → **Booleen**
ouvrir: Polyligne → Polyligne
fermer: Polyligne → Polyligne
nbPoints: Polyligne → **NaturelNonNul**
iemePoint: Polyligne × **NaturelNonNul** ↛ Point
ajouterPoint: Polyligne × Point × **NaturelNonNul** ↛ Point
supprimerPoint: Polyligne × **NaturelNonNul** ↛ Polyligne
longueur: Polyligne → **ReelPositif**
translater: Polyligne × Point2D → Polyligne
realiserRotation: Polyligne × Point2D × **Reel** → Polyligne
Préconditions:
$polyligne(pt1, pt2)$: $pt1 \neq pt2$
$iemePoint(pl, i)$: $i \leq nbPoints(pl)$
$ajouterPoint(pl, pt, i)$: $i \leq nbPoints(pl)$ et $\forall j \in 1..nbPoints(pl), iemePoint(pl, j) \neq pt$
$supprimerPoint(pl, i)$: $i \leq nbPoints(pl)$ et $nbPoints(pl) \geq 3$

Remarque(s) :
— Il est à noter que les trois dernières opérations ne sont pas obligatoires, elles pourraient être conçues en tant qu'utilisateur du TAD Polyligne.

6.2.2 Conception préliminaire

Proposez la signature des fonctions et procédures pour le type `Polyligne`.

Correction proposée:

— **fonction** polyligne (pt1,pt2 : Point2D) **:** Polyligne
  ⌊**précondition(s)** $pt1 \neq pt2$
— **fonction** estFermee (pl , Polyligne) **: Booleen**
— **procédure** fermer (**E/S** pl : Polyligne)
— **procédure** ouvrir (**E/S** pl : Polyligne)
— **fonction** nbPoints (pl : Polyligne) **: NaturelNonNul**
— **fonction** iemePoint (pl : Polyligne, position : **NaturelNonNul**) **:** Point2D
  ⌊**précondition(s)** $position \leq nbPoints(pl)$
— **procédure** ajouterPoint (**E/S** pl : Polyligne,**E** pt : Point2D, position : **NaturelNonNul**)
  ⌊**précondition(s)** $position \leq nbPoints(pl) + 1$ et $\forall i \in 1..nbPoints(pl), iemePoint(pl, i) \neq pt$

— **procédure** supprimerPoint **(E/S** pl : Polyligne,**E** position : **NaturelNonNul)**
  ⌊**précondition(s)** $position \leq nbPoints(pl)$ et $nbPoints(pl) \geq 3$
— **fonction** longueur (pl : Polyligne) **: ReelPositif**
— **procédure** translater **(E/S** pl : Polyligne,**E** vecteur : Point2D**)**
— **procédure** realiserRotation **(E/S** pl : Polyligne,**E** centre : Point2D, angleEnRadian : **Reel)**

6.2.3 Conception détaillée

On propose de représenter le type `Polyligne` de la façon suivante :

Type Polyligne **= Structure**
  lesPts : **Tableau**[1..MAX] **de** Point2D
  nbPts : **Naturel**
  estFermee : **Booleen**
finstructure

Proposez les fonctions et procédures correspondant aux opérations suivantes :

— créer une polyligne ouverte à partir de deux `Point2D` ;
— ouvrir une polyligne ;
— translater une polyligne.

Correction proposée:

fonction polyligne (pt1,pt2 : Point2D) **:** Polyligne
  Déclaration resultat : Polyligne
debut
  resultat.nbPts ← 2
  resultat.lesPts[1] ← pt1
  resultat.lesPts[2] ← pt2
  resultat.estFermee ← FAUX
  retourner resultat
fin
procédure ouvrir **(E/S** pl : Polyligne**)**
debut
  pl.estFermee ← FAUX
fin
procédure translater **(E/S** pl : Polyligne,**E** vecteur : Point2D**)**
  Déclaration i : **Naturel**
debut
  pour i ←1 **à** nbPoints(pl) **faire**
    Point2D.translater(pl.lesPts[i],vecteur)
  finpour
fin

Remarque(s) :

— Il est à noter que cette dernière procédure aurait pu être écrite en utilisant le principe d'encapsulation :
  procédure translater **(E/S** pl : Polyligne,**E** vecteur : Point2D**)**

Déclaration i : **Naturel**

debut
pour i ←1 **à** nbPoints(pl) **faire**
temp ← iemePoint(pl,i)
Point2D.translater(temp,vecteur)
supprimerPoint(pl,i)
ajouterPoint(pl,temp,i)
finpour
fin
Mais cela met en avant le fait qu'il manque une opération *remplacer* non obligatoire mais qui facilite la vie des utilisateurs du TAD.

6.3 Utilisation d'une polyligne

Dans cette partie, nous sommes utilisateur du type `Polyligne` et nous respectons le principe d'encapsulation.

6.3.1 Point à l'intérieur

Nous supposons posséder la fonction suivante qui permet de calculer l'angle orienté en degré formé par les segments $(ptCentre, pt1)$ et $(ptCentre, pt2)$:

— **fonction** angle (ptCentre,pt1,pt2 : Point2D) **: Reel**

⌊**précondition(s)** pt1≠ptCentre et pt2≠ptCentre

Il est possible de savoir si un point pt est à l'intérieur ou à l'extérieur d'une polyligne fermée en calculant la somme des angles orientés formés par les segments issus de pt vers les points consécutifs de la polyligne. En effet si cette somme en valeur absolue est égale à 360° alors le point pt est à l'intérieur de la polyligne, sinon il est à l'extérieur.

Par exemple, sur la figure 6.3, on peut savoir algorithmiquement que pt est à l'intérieur de la polyligne car $|\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \alpha_5| = 360$.

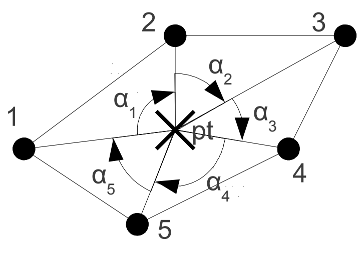


FIGURE 6.3 – Point à l'intérieur d'une polyligne

Proposez le code de la fonction suivante :`estALInterieur`
fonction estALinterieur (p : Polyligne, pt : Point2D) **: Booleen**

⌊**précondition(s)** estFerme(p) et non estSurLaFrontiere(pt,p)

Correction proposée:

fonction estALinterieur (p : Polyligne, pt : Point2D) **: Booleen**

⌊**précondition(s)** estFerme(p) et non estSurLaFrontiere(pt,p)

Déclaration i : **Naturel**
sommeAngle : **Reel**

debut
sommeAngle ← 0
pour i ←1 **à** nbPoints(p)-1 **faire**
sommeAngle ← sommeAngle+angle(pt,iemePoint(p,i),iemePoint(p,i+1))
finpour
sommeAngle ← sommeAngle+angle(pt,iemePoint(p,nbPoints(p)),iemePoint(p,1))
retourner sommeAngle=360 ou sommeAngle=-360
fin

6.3.2 Surface d'une polyligne par la méthode de monté-carlo

Une des façons d'approximer la surface d'une polyligne est d'utiliser la méthode de Monté-Carlo. Le principe de cette méthode est de ≪ calculer une valeur numérique en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes ≫ (Wikipédia). Dans le cas du calcul d'une surface, il suffit de tirer au hasard des points qui sont à l'intérieur du plus petit rectangle contenant la polyligne. La surface S de la polyligne pourra alors être approximée par la formule suivante :

$$S \approx \text{SurfaceDuRectangle} \times \frac{\text{Nb points dans la polyligne}}{\text{Nb points total}}$$

Par exemple, sur la figure 6.4, en supposant que le rectangle fasse 3 cm de hauteur et $4,25$ cm de largeur, et qu'il y a 28 points sur 39 qui sont à l'intérieur de la polyligne, sa surface S peut être approximée par :

$$S \approx 3 \times 4,25 \times \frac{28}{38} = 9,39 \text{ cm}^2$$

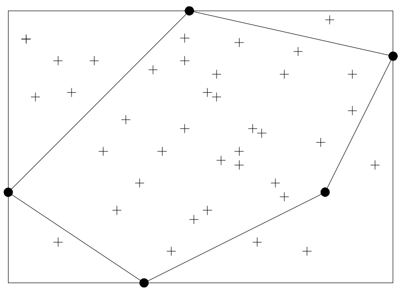


FIGURE 6.4 – Calcul de la surface d'une polyligne par la méthode de Monté-Carlo

On suppose posséder la procédure suivante qui permet d'obtenir un réel aléatoire entre une borne minimum et une borne maximum :

— **procédure** reelAleatoire (**E** borneMin,bornneMax : **Reel**, **S** leReel : **Reel)**

1. Proposez l'analyse descendante pour le calcul d'une surface d'une polyligne à l'aide de la méthode de Monté-Carlo.
   Correction proposée:

surfacePolyligne Polyligne $\times$ **Naturel** $\to$ **Reel**

- **rectangleEnglobant** Polyligne $\to$ Point2D $\times$ Point2D
- **surfaceRectangle** Point2D $\times$ Point2D $\to$ **Reel**
- **pointAleatoireDansRectangle** Point2D $\times$ Point2D $\to$ Point2D

2. Donnez les signatures des procédures et fonctions de votre analyse descendante.
Correction proposée:

— **fonction** surfacePolyligne **(**p : Polyligne, nbPoints : **Naturel) : Reel**
— **fonction** rectangleEnglobant **(**p : Polyligne**) :** Point2D, Point2D
— **fonction** surfaceRectangle **(**ptBasGauche,ptHautDroit : Point2D**) : Reel**
— **procédure** pointAleatoireDansRectangle **(E** ptBasGauche,ptHautDroit : Point2D, **S** lePoint : Point2D**)**

3. Donnez l'algorithme de l'opération principale (au sommet de votre analyse descendante).
Correction proposée:

fonction surfacePolyligne **(**p : Polyligne, nbPoints : **NaturelNonNul) : Reel**
⌊**précondition(s)** estFerme(p) et not tousLesPointsAlignes(p)

Déclaration ptBasGauche, ptHautDroit, pt : Point2D
i, nbDans, nbPointsTotal : **Naturel**

debut
ptBasGauche,ptHautDroit ← rectangleEnglobant(p)
surface ← surfaceRectangle(ptBasGauche,ptHautDroit)
nbDans ← 0
nbPointsTotal ← 0
tant que nbPointsTotal≠nbPoints **faire**
pointAleatoireDansRectangle(ptBasGauche,ptHautDroit, pt)
si non estSurLaFrontiere(p, pt) **alors**
nbPointsTotal ← nbPointsTotal+1
si estALinterieur(p,pt) **alors**
nbDans ← nbDans+1
finsi
finsi
fintantque
retourner surface*nbDans/nbPointsTotal
fin

Chapitre 7

Tri par tas

Attendus d'apprentissages disciplinaires évalués

- AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple
- CD102 : Calculer une complexité dans le pire et le meilleur des cas
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs
- CD501 : Comprendre les algorithmes des différents tris et leurs complexités

7.1 Qu'est ce qu'un tas ?

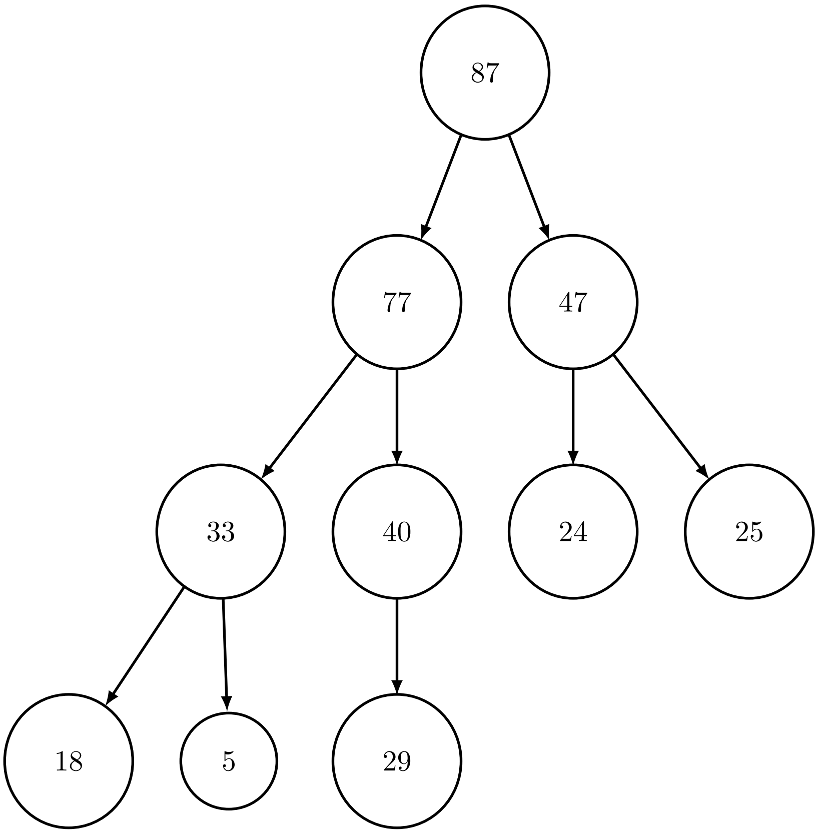
Un tas est un arbre binaire particulier : la valeur de chaque noeud est supérieure aux valeurs contenues dans ses sous-arbres et l'arbre est rempli par niveau (de gauche à droite), un nouveau niveau n'étant commencé que lorsque le précédent est complet.

Un tas peut être représenté l'aide d'un tableau t de telle sorte que les fils gauche et droit de $t[i]$ sont respectivement $t[2*i]$ et $t[2*i+1]$.

Dessinez l'arbre binaire représenté par le tableau t suivant :

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
t	87	77	47	33	40	24	25	18	5	29

Correction proposée:



7.2 Fonction *estUnTas*

Donnez l'algorithme récursif de la fonction suivante qui permet de savoir si un tableau t de n éléments significatifs représente un tas à partir de la racine de position i :

— **fonction** estUnTas **(**t : **Tableau[**1..MAX**] d'Entier**, i,n : **Naturel) : Booleen**

 ⌊**précondition(s)** i≤n

Correction proposée:

fonction estUnTas **(**t : **Tableau[**1..MAX**] d'Entier**, i,n : **Naturel) : Booleen**

 ⌊**précondition(s)** i≤n

debut
  si 2*i > n **alors**
    retourner VRAI
  sinon
    si 2*i+1 > n **alors**
      retourner t[i]≥t[2*i]
    sinon
      si t[i]≥max(t[2*i],t[2*i+1]) **alors**
        retourner estUnTas(t,2*i,n) et estUnTas(t,2*i+1,n)
      sinon
        retourner FAUX
      finsi
    finsi
  finsi

fin

7.3 Procédure *faireDescendre*

À l'issue de l'appel à cette procédure *faireDescendre*, l'arbre (représenté par un tableau) dont la racine est en position i sera un tas. On présuppose que les deux arbres dont les racines sont positionnées en $2i$ et $2i+1$ sont des tas.

La signature de cette procédure est :

— **procédure** faireDescendre **(E/S** t : **Tableau[**1..MAX**] d'Entier**,**E** i,n : **Naturel)**

1. En supposant que la première valeur du tableau t de la partie 7.1 ne soit pas 87 mais 30. Donnez les valeurs de t après l'appel `faireDescendre(t,1,10)`.
2. Proposez l'algorithme de la procédure *faireDescendre*.
3. Donnez la complexité dans le pire des cas de votre algorithme. Justifiez.

Correction proposée:

1. On obtient alors le tableau :

77	40	47	33	30	24	25	18	5	29

2. L'algorithme est :

fonction indiceDuMax **(**t : **Tableau[**1..MAX**] d'Entier**, n,i,j : **Naturel) : Naturel**

⌊**précondition(s)** i≤n et i≤j

debut
  si j≤n **alors**
    si t[i]>t[j] **alors**
      retourner i
    sinon
      retourner j
    finsi
  sinon
    retourner i
  finsi
fin

Version itérative

procédure faireDescendre **(E/S** t : **Tableau[**1..MAX**] d'Entier**,**E** i,n : **Naturel)**

Déclaration elementBienPositionne :**Booleen**
posDuMax :**Naturel**

debut
  elementBienPositionne ← FAUX
  tant que non elementBienPositionne **faire**
    si 2*i≤n **alors**
      // dans ce cas i ne référence pas une feuille
      posDuMax ← indiceDuMax(t,n,2*i,2*i+1)
      si t[i]<t[posDuMax] **alors**
        echanger(t[i],t[posDuMax])
        i ← posDuMax
      sinon
        elementBienPositionne ← VRAI

finsi
sinon
elementBienPositionne ← VRAI
finsi
fintantque
fin

Version récursive

procédure faireDescendre **(E/S** t : **Tableau[**1..MAX**] d'Entier**,**E** i,n : **Naturel)**

Déclaration posDuMax :**Naturel**

debut
si 2*i≤n **alors**
posDuMax ← indiceDuMax(t,n,2*i,2*i+1)
si t[i]<t[posDuMax] **alors**
echanger(t[i],t[posDuMax])
faireDescendre(t,posDuMax,n)
finsi
finsi
fin

3. À chaque itération l'indice de i est multiplié par 2 (à un près) jusqu'à ce que i soit plus grand que n, la complexité est donc en $log_2(n)$.

7.4 Procédure *tamiser*

L'objectif de cette procédure est de transformer un tableau de n éléments significatifs quelconque en un tas. Pour ce faire on part du milieu du tableau en remontant jusqu'au premier élément du tableau pour qu'à l'issue de chaque itération l'arbre représenté par le tableau dont la racine est à la position i soit un tas.

1. Soit le tableau t suivant :

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
t	33	77	25	18	40	24	47	87	5	29

Donnez les valeurs de ce tableau à l'issue de chaque itération.

2. Proposez l'algorithme de la procédure *tamiser*.
3. Donnez la complexité dans le pire des cas de votre algorithme. Justifiez.

Correction proposée:

1. À l'issue de chaque itération on a :

i=5	33	77	25	18	40	24	47	87	5	29
i=4	33	77	25	87	40	24	47	18	5	29
i=3	33	77	47	87	40	24	25	18	5	29
i=2	33	87	47	77	40	24	25	18	5	29
i=1	87	77	47	33	40	24	25	18	5	29

2. On obtient l'algorithme :

procédure tamiser **(E/S** t : **Tableau[**1..MAX**] d'Entier**,**E** n : **Naturel)**

Déclaration i : **Naturel**

debut
pour i ←n div 2 **à** 1 **pas de** -1 **faire**

faireDescendre(t,i,n)
finpour
fin

3. L'algorithme est une boucle déterministe dont l'une des bornes est fonction de la taille du problème n, la complexité est donc n fois la complexité du corps de cette itération. On a donc une complexité dans le pire des cas qui de $n * log_2(n)$.

7.5 Procédure *trierParTas*

Le principe du tri par tas est simple. Après avoir transformé le tableau t composé de n éléments significatifs en un tas, cet algorithme est composé d'itérations i (allant de n jusqu'à 2) qui :

— échange $t[1]$ et $t[i]$;

— s'assure que le tableau de $i - 1$ éléments significatifs soit un tas.

Voici les différentes étapes de cet algorithme une fois que le tableau t de la partie 7.4 ait été transformé en tas (tableau de la partie 7.1) :

1	77	40	47	33	29	24	25	18	5	87
2	47	40	25	33	29	24	5	18	77	87
3	40	33	25	18	29	24	5	47	77	87
4	33	29	25	18	5	24	40	47	77	87
5	29	24	25	18	5	33	40	47	77	87
6	25	24	5	18	29	33	40	47	77	87
7	24	18	5	25	29	33	40	47	77	87
8	18	5	24	25	29	33	40	47	77	87
9	5	18	24	25	29	33	40	47	77	87

1. Dessinez l'analyse descendante *a posteriori* de ce problème.
2. Proposez l'algorithme de la procédure *trierParTas*.
3. Donnez la complexité dans le pire des cas de votre algorithme. Justifiez.

Correction proposée:

1. Analyse descendante :

   trierParTas Tableau[1..MAX**] d'Entier** $\times$ **Naturel** $\rightarrow$ **Tableau[**1..MAX**] d'Entier**

   tamiser Tableau[1..MAX**] d'Entier** $\times$ **Naturel** $\rightarrow$ **Tableau[**1..MAX**] d'Entier**

   faireDescendre Tableau[1..MAX**] d'Entier** $\times$ **Naturel** $\times$ **Naturel** $\rightarrow$ **Tableau[**1..MAX**] d'Entier**

2. L'algorithme

   procédure trierParTas **(E/S** t : **Tableau[**1..MAX**] d'Entier**,**E** n : **Naturel)**
   debut
   tamiser(t,n)
   pour i ←n **à** 2 **pas de** -1 **faire**
   echanger(t[1],t[i])
   faireDescendre(t,1,i-1)
   finpour
   fin

3. Complexité : l'algorithme est composé d'un schéma séquentiel à deux instructions : tamiser est en $n * log_2(n)$ et la deuxième instruction (le pour) est aussi en $n * log_2(n)$. La complexité dans le pire des cas est en $n * log_2(n)$.

Chapitre 8

Sudoku

Attendus d'apprentissages disciplinaires évalués

- AN201 : Identifier les dépendances d'un TAD
- AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
- AN205 : Formaliser les préconditions d'une opération d'un TAD
- AN205 : Formaliser les préconditions d'une opération d'un TAD
- AN301 : Lister les collections usuelles
- CP003 : Choisir entre une fonction et une procédure
- CD003 : Utiliser le principe d'encapsulation
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs

Le jeu du Sudoku est composé d'une grille carrée de 9 cases de côté. Ce jeu consiste ≪ à compléter toute la grille avec des chiffres allant de 1 à 9. Chaque chiffre ne doit être utilisé qu'une seule fois par ligne, par colonne et par carré de neuf cases ≫ [1].

On suppose que l'on numérote les lignes, les colonnes et les carrés d'une grille de Sudoku de 1 à 9.

La grille présentée par la figure 8.1 présente une grille de Sudoku à compléter.

Soit les TAD Coordonnee et GrilleSudoku suivants :

Nom: Coordonnee
Utilise: **Naturel**
Opérations:
- coordonnee: $1..9 \times 1..9 \rightarrow$ Coordonnee
- obtenirLigne: Coordonnee $\rightarrow 1..9$
- obtenirColonne: Coordonnee $\rightarrow 1..9$
- obtenirCarre: Coordonnee $\rightarrow 1..9$

Axiomes:
- obtenirColonne(coordonnee(c,l))=c
- obtenirLigne(coordonnee(c,l))=l
- obtenirCarre(c)=3*((obtenirLigne(c)-1) div 3)+((obtenirColonne(c)-1) div3)+1

Nom: GrilleSudoku
Utilise: **Naturel**, Coordonnee, **Booleen**
Opérations:
- grilleSudoku: $\rightarrow$ GrilleSudoku
- caseVide: GrilleSudoku $\times$ Coordonnee $\rightarrow$ **Booleen**

1. Définition donnée par le journal le Monde.

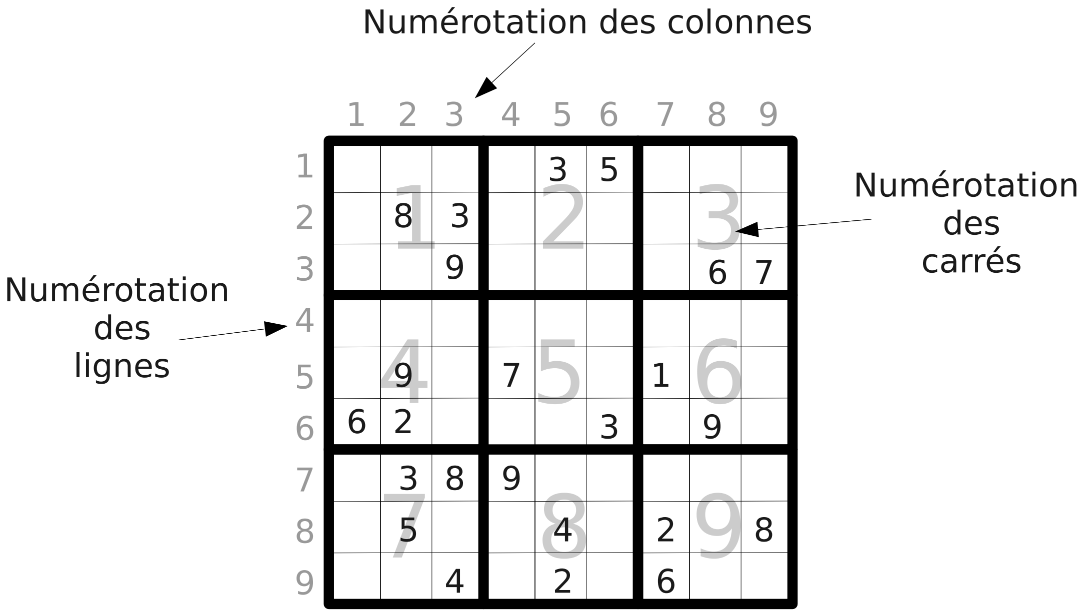


FIGURE 8.1 – Exemple de grille de Sudoku

	obtenirChiffre:	GrilleSudoku $\times$ Coordonnee $\nrightarrow$ 1..9
	fixerChiffre:	GrilleSudoku $\times$ Coordonnee $\times$ 1..9 $\nrightarrow$ GrilleSudoku
	viderCase:	GrilleSudoku $\times$ Coordonnee $\nrightarrow$ GrilleSudoku
Sémantiques:	grilleSudoku:	permet de créer une grille de Sudoku vide
	caseVide:	permet de savoir si une case d'une grille de Sudoku vide
	obtenirChiffre:	permet d'obtenir le chiffre d'une case non vide
	fixerChiffre:	permet de fixer un chiffre d'une case vide
	viderCase:	permet d'enlever le chiffre d'une case non vide
Préconditions:	obtenirChiffre(g,c):	non caseVide(g,c)
	fixerChiffre(g,c,v):	caseVide(g,c)
	viderCase(g,c):	non caseVide(g,c)

8.1 Conception préliminaire

Donnez la signature des fonctions et procédures correspondant aux deux TAD précédents.

Correction proposée:

— **fonction** coordonnee (c,l : 1..9) **:** Coordonnee

— **fonction** obtenirLigne (c : Coordonnee) **:** 1..9

— **fonction** obtenirColonne (c : Coordonnee) **:** 1..9

— **fonction** obtenirCarre (c : Coordonnee) **:** 1..9

— **fonction** grilleSudoku () **:** GrilleSudoku

— **fonction** caseVide (g : GrilleSudoku, c : Coordonnee) **: Booleen**

— **fonction** obtenirChiffre (g : GrilleSudoku, c : Coordonnee) **:** 1..9

⌊**précondition(s)** non caseVide(g,c)

— **procédure** fixerChiffre **(E/S** g : GrilleSudoku,**E** c : Coordonnee, v : 1..9**)**
  ⌊**précondition(s)** caseVide(g,c)
— **procédure** viderCase **(E/S** g : GrilleSudoku,**E** c : Coordonnee**)**
  ⌊**précondition(s)** non caseVide(g,c)

8.2 Conception détaillée

On se propose de concevoir le TAD Coordonnee de la façon suivante :

Type Coordonnee **= Structure**
  ligne : 1..9
  colonne : 1..9
finstructure

Donnez les algorithmes des fonctions correspondant aux opérations de ce TAD.

Correction proposée:

fonction coordonnee (c,l : 1..9) **:** Coordonnee
  Déclaration resultat : Coordonnee
debut
  resultat.colonne ← c
  resultat.ligne ← l
  retourner resultat
fin
fonction obtenirLigne (c : Coordonnee) **:** 1..9
debut
  retourner c.ligne
fin
fonction obtenirColonne (c : Coordonnee) **:** 1..9
debut
  retourner c.colonne
fin
fonction obtenirCarre (c : Coordonnee) **:** 1..9
debut
  retourner 3*((c.ligne-1) div 3)+(c.colonne -1) div 3+1
fin

8.3 Fonctions métiers

On se propose d'écrire des fonctions et procédures permettant de vérifier ou d'aider à la résolution manuelle d'une grille de Sudoku.

1. Donnez l'algorithme de la fonction suivante qui permet de savoir si une grille de Sudoku est totalement remplie (sans vérifier sa validité) :
   — **fonction** estRemplie (g : GrilleSudoku) **: Booleen**
2. On suppose que l'on possède les fonctions suivantes qui permettent d'obtenir l'ensemble des chiffres déjà fixés d'une colonne, d'une ligne ou d'un carré :
   — **fonction** obtenirChiffresDUneLigne (g : GrilleSudoku, ligne : 1..9) **:** Ensemble$<$ 1..9 $>$
   — **fonction** obtenirChiffresDUneColonne (g : GrilleSudoku, colonne : 1..9) **:** Ensemble$<$ 1..9 $>$

— **fonction** obtenirChiffresDUnCarre (g : GrilleSudoku, carre : 1..9) **:** Ensemble$< 1..9 >$

Donnez l'algorithme de la fonction suivante qui permet de savoir si on peut mettre un chiffre dans une case vide sans contredire la règle donnée en introduction :

— **fonction** estChiffreValable (g : GrilleSudoku, chiffre : 1..9, case : Coordonnee) **: Booleen**
⌊**précondition(s)** caseVide(g,case)

3. Donnez l'algorithme la fonction suivante qui donne la liste des solutions possibles pour une case vide :

   — **fonction** obtenirSolutionsPossibles (g : GrilleSudoku, case : Coordonnee) **:** Liste$< 1..9 >$
   ⌊**précondition(s)** caseVide(g,case)

4. Donnez l'algorithme de la fonction suivante qui cherche la solution d'une grille de sudoku g (le booléen indique s'il y a effectivement une solution) :

   — **fonction** chercherSolution (g : GrilleSudoku) **:** GrilleSudoku, **Booleen**

Correction proposée:

1.
```
fonction estRemplie (g : GrilleSudoku) : Booleen
   Déclaration i,j : 1..9
               c : Coordonnee
               resultat : Booleen
debut
   resultat ← VRAI
   finDeBoucle ← FAUX
   i ← 1
   j ← 1
   tant que resultat et non finDeBoucle faire
      c ← coordonnee(i,j)
      si estVide(g,c) alors
         resultat ← FAUX
      sinon
         si i=9 alors
            si j=9 alors
               finDeBoucle ← VRAI
            sinon
               i ← 1
               j ← j+1
            finsi
         sinon
            i ← i+1
         finsi
      finsi
   fintantque
   retourner resultat
fin
```

2.
```
fonction estChiffreValable (g : GrilleSudoku, chiffre : 1..9, case : Coordonnee) : Booleen
   ⌊précondition(s) caseVide(g,case)
```

```
    Déclaration  e1,e2,e3 : Ensemble< 1..9 >
debut
    e1 ← obtenirChiffresDUneLigne(g,obtenirLigne(c))
    e2 ← obtenirChiffresDUneColonne(g,obtenirColonne(c))
    e3 ← obtenirChiffresDUnCarre(g,obtenirCarre(c))
    retourner non estPresent(e1,chiffre) et non estPresent(e2,chiffre) et non estPresent(e3,chiffre)
fin
```

3.

```
fonction obtenirSolutionsPossibles (g : GrilleSudoku, case : Coordonnee) : Liste< 1..9 >
    ⌊précondition(s)  caseVide(g,case)
    Déclaration  resultat : Liste< 1..9 >
                 i : 1..9
debut
    resultat ← liste()
    pour i ←1 à 9 faire
        si estChiffreValable(g,i,case) alors
            inserer(resultat,1,i)
        finsi
    finpour
    retourner resultat
fin
```

4.

```
fonction premiereCaseVide (g) : Coordonnee
    ⌊précondition(s)  non estRemplie(g)
    Déclaration  i,j : 1..9
                 c : Coordonnee
debut
    trouve ← FAUX
    i ← 1
    tant que non trouve et i≤9 faire
        j ← 1
        tant que non trouve et i≤9 faire
            c ← coordonnee(i,j)
            si caseVide(g,c) alors
                trouve ← VRAI
            finsi
            j ← j+1
        fintantque
        i ← i+1
    fintantque
    retourner c
fin
fonction chercherSolution (g : GrilleSudoku) : GrilleSudoku, Booleen
    Déclaration  temp, sol : GrilleSudoku
                 k : 1..9
                 l : Liste<1..9>
                 trouve : Booleen
```

```
debut
    si estRemplie(g) alors
        retourner g, VRAI
    sinon
        trouve ← FAUX
        c ← premiereCaseVide(g)
        l ← obtenirSolutionsPossibles(g,c)
        k ← 1
        tant que non trouve et k≤longueur(l) faire
            temp ← g
            fixerChiffre(temp,c,obtenirElement(l,k))
            sol, trouve ← chercherSolution(temp)
            k ← k+1
        fintantque
        retourner sol, trouve
    finsi
fin
```

Chapitre 9

Liste

Attendus d'apprentissages disciplinaires évalués

— AN301 : Lister les collections usuelles
— CP003 : Choisir entre une fonction et une procédure
— CP004 : Concevoir une signature (préconditions incluses)
— CP005 : Choisir un passage de paramètre (E, S, E/S)
— CD003 : Utiliser le principe d'encapsulation
— CD201 : Identifier et résoudre le problème des cas non récursifs
— CD202 : Identifier et résoudre le problème des cas récursifs
— CD401 : Concevoir et utiliser des listes chaînées
— CD901 : Concevoir un type de données adapté à la situation en terme d'espace mémoire et d'efficacité

9.1 SDD ListeChainee

9.1.1 Type et signatures de fonction et procédure

Après avoir rappelé le SDD ListeChainee dans le paradigme de la programmation structurée, donnez les signatures des fonctions et procédures permettant de l'utiliser.

Correction proposée:
Voir le cours

9.1.2 Utilisation

1. Écrire une fonction booléenne itérative, `estPresent`, qui permet de savoir si un élément est présent dans une liste chaînée.
2. Écrire une fonction booléenne récursive, `estPresent`, qui permet de savoir si un élément est présent dans une liste chaînée.
3. Écrire une procédure récursive, `concatener`, qui concatène deux listes chaînées.
4. Écrire une procédure récursive, `inverser`, qui permet d'inverser une liste chaînée.
5. Écrire une procédure itérative, `inverser`, qui permet d'inverser une liste chaînée.

Correction proposée:

1. fonction estPresent itérative

   fonction estPresent **(**l : ListeChainee, cherche : Element**) : Booleen**

   ```
   Déclaration resultat : Booleen
               liste : ListeChainee
   debut
     resultat ← FAUX
     liste ← l
     tant que non estVide(liste) et non resultat faire
       si obtenirElement(liste) = cherche alors
         resultat ← VRAI
       sinon
         liste ← obtenirListeSuivante(liste)
       finsi
     fintantque
     retourner resultat
   fin
   ```

2. est présent récursif

   fonction estPresent **(**liste : ListeChainee, cherche : Element**) : Booleen**

   ```
   debut
     si estVide(liste) alors
       retourner FAUX
     sinon
       si obtenirElement(liste) = cherche alors
         retourner VRAI
       sinon
         retourner estPresent(obtenirListeSuivante(liste),cherche)
       finsi
     finsi
   fin
   ```

3. concaténation

   procédure concatener **(E/S** l1 : ListeChainee,**E** l2 : ListeChainee**)**

   ```
   Déclaration temp : ListeChainee
   debut
     si estVide(l1) alors
       l1 ← l2
     sinon
       si non estVide(l2) alors
         temp ← obtenirListeSuivante(l1)
         concatener(temp,l2)
         si estVide(obtenirListeSuivante(l1)) alors
           fixerListeSuivante(l1, temp)
         finsi
       finsi
     finsi
   fin
   ```

4. inverser (récursif)

procédure inverser **(E/S** l : ListeChainee**)**

  Déclaration temp : ListeChainee

debut
  si non estVide(l) **alors**
    temp ← obtenirListeSuivante(l)
    inverser(temp)
    fixerListeSuivante(l,listeChainee())
    concatener(temp,l)
    l ← temp
  finsi
fin

5. inverser (itératif)

procédure inverser **(E/S** l : ListeChainee**)**

  Déclaration resultat,temp : ListeChainee

debut
  resultat ← listeVide()
  tant que non estVide(l) **faire**
    temp ← l
    l ← obtenirListeSuivante(l)
    fixerListeSuivante(temp,resultat)
    resultat ← temp
  fintantque
  l ← resultat
fin

9.2 Conception détaillée d'une liste ordonnée d'entiers à l'aide d'une liste chainée

Cet exercice propose de concevoir le type ListeOrdonneeDEntiers (ou LODE) avec le SDD ListeChainee de l'exercice précédent.

1. Proposez une conception détaillée du type ListeOrdonneeDEntiers
2. Ecrire les fonctions/procédures creationListeOrdonneeDEntiers, inserer, supprimer un élément (le premier, et que l'on sait présent), obtenirIemeElement à la ième position et longueur proposées par ce type

Correction proposée:

1.

Type ListeOrdonneeDEntiers **= Structure**
  entiers : ListeChainee<**Entier**>
  nbEntiers : **Naturel**
finstructure

2.

fonction longueur (l :ListeOrdonneeDEntiers) **: Naturel**
debut
  retourner l.nbEntiers

```
fin

fonction obtenirEntiers (l :ListeOrdonneeDEntiers) : ListeChainee<Entier>
debut
    retourner l.entiers
fin

procédure fixerNbEntiers (E/S l : ListeOrdonneeDEntiers, E valeur : Naturel)
debut
    l.nbEntiers ← valeur
fin

procédure fixerEntiers (E/S l : ListeOrdonneeDEntiers, E liste : ListeChainee<Entier>)
debut
    l.entiers ← liste
fin

fonction listeOrdonneeDEntiers () : ListeOrdonneeDEntiers
    Déclaration   resultat : ListeOrdonneeDEntiers
debut
    fixerEntiers(resultat, listeChainee())
    fixerNbEntiers(resultat, 0)
    retourner resultat
fin

// Version itérative

procédure insererDansListeChainee (E/S l : ListeChainee <Entier>, E element : Entier)
    Déclaration   parcours, nouveau, temporaire : ListeChainee<Entier>
debut
    si estVide(l) alors
        ajouter(l,element)
    sinon
        si obtenirElement(l) > element alors
            ajouter(l,element)
        sinon
            g ← l
            d ← obtenirListeSuivante(g)
            tant que non estVide(d) et obtenirElement(d) < element faire
                g ← d
                d ← obtenirListeSuivante(g)
            fintantque
            ajouter(d,element)
            fixerListeSuivante(g,d)
        finsi
    finsi
fin
```

// Version récursive

```
procédure insererDansListeChainee (E/S l : ListeChainee <Entier>, E element : Entier)
   Déclaration temp : ListeChainee<Entier>
debut
   si estVide(l) alors
      ajouter(l, element)
   sinon
      si obtenirElement(l) > element alors
         ajouter(l,element)
      sinon
         temp ← obtenirListeSuivante(l)
         insererDansListeChainee(temp, element)
         fixerListeSuivante(l, temp)
      finsi
   finsi
fin

procédure inserer (E/S l : ListeOrdonneeDEntiers, E element : Entier)
   Déclaration temp : ListeChainee<Entier>
debut
   temp ← obtenirEntiers(l)
   insererDansListeChainee(temp, element)
   fixerEntiers(l, temp)
   fixerNbEntiers (l, longueur(l) + 1)
fin

procédure supprimerDansListeChainee (E/S l : ListeChainee, E e : Entier)
   ⌊précondition(s) estPresent(l, e)
   Déclaration temp : ListeChainee <Entier>
debut
   si obtenirElement(l) = e alors
      supprimerTete(l)
   sinon
      temp ← obtenirListeSuivante(l,e)
      supprimerDansListeChainee(temp,e)
      fixerListeSuivante(l, temp)
   finsi
fin

procédure supprimer (E/S l : ListeOrdonneeDEntiers, E element : Entier)
   ⌊précondition(s) estPresent(l, e)
   Déclaration temp : ListeChainee <Entier>
debut
   temp ← obtenirEntiers(l)
   supprimerDansListeChainee(temp,element)
```

```
        fixerEntiers(l, temp)
        fixerNbEntiers(l, longueur(l) - 1)
    fin

    fonction obtenirIemeElement (liste : ListeOrdonneeDEntiers, i : NaturelNonNul) : Entier
        ⌊précondition(s)  i ≤ longueur(liste)

        Déclaration  l1 : ListeOrdonneeDEntiers
                     j : Naturel
    debut
        l1 ← obtenirEntiers(liste)
        pour j ←2 à i faire
            l1 ← obtenirListeSuivante(l1)
        finpour
        retourner obtenirElement(l1)
    fin

    fonction longueur (liste : ListeOrdonneeDEntiers) : Naturel
    debut
        retourner longueur(liste)
    fin
```

3.

9.3 Utilisation : Liste ordonnée d'entiers

Écrire une fonction, fusionner, qui permet de fusionner deux listes ordonnées

Correction proposée:

```
procédure insererUneListeOrdonneeDEntiers (E/S dans : ListeOrdonneeDEntiers, E liste : ListeOrdonnee-
DEntiers)
    Déclaration  i : Naturel
debut
    pour i ←1 à longueur(liste) faire
        inserer(dans, obtenirIemeElement(liste, i))
    finpour
fin

fonction fusionner (l1,l2 : ListeOrdonneeDEntiers) : ListeOrdonneeDEntiers
    Déclaration  resultat : ListeOrdonneeDEntiers
debut
    resultat ← listeOrdonneeDEntiers()
    insererUneListeOrdonnéeDEntiers(resultat, l1)
    insererUneListeOrdonnéeDEntiers(resultat, l2)
    retourner resultat
fin
```

Chapitre 10

Arbre Binaire de Recherche (ABR)

Attendus d'apprentissages disciplinaires évalués

— CD201 : Identifier et résoudre le problème des cas non récursifs
— CD202 : Identifier et résoudre le problème des cas récursifs
— CD403 : Concevoir et utiliser des arbres (binaires, n-aires)
— CD601 : Concevoir des collections à l'aide de SDD
— CD602 : Comprendre les algorithmes d'insertion et de suppression (naïfs et AVL) dans un arbre binaire de recherche
— CD901 : Concevoir un type de données adapté à la situation en terme d'espace mémoire et d'efficacité

10.1 Conception préliminaire et utilisation d'un ABR

Pour rappel, le TAD ABR modélisant un Arbre Binaire de Recherche est défini de la façon suivante :

Nom: ABR (ArbreBinaireDeRecherche)
Paramètre: Element
Utilise: **Booleen**
Opérations:
aBR: $\rightarrow$ ABR
estVide: ABR $\rightarrow$ **Booleen**
insérer: ABR $\times$ Element $\rightarrow$ ABR
supprimer: ABR $\times$ Element $\rightarrow$ ABR
estPresent: ABR $\times$ Element $\rightarrow$ **Booleen**
obtenirElement: ABR $\nrightarrow$ Element
obtenirFilsGauche: ABR $\nrightarrow$ ABR
obtenirFilsDroit: ABR $\nrightarrow$ ABR

Axiomes:
- estVide(aBR())
- non estVide(insérer(e,a))
- obtenirElement(insérer(e,aBR()))=e
- obtenirFilsGauche(insérer(e,a))=insérer(e,obtenirFilsGauche(a) et obtenirElement(a)$>e$
- obtenirFilsDroit(insérer(e,a))=insérer(e,obtenirFilsDroit(a) et obtenirElement(a)$<e$

...

Préconditions: obtenirElement(a): $non(estVide(a))$
obtenirFilsGauche(a): $non(estVide(a))$
obtenirFilsDroit(a): $non(estVide(a))$

1. Donner les signatures des fonctions et procédures d'un ABR.
2. Écrire une procédure récursive, `afficherEnOrdreCroissant`, qui affiche, en ordre croissant, tous les éléments d'un ABR.
3. Écrire une procédure récursive, `afficherEnOrdreDecroissant`, qui affiche, en ordre décroissant, tous les éléments d'un ABR.
4. Écrire une fonction récursive, `hauteur`, qui calcule la hauteur d'un ABR (-1 si l'arbre est vide, 0 s'il n'y a qu'un seul élément).
5. Écrire une fonction récursive, `nbElements`, qui calcule le nombre d'éléments d'un arbre.

Correction proposée:

1. *Voir le cours*
2.
```
procédure afficherEnOrdreCroissant (E a : ABR)
debut
    si non estVide(a) alors
        afficherEnOrdreCroissant(obtenirFilsGauche(a))
        ecrire(obtenirElement(a))
        afficherEnOrdreCroissant(obtenirFilsDroit(a))
    finsi
fin
```
3.
```
procédure afficherEnOrdreDecroissant (E a : ABR)
debut
    si non estVide(a) alors
        afficherEnOrdreDecroissant(obtenirFilsDroit(a))
        ecrire(obtenirElement(a))
        afficherEnOrdreDecroissant(obtenirFilsGauche(a))
    finsi
fin
```
4.
```
fonction maximum (a, b : Element) : Element
debut
    si a > b alors
        retourner a
    sinon
        retourner b
    finsi
fin

fonction hauteur (a : ABR) : Entier
debut
    si estVide(a) alors
        retourner -1
```

```
    sinon
        retourner 1+maximum(hauteur(obtenirFilsGauche(a)),hauteur(obtenirFilsDroit(a)))
    finsi
fin
```

5.

```
fonction nbElements (a : ABR) : Naturel
debut
    si estVide(a) alors
        retourner 0
    sinon
        retourner 1+nbElements(obtenirFilsGauche(a))+nbElements(obtenirFilsDroit(a))
    finsi
fin
```

10.2 Une conception détaillée : ABR

Nous allons concevoir le type ABR à l'aide du SDD ArbreBinaire

1. Rappeler le SDD ArbreBinaire (type et signatures des fonctions et procédures)
2. Proposer une implantation du type ABR
3. Expliciter la fonction booléenne : `estPresent`.
4. Expliciter la procédure d'insertion : `inserer`.
5. Expliciter la procédure de suppression : `supprimer`.

Correction proposée:

1.

```
Type ArbreBinaire = ^ Noeud
Type Noeud = Structure
    lElement : Element
    filsGauche : ArbreBinaire
    filsDroit : ArbreBinaire
finstructure

fonction arbreBinaire () : ArbreBinaire
fonction estVide (a : ArbreBinaire) : Booleen
fonction ajouterRacine (fg,fd : ArbreBinaire,e : Element) : ArbreBinaire
fonction obtenirElement (a : ArbreBinaire) : Element
    ⌊précondition(s)  non estVide(a)
fonction obtenirFilsGauche (a : ArbreBinaire) : ArbreBinaire
    ⌊précondition(s)  non estVide(a)
fonction obtenirFilsDroit (a : ArbreBinaire) : ArbreBinaire
    ⌊précondition(s)  non estVide(a)
procédure fixerFilsGauche (E a : ArbreBinaire, ag : ArbreBinaire)
    ⌊précondition(s)  non estVide(a)
procédure fixerFilsDroit (E a : ArbreBinaire, ad : ArbreBinaire)
```

⌊**précondition(s)** non estVide(a)

procédure supprimerRacine **(E/S** a : ArbreBinaire, **S** fg,fd : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a)

procédure supprimer **(E/S** a : ArbreBinaire**)**

2. **Type** ABR **=** ArbreBinaire
3.

```
fonction estPresent (a : ABR, e : Element) : Booleen
    Déclaration temp : ABR
debut
    si estVide(a) alors
        retourner FAUX
    sinon
        si e=obtenirElement(a) alors
            retourner VRAI
        sinon
            si e<obtenirElement(a) alors
                retourner estPresent(obtenirFilsGauche(a),e)
            sinon
                retourner estPresent(obtenirFilsDroit(a),e)
            finsi ‘
        finsi
    finsi
fin
```

4.

```
procédure inserer (E/S a : ABR, E e : Element)
    Déclaration temp : ABR
debut
    si estVide(a) alors
        a ← ajouterRacine(arbreBinaireRecherche(), arbreBinaireRecherche(), e)
    sinon
        si e≤obtenirElementRacine(a) alors
            temp ← obtenirFilsGauche(a)
            inserer(temp, e)
            fixerFilsGauche(a, temp)
        sinon
            temp ← obtenirFilsDroit(a)
            inserer(temp, e)
            fixerFilsDroit(a, temp)
        finsi
    finsi
fin
```

5.

```
procédure supprimer (E/S a : ABR ; E e : Element)
    Déclaration nouveauSommet : Element
                temp,tempG,tempD : ABR
debut
```

```
    si non estVide(a) alors
        si e < obtenirElement(a) alors
            temp ← obtenirFilsGauche(a)
            supprimer(temp,e)
            fixerFilsGauche(a,temp)
        sinon
            si e > obtenirElement(a) alors
                temp ← obtenirFilsDroit(a)
                supprimer(temp,e)
                fixerFilsDroit(a,temp)
            sinon
                si estVide(obtenirFilsGauche(a)) et estVide(obtenirFilsDroit(a)) alors
                    ArbreBinaire.supprimerRacine(a,tempG,tempD)
                sinon
                    si estVide(obtenirFilsGauche(a)) ou estVide(obtenirFilsDroit(a)) alors
                        ArbreBinaire.supprimerRacine(a,tempG,tempD)
                        si estVide(tempG) alors
                            a ← tempD
                        sinon
                            a ← tempG
                        finsi
                    sinon
                        ArbreBinaire.supprimerRacine(a,tempG,tempD)
                        nouveauSommet ← obtenirElement(lePlusGrand(tempG))
                        supprimer(tempG,nouveauSommet)
                        a ← ArbreBinaire.ajouterRacine(nouveauSommet,tempG,tempD)
                    finsi
                finsi
            finsi
        finsi
    finsi
fin
```

Chapitre 11

Arbres AVL

Pour rappel un AVL est un ABR qui conserve l'équilibre entre tous ces fils (à +-1 près) après les opérations d'insertion et de supression.

1. Expliciter les procédures de "simple rotation", `faireSimpleRotationADroite` et `faireSimpleRotationAGauche`, et de "double rotations", `faireDoubleRotationADroite` et `faireDoubleRotationAGauche`.
   Correction proposée:

   procédure faireSimpleRotationADroite **(E/S** a : ABR**)**

   ⌊**précondition(s)** non(estVide(a)) et non(estVide(obtenirFilsGauche(a)))

   Déclaration temp : ABR

   debut
   temp ← obtenirFilsGauche(a)
   fixerFilsGauche(a,obtenirFilsDroit(temp))
   fixerFilsDroit(temp,a)
   a ← temp
   fin
   procédure faireDoubleRotationADroite **(E/S** a : ABR**)**

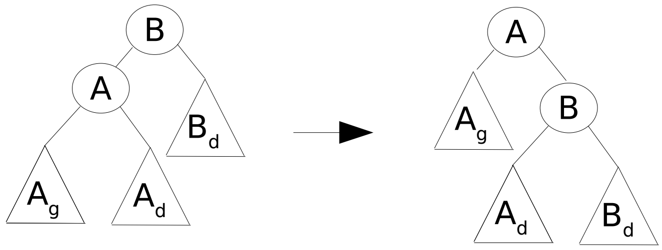
   ⌊**précondition(s)** non(estVide(a)) et non(estVide(obtenirFilsGauche(a)))
   et non(estVide(obtenirFilsDroit(obtenirFilsGauche(a)))

   Déclaration temp : ABR

   debut
   temp ← obtenirFilsGauche(a)
   faireSimpleRotationAGauche(temp)
   fixerFilsGauche(a,temp)
   faireSimpleRotationADroite(a)
   fin

2. Montrer que les simples et doubles rotations conservent la propriété d'un ABR (en considérant que l'arbre ne contient pas de doublons).
   Correction proposée:

   — Simple rotation à droite (même raisonnement à gauche)



Puisque l'arbre de la figure de gauche est un ABR on sait que :

- tous les éléments de A_g sont plus petit que A et B et les éléments de A_d et B_d ;
- tous les éléments de A_d sont plus grands que A et plus petit que B et les éléments B_d ;
- tous les éléments de B_d sont plus grands que B et A ;

Dès lors l'arbre de droite est aussi un ABR.

— Double rotation (à droite ou à gauche) : puisque ces opérations sont une combinaison de simples rotations, l'arbre résultat est donc aussi un ABR.

3. Expliciter la procédure d'équilibrage d'un arbre qui aurait deux sous-arbres équilibrés mais qui pourrait ne pas être équilibré.

Correction proposée:

procédure equilibrer **(E/S** a : ABR**)**
debut
  si hauteur(obtenirFilsGauche(a))>hauteur(obtenirFilsDroit(a))+1 **alors**
    si hauteur(obtenirFilsGauche(obtenirFilsGauche(a))) $\geq$ hauteur(obtenirFilsDroit(obtenirFilsGauche(a)))
    alors
      faireSimpleRotationDroite(a)
    sinon
      faireDoubleRotationDroite(a)
    finsi
  sinon
    si hauteur(obtenirFilsDroit(a))>hauteur(obtenirFilsGauche(a))+1 **alors**
      si hauteur(obtenirFilsGauche(obtenirFilsDroit(a))) $\leq$ hauteur(obtenirFilsDroit(obtenirFilsDroit(a)))
      alors
        faireSimpleRotationGauche(a)
      sinon
        faireDoubleRotationGauche(a)
      finsi
    finsi
  finsi
fin

4. Expliciter la procédure d'insertion : `inserer`.

Correction proposée:

Il suffit de reprendre la procédure d'insertion vu à la section 10.2 et d'appeler `equilibrer` après chaque insertion dans le fils gauche ou droit.

Il est à noter que des conceptions d'AVL ajoutent au sein de chaque nœud la hauteur de l'arbre courant afin de ne pas recalculer cette hauteur qui coûte $O(n)$.

5. Expliciter la procédure de suppression : `supprimer`.

Correction proposée:

Il y a deux façons de résoudre ce problème :

(a) Reprendre l'algorithme de suppression d'un élément dans un ABR vu à la section 10.2 puis rééquilibrer après chaque suppression

(b) Redescendre la valeur à supprimer vers une feuille par une utilisation des simples ou doubles rotations

```
procédure supprimer (E/S a : ABR, E e : Element)
   Déclaration nouveauSommet : Element
               temp,tempG,tempD : ABR
debut
   si non estVide(a) alors
      si e < obtenirElement(a) alors
         temp ← obtenirFilsGauche(a)
         supprimer(temp,e)
         fixerFilsGauche(a,temp)
         equilibrer(a)
      sinon
         si e > obtenirElement(a) alors
            temp ← obtenirFilsDroit(a)
            supprimer(temp,e)
            fixerFilsDroit(a,temp)
            equilibrer(a)
         sinon
            si estVide(obtenirFilsGauche(a)) et estVide(obtenirFilsDroit(a)) alors
               ArbreBinaire.supprimerRacine(a,tempG,tempD)
            sinon
               si hauteur(obtenirFilsGauche(a)) > hauteur(obtenirFilsDroit(a)) alors
                  si hauteur(obtenirFilsGauche(obtenirFilsGauche(a))) > hauteur(obtenirFilsDroit(
                  obtenirFilsGauche(a))) alors
                     faireSimpleRotationADroite(a)
                  sinon
                     faireDoubleRotationADroite(a)
                  finsi
                  temp ← obtenirFilsDroit(a)
                  supprimer(temp,e)
                  fixerFilsDroit(a,temp)
               sinon
                  équivalent mais avec des simple et double rotation à gauche
               finsi
            finsi
         finsi
      finsi
   finsi
fin
```

Chapitre 12

Graphes

Attendus d'apprentissages disciplinaires évalués

- AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple
- AN201 : Identifier les dépendances d'un TAD
- AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
- AN204 : Formaliser des opérations d'un TAD
- AN205 : Formaliser les préconditions d'une opération d'un TAD
- AN206 : Formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD
- CP003 : Choisir entre une fonction et une procédure
- CP004 : Concevoir une signature (préconditions incluses)
- CP005 : Choisir un passage de paramètre (E, S, E/S)
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs
- CD801 : Concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)
- CD804 : Comprendre des algorithmes de recherche du plus court chemin : Dijkstra et A*

12.1 Le labyrinthe

L'objectif de cet exercice est d'étudier le problème du labyrinthe, c'est-à-dire créer un algorithme permettant de trouver le chemin qui mène de l'entrée à la sortie (cf. figure 12.1).

12.1.1 Partie publique

Un labyrinthe est composé de cases. On accède à une case à partir d'une case et d'une direction. Les directions possibles sont `Nord`, `Sud`, `Est` et `Ouest`.

Par exemple, comme le montre la figure 12.2 le labyrinthe précédent peut être considéré comme étant composé de 25 cases. La case numéro 6 est la case d'entrée. La case 20 est la case de sortie. La case 8 est accessible depuis la case 13 avec la direction `Nord`.

Le TAD labyrinthe

Les opérations disponibles sur un labyrinthe sont les suivantes :

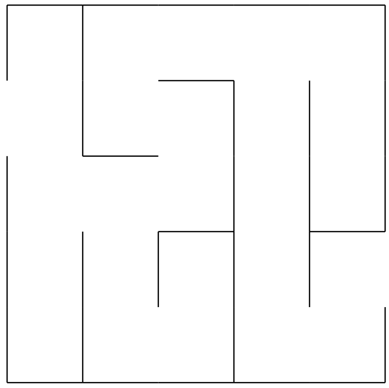


FIGURE 12.1 – Un labyrinthe

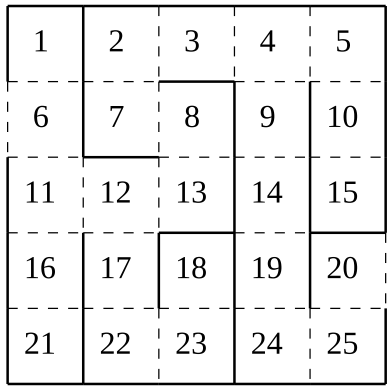


FIGURE 12.2 – Un labyrinthe composé de cases

— créer un labyrinthe,
— obtenir la case d'entrée,
— savoir si une case est la case de sortie,
— obtenir une liste de directions possibles depuis une case donnée,
— obtenir la case accessible depuis une case avec une direction.

1. Donnez le type `Direction`
   Correction proposée:

   Type Direction = {Nord,Sud,Est,Ouest}

2. Donnez le TAD `Labyrinthe`
   Correction proposée:

Nom:	Labyrinthe	
Utilise:	Ensemble, Direction, **NaturelNonNul**	
Opérations:	labyrinthe:	**NaturelNonNul** × **NaturelNonNul** → Labyrinthe
	caseDEntree:	Labyrinthe → **NaturelNonNul**
	estCaseDeSortie:	Labyrinthe × **NaturelNonNul** → **Booleen**
	directionsPossibles:	Labyrinthe × **NaturelNonNul** → Liste<Direction>
	caseDestination:	Labyrinthe × **NaturelNonNul** × Direction ↛ **NaturelNonNul**
Préconditions:	caseDestination(l,c,d):	estPresent(directionsPossibles(l,c),d)

Algorithme du petit-poucet

Une solution pour trouver la sortie est d'utiliser le principe du petit poucet, c'est-à-dire mettre un caillou sur les cases rencontrées.

Pour ne pas modifier le TAD Labyrinthe, plutôt que de marquer une case avec un caillou on peut ajouter une case à un ensemble. Pour vérifier si on a déja rencontré une case, il suffit alors de vérifier si la case est présente dans l'ensemble.

Proposer le corps de la procédure suivante qui permet de trouver le chemin de sortie (s'il existe) à partir d'une case donnée :

procédure calculerCheminDeSortie (**E** l : Labyrinthe, caseCourante : **NaturelNonNul**, **E/S** casesVisitees : Ensemble<**NaturelNonNul**>, **S** permetDAllerJusquALaSortie : **Booleen**, lesDirectionsASuivre : Liste<Direction>)

Correction proposée:

procédure calculerCheminDeSortie (**E** l : Labyrinthe, caseCourante : **NaturelNonNul**, **E/S** casesVisitees : Ensemble<**NaturelNonNul**>, **S** permetDAllerJusquALaSortie : **Booleen**, lesDirectionsASuivre : Liste<Direction>)

```
    Déclaration directions : Liste<Direction>
                i : Naturel
                solutionTrouvee : Booleen
                caseTest : NaturelNonNul

debut
    si estCaseDeSortie(caseCourante) alors
        permetDAllerJusquALaSortie ← VRAI
        lesDirectionsASuivre ← liste()
    sinon
        si non estPresent(casesVisitees,caseCourante) alors
            casesVisitees ← ajouter(casesVisitees,caseCourante)
            directions ← directionsPossibles(l,caseCourante)
            permetDAllerJusquALaSortie ← FAUX
            i ← 1
            tant que i≤longueur(directions) et non permetDAllerJusquALaSortie faire
                caseTest ← caseDestination(l,obtenirElement(directions,i))
                calculerCheminDeSortie(l,caseTest,casesVisitees,permetDAllerJusquALaSortie,
                lesDirectionsASuivre)
                si permetDAllerJusquALaSortie alors
                    ajouter(lesDirectionsASuivre,obtenirElement(directions,i))
                finsi
                i ← i+1
            fintantque
        sinon
            permetDAllerJusquALaSortie ← FAUX
        finsi
    finsi
fin
```

12.1.2 Partie privée

Le graphe

On peut représenter un labyrinthe à l'aide d'un graphe étiqueté et valué. On considère dans ce cas que les valeurs des nœuds du graphe sont les cases du labyrinthe et les arcs étiquetés par les directions.

Dessinez le graphe associé à l'exemple de la figure 12.3.

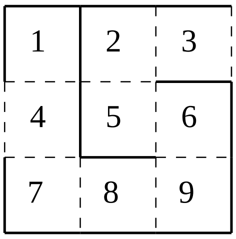
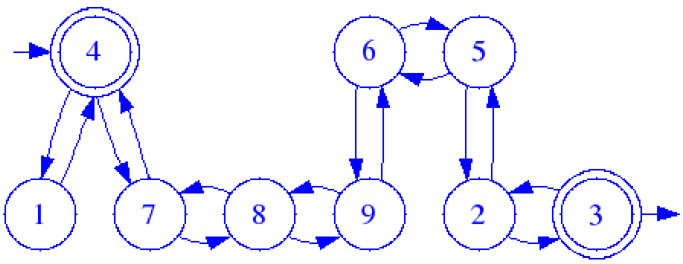


FIGURE 12.3 – Un labytinthe composé de 9 cases

Correction proposée:



Représentation du graphe

Proposez la matrice d'adjascence du graphe précédent.

Correction proposée:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
1				V					
2			V		V				
3		V							
4	V						V		
5		V				V			
6					V				V
7				V				V	
8							V		V
9						V		V	

12.2 Algorithme de Dijkstra

En utilisant l'algorithme de Dijkstra, donnez l'arbre recouvrant pour le graphe présenté par la figure 12.4 depuis le sommet 1 qui permet d'obtenir tous les chemins les plus courts depuis ce sommet.

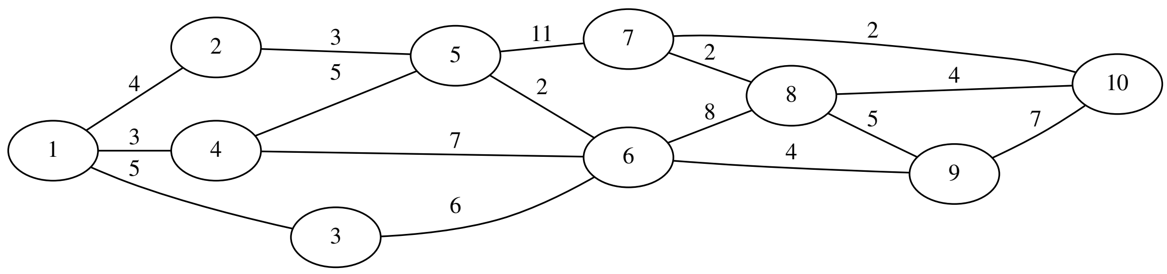
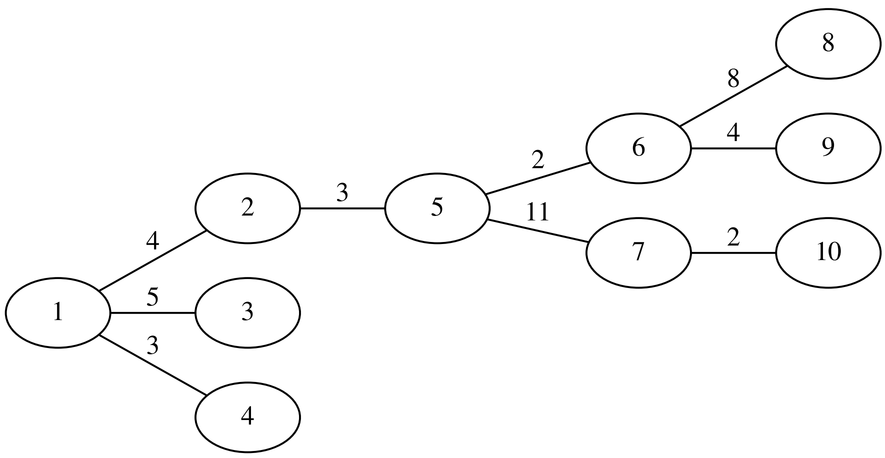


FIGURE 12.4 – Un graphe valué positivement

Correction proposée:



12.3 Skynet d'après Codingame©

Un arbre recouvrant

Nous avons vu en cours que l'algorithme de Dijkstra permet d'obtenir un arbre a recouvrant depuis un sommet s sur un graphe valué avec des nombres positifs tel que le chemin de a reliant s a tout sommet du graphe est le plus court. Cet algorithme est le suivant :

fonction dijkstra (g : Graphe<Sommet,,**ReelPositif**>, s : Sommet) **:** Arbre<Sommet>, Dictionnaire<Sommet, **ReelPositif**>

⌊**précondition(s)** sommetPresent(g,s)

Déclaration arbreRecouvrant : Arbre<Sommet>, cout : Dictionnaire<Sommet,**ReelPositif**>
l : Liste<Liste<Sommet>>, c : **ReelPositif**
sommetDeA, sommetAAjouter : Sommet

debut

```
    arbreRecouvrant ← arbreInitial(s)
    cout ← dictionnaire()
    ajouter(cout,s,0)
    l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)
    tant que non estVide(l) faire
        sommetDeA,sommetAAjouter,c ← arcMinimal(g,l,cout)
        ajouter(cout,
                sommetAAjouter,
                obtenirValeur(cout,sommetDeA)+c
        )
        ajouterCommeFils(arbreRecouvrant,sommetDeA,sommetAAjouter)
        l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)
    fintantque
    retourner arbreRecouvrant, cout
fin
```

Tel que :

- `arbreInitial` crée un arbre possédant uniquement le noeud s
- `arcsEntreArbreEtGraphe` permet d'obtenir la liste des arcs présents dans le graphe G, dont le sommet source est présent dans l'arbre mais pas le sommet destination ;
- `arcMinimal` permet d'identifier l'arc (sommet source, sommet destination) dont le sommet destination est le plus proche (au sens du dictionnaire de $cout$) des sommets de a ainsi que le coût supplémentaire pour l'atteindre
- `ajouterCommeFils` permet d'ajouter un sommet dans l'arbre en spécifiant son père.

Signatures

Donnez les signatures des sous-programmes précédents.

Correction proposée:

- **fonction** arbreInitial **(**s : Sommet**) :** Arbre<Sommet>
- **fonction** arcsEntreArbreEtGraphe **(**g : Graphe<Sommet,**ReelPositif**>, a : Arbre<Sommet>**) :** Liste<LIste<Sommet
- **fonction** arcMinimal **(**g : Graphe, arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet, **ReelPositif**>**) :** Sommet, Sommet, **ReelPositif**

  ⌊**précondition(s)** non estVide(arcs)
- **procédure** ajouterCommeFils **(E/S** a : Arbre<Sommet>, **E** sommetPere, sommetFils : Sommet**)**

Algorithme

Donnez l'algorithme de la fonction `sommetsAccessiblesDepuisArbre` (n'oubliez pas de décomposer le problème si besoin).

Correction proposée:

Analyse :

- arcsEntreArbreEtGraphe : Graphe × Arbre<Sommet> → Liste<Liste<Sommet>>
  - sommetsDeLArbre : Arbre<Sommet> → Liste<Sommet>
  - estPresent : Liste<Sommet> × Sommet → **Booleen**

Conception préliminaire :

— **fonction** sommetsDeLArbre (a : Arbre<Sommet>) **:** Liste<Sommet>

— **fonction** estPresent (l : Liste<Sommet>, s : Sommet) **: Booleen**

Conception détaillée :

```
fonction sommetsDeLArbre (a : Arbre<Sommet>) : Liste<Sommet>
   Déclaration  res : Liste<Sommet>
debut
   res ← liste()
   parcourir(a, res)
   retourner temp
fin
procédure parcourir (E a : Arbre<Sommet>, E/S l : Liste<Sommet>)
debut
   si non estVide(a) alors
      inserer(temp,1,obtenirElement(a))
      pour chaque f de obtenirFils(a)
         parcourir(a, l)
      finpour
   finsi
fin
fonction estPresent (l : Liste<Sommet>, s : Sommet ) : Booleen
   Déclaration  i : Naturel
debut
   i ← 1
   tant que i≤longueur(l) et obtenirElement(l,i)≠s faire
      i ← i+1
   fintantque
   retourner i>longueur(l)
fin
fonction arcsEntreArbreEtGraphe (g : Graphe<Sommet,ReelPositif>, a : Arbre<Sommet>) : Liste<LIste<Sommet>>

   Déclaration  res : Liste<Liste<Sommet>>
                arc :Liste<Sommet>
                sommetsDeA : Liste<Sommet>
debut
   res ← liste()
   sommetsDeA ← sommetsDeLArbre(a)
   pour chaque s1 de sommetsDeA
      pour chaque s2 de obtenirSommetAdjascent(g,s1)
         si non estPresent(sommetsDaA,s2) alors
            arc ← liste()
            inserer(arc,1,s1)
            inserer(arc,2,s2)
            inserer(res,1,arc)
         finsi
      finpour
   finpour
```

```
    retourner res
fin
```

12.3.1 Le chemin le plus court

Donnez l'algorithme de la fonction suivante qui permet d'obtenir le chemin (une liste de sommets) le plus court permettant d'aller d'un sommet $s1$ à un sommet $s2$ d'un graphe valué avec des nombres positifs :

— **fonction** cheminPlusCourt **(**g :Graphe, s1,s2 : Sommet**) :** Liste<Sommet>
  ⌊**précondition(s)** sommetPresent(g,s1) et sommetPresent(g,s2)

Correction proposée:

```
fonction cheminPlusCourtR (a : Arbre<Sommet>, sCible : Sommet) : Liste<Sommet>
    Déclaration chemin : Liste<Sommet>
                fils : Sommet
debut
    chemin ← liste()
    si estVide(a) alors
        retourner chemin
    sinon
        si obtenirElement(a)=sCible alors
            inserer(chemin,1,sCible)
            retourner chemin
        sinon
            i ← 1
            tant que i≤longueur(obtenirFils(a)) et estVide(chemin) faire
                chemin ← cheminPlusCourtR(obtenirElement(obtenirFils(a),i),sCible)
                si estVide(chemin) alors
                    i ← i+1
                finsi
            fintantque
            si non estVide(chemin) alors
                inserer(chemin,1,obtenirElement(obtenirFils(a),i))
            finsi
            retourner chemin
        finsi
    finsi
fin
fonction cheminPlusCourt (g : Graphe, s1,s2 : Sommet) : Liste<Sommet>
    ⌊précondition(s) sommetPresent(g,s1) et sommetPresent(g,s2)
    Déclaration a : Arbre<Sommet>
                c : Dictionnaire<Sommet,ReelPositif>
                res : Liste<Sommet>
debut
    a,c ← dijkstra(g,s1)
    retourner cheminPlusCourtR(a,s2,res)
fin
```

12.3.2 Skynet le virus

Le site Web `www.codingame.com` propose des exercices ludiques de programmation. L'un des exercices, ≪ Skynet le virus ≫ est présenté de la façon suivante :

≪ Votre virus a créé une *backdoor* sur le réseau Skynet vous permettant d'envoyer de nouvelles instructions au virus en temps réel. Vous décidez de passer à l'attaque active en empêchant Skynet de communiquer sur son propre réseau interne. Le réseau Skynet est divisé en sous-réseaux. Sur chaque sous-réseau un agent Skynet a pour tâche de transmettre de l'information en se déplaçant de noeud en noeud le long de liens et d'atteindre une des passerelles qui mène vers un autre sous-réseau. Votre mission est de reprogrammer le virus pour qu'il coupe les liens dans le but d'empêcher l'agent Skynet de sortir de son sous-réseau et ainsi d'informer le *hub* central de la présence de notre virus. ≫

Bref, l'agent Skynet (S) est sur un graphe (par exemple celui de la figure 12.5 où les identifiants des sommets ne sont pas indiqués) valué (avec la valeur 1 pour chaque arc) dont certains sommets sont des passerelles (P). Le but du jeu est d'empécher l'agent skynet d'atteindre une des passerelles en supprimant le moins d'arcs du graphe.

L'algorithme de ce jeu est proposé par la procédure `skynet`. L' `agentSkynet` parcourt le graphe (grâce à la fonction `seDeplace`) de sommet en sommet à chaque itération. Pour résoudre ce problème, il faut couper un arc du graphe à chaque itération de façon à ce que l'agent Skynet ne puisse pas atteindre l'une des passerelles. De plus il faut faire le moins de coupures possibles (le score est fonction de ce paramètre). Pour cela il suffit de supprimer le premier arc du chemin le plus court entre l'`agenSkynet` et la plus proche passerelle.

Complétez l'algorithme de la procédure `skynet` (remplacer les ... par une ou plusieurs instructions).

procédure skynet (**E/S** g : Graphe<Sommet>, **E** agentSkynet : Sommet, **S** agentSkynetAAtteindPasserelle : **Booleen)**

Déclaration passerelles : Liste<Sommet>
s : Sommet
...

debut
passerelles ← sommetsDesPasserelles(g)
tant que agentSkynetPeutAtteindreUnePasserelle(g,agentSkynet) et non estPresent(passerelles, agentSkynet) **faire**
...
supprimerArc(g,agentSkynet,s)
agentSkynet ← seDeplace(g, agentSkynet)
fintantque
agentSkynetAAtteindPasserelle ← estPresent(agentSkynet,passerelles)
fin

Correction proposée:

procédure skynet (**E/S** g : Graphe<Sommet>, **E** agentSkynet : Sommet, **S** agentSkynetAAtteindPasserelle : **Booleen)**

Déclaration passerelles : Liste<Sommet>
s1,s2,p : Sommet
chMin,temp : Liste<Sommet>
lmin : **Naturel**

debut
passerelles ← sommetsDesPasserelles(g)
tant que agentSkynetPeutAtteindreUnSommet(g,agentSkynet) et non estPresent(passerelles, agentSkynet) **faire**
chMin ← cheminPlusCourt(g,agentSkynet,obtenirElement(passerelles,1))

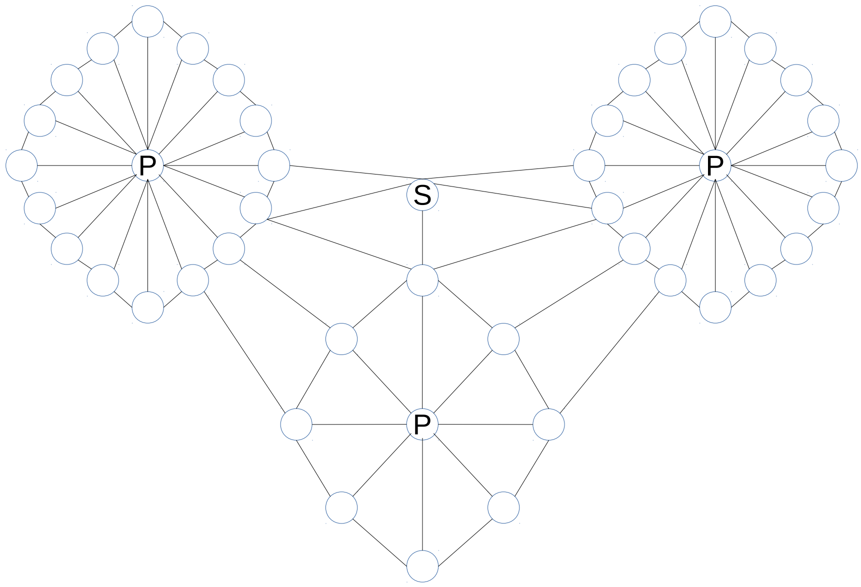


FIGURE 12.5 – Un sous réseau Skynet

```
        lmin ← longueur(chMin)
        pour chaque p de passerelles
            ch ← cheminPlusCourt(g,agentSkynet,p)
            si non estVide(ch) et longueur(ch)<lmin alors
                lmin ← longueur(ch)
                chMin ← ch
            finsi
        finpour
        s ← obtenirElement(ch,2)
        supprimerArc(g,agentSkynet,s)
        agentSkynet ← seDeplace(g, agentSkynet)
    fintantque
    agentSkynetAAtteindPasserelle ← estPresent(agentSkynet,passerelles)
fin
```

Chapitre 13

Programmation dynamique

Attendus d'apprentissages disciplinaires évalués

— AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple
— CD701 : Définir la programmation dynamique
— CD702 : Appliquer la programmation dynamique pour des cas simples
— CD801 : Concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)

13.1 L'algorithme de Floyd-Warshall

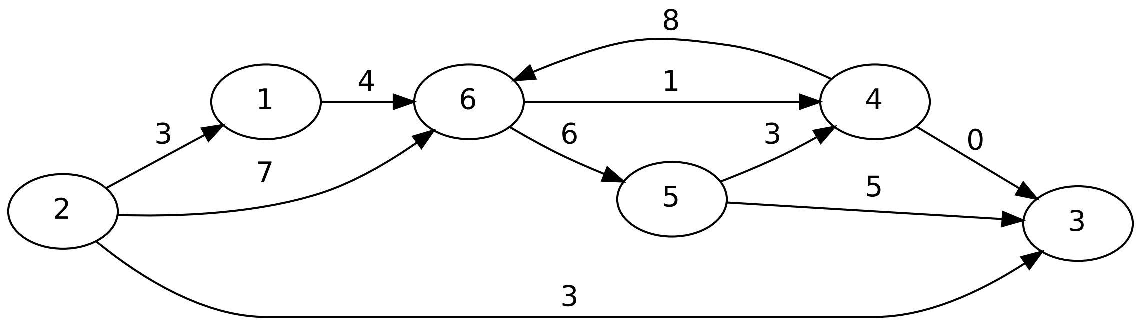


FIGURE 13.1 – Un graphe orienté valué

L'algorithme de Floyd-Warshall est un algorithme qui permet de calculer la longueur du plus court chemin entre tous les nœuds d'un graphe orienté valué positivement.

« L'algorithme repose sur la remarque suivante : si $(a_0, \ldots, a_i, \ldots, a_p)$ est un plus court chemin de a_0 à a_p , alors $(a_0, ..., a_i)$ est un plus court chemin de a_0 à a_i , et $(a_i, ..., a_p)$ un plus court chemin de a_i à a_p . De plus, comme les arêtes sont valuées positivement, tout chemin contenant un cycle est nécessairement plus long que le même chemin sans le cycle, si bien qu'on peut se limiter à la recherche de plus courts chemins passant par des sommets deux à deux distincts.

Floyd montre donc qu'il suffit de calculer la suite de matrices définies par :

$$M^k_{i,j} = min(M^{k-1}_{i,j}, M^{k-1}_{i,k} + M^{k-1}_{k,j}).»$$ [1]

tel que M^0 est la matrice d'adjacence du graphe avec :

— les nœuds qui sont numérotés de 1 à n (et k varie de 1 à n) ;

— $M^0_{i,i} = 0$;

— $M^0_{i,j} = +\infty$ s'il n'existe pas d'arc reliant i à j.

1. Donnez la matrice d'adjacence M^0 du graphe proposé par la figure 13.1 (pour plus de clarté, vous pouvez ne pas noter les $+\infty$).
2. Donnez les matrices M de Floyd pour k variant de 1 à 6.
3. À partir de la matrice M^6 donnez la longueur du plus court chemin reliant le nœud 2 au nœud 4.

Correction proposée:

1.

$$M^0 = \begin{array}{cc} & \begin{array}{cccccc}1&2&3&4&5&6\end{array} \\ \begin{array}{c}1\\2\\3\\4\\5\\6\end{array} & \begin{pmatrix} 0 & & & & & 4 \\ 3 & 0 & 6 & & & 7 \\ & & 0 & & & \\ & & & 0 & & 8 \\ & & 5 & 3 & 0 & \\ & & & 1 & 6 & 0 \end{pmatrix}\end{array}$$

2.

$$M^1 = \begin{array}{cc} & \begin{array}{cccccc}1&2&3&4&5&6\end{array} \\ \begin{array}{c}1\\2\\3\\4\\5\\6\end{array} & \begin{pmatrix} 0 & & & & & 4 \\ 3 & 0 & 6 & & & 7 \\ & & 0 & & & \\ & & & 0 & & 8 \\ & & 5 & 3 & 0 & \\ & & & 1 & 6 & 0 \end{pmatrix}\end{array}$$

$$M^3 = M^2 = M^1$$

$$M^4 = \begin{array}{cc} & \begin{array}{cccccc}1&2&3&4&5&6\end{array} \\ \begin{array}{c}1\\2\\3\\4\\5\\6\end{array} & \begin{pmatrix} 0 & & & & & 4 \\ 3 & 0 & 6 & & & 7 \\ & & 0 & & & \\ & & & 0 & & 8 \\ & & 3 & 3 & 0 & 11 \\ & & 1 & 1 & 6 & 0 \end{pmatrix}\end{array}$$

$$M^5 = M^4$$

$$M^6 = \begin{array}{cc} & \begin{array}{cccccc}1&2&3&4&5&6\end{array} \\ \begin{array}{c}1\\2\\3\\4\\5\\6\end{array} & \begin{pmatrix} 0 & & 5 & 5 & 10 & 4 \\ 3 & 0 & 6 & 8 & 13 & 7 \\ & & 0 & & & \\ & & & 0 & 14 & 8 \\ & & 3 & 3 & 0 & 11 \\ & & 1 & 1 & 6 & 0 \end{pmatrix}\end{array}$$

3. Le longueur du chemin le plus court allant de 2 à 4 est donnée par $M^6_{2,4} = 8$

1. `http://www.nimbustier.net/publications/djikstra/floyd.html`

13.2 La distance de Levenshtein

≪ La distance de Levenshtein est une distance mathématique donnant une mesure de la similarité entre deux mots. Elle est égale au nombre minimal de lettres qu'il faut supprimer, insérer ou remplacer pour passer d'un mot à l'autre.

On appelle distance de Levenshtein entre deux mots M et P le coût minimal pour aller de M à P en effectuant les opérations élémentaires suivantes :

— substitution d'une lettre de M en une lettre de P ;

— ajout dans M d'une lettre de P ;

— suppression d'une lettre de M.

On associe ainsi à chacune de ces opérations un coût. Le coût est toujours égal à 1, sauf dans le cas d'une substitution de lettres identiques, il vaut alors 0. ≫ (inspiré de Wikipédia).

Pour calculer cette distance on utilise un matrice m de taille $|P|+1 \times |M|+1$ (tel $|s|$ représente la longueur d'un mot s) indicée à partir de 0, tel que :

$$m_{0,j} = j, j \in 0..|M|$$

$$m_{i,0} = i, i \in 0..|P|$$

$$m_{i,j} = min(m_{i,j-1} + 1, m_{i-1,j} + 1, m_{i-1,j-1} + 1_{P_i,M_j}), i \in 0..|P|, j \in 0..|M|$$

tel que $1_{P_i,M_j}$ vaut 0 si $P_i = M_j$ (la ième lettre de P est égale à la jème lettre de M), 1 sinon.
La distance de Levenshtein est alors égale à $m_{|P|,|M|}$.

1. Remplissez la matrice suivante pour calculer la distance de Levenshtein entre les deux mots "voiture" et "toile".

$$m = \begin{array}{c} \\ \llcorner\!\lrcorner \\ t \\ o \\ i \\ l \\ e \end{array} \begin{array}{c} \llcorner\!\lrcorner \quad v \quad o \quad i \quad t \quad u \quad r \quad e \\ \begin{pmatrix} & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \end{pmatrix} \end{array}$$

Correction proposée:

$$m = \begin{array}{c|cccccccc} & \llcorner\!\lrcorner & v & o & i & t & u & r & e \\ \hline \llcorner\!\lrcorner & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ t & 1 & 1 & 2 & 3 & 3 & 4 & 5 & 6 \\ o & 2 & 2 & 1 & 2 & 3 & 4 & 5 & 6 \\ i & 3 & 3 & 2 & 1 & 2 & 3 & 4 & 5 \\ l & 4 & 4 & 3 & 2 & 2 & 3 & 4 & 5 \\ e & 5 & 5 & 4 & 3 & 3 & 3 & 4 & 4 \end{array}$$

2. À quel paradigme de conception appartient cet algorithme ? Justifiez.
Correction proposée:
C'est un algorithme de programmation dynamique car :
   - c'est un algorithme du type "diviser pour régner" qui calcule tout d'abord les résultats de base pour les assembler et ainsi calculer le résultat recherché : la valeur $m_{i,j}$ est la distance de Levenshtein entre les i premières lettres du mot M et les j premières lettre du mot P.
   - il utilise un tableau pour stocker des valeurs qui sont susceptibles d'être calculées plusieurs fois.

3. Donnez l'algorithme de la fonction qui permet de calculer la distance de Levenshtein entre deux mots.
   Correction proposée:

   fonction cout (c1, c2 : **Caractere) : Naturel**
   debut
   si c1=c2 **alors**
   retourner 0
   finsi
   retourner 1
   fin
   fonction distanceLevenhstein (mot1, mot2 : **Chaine de caracteres) : Naturel**
   ⌊**précondition(s)** longeur(mot1)≤MAX et longueur(mot2)≤MAX

   Déclaration m : **Tableau[**0..MAX**][**0..MAX**] de Naturel**

   debut
   pour i ←0 **à** longueur(mot1) **faire**
   m[i,0] ← i
   finpour
   pour j ←0 **à** longueur(mot2) **faire**
   m[0,j] ← j
   finpour
   pour i ←1 **à** longueur(mot1 **faire**
   pour j ←1 **à** longueur(mot2) **faire**
   m[i,j] ← min3(m[i,j-1]+1, m[i-1,j]+1, m[i-1,j-1] + cout(iemeCaractere(mot1,i), iemeCaractere(mot2,j)))
   finpour
   finpour
   retourner m[longueur(mot1), longueur(mot2)]
   fin