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.

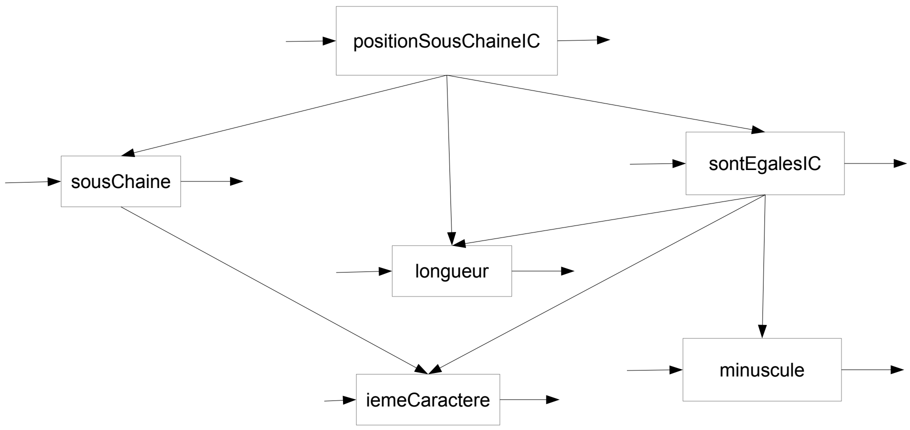


FIGURE 1.1 – Une analyse descendante

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.

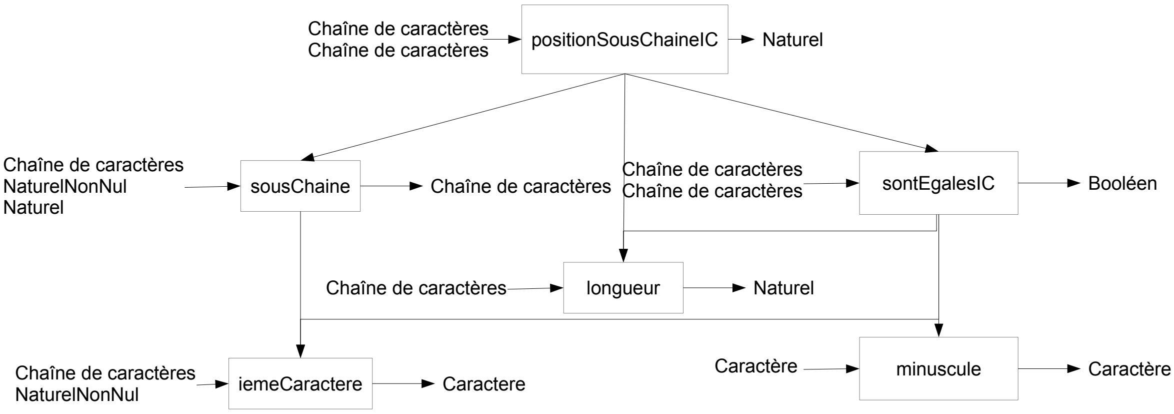
L'opération `positionSousChaineIC` retournera la première position de la chaîne recherchée dans la chaîne si cette première est présente, 0 sinon.

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)$>$0
longueur(chaineARechercher)$\leq$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$\leq$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)$>$0
longueur(chaineARechercher)$\leq$longueur(chaine)

Déclaration i : **Naturel**

debut
i ← 1
tant que i+longueur(chaineARechercher)-1$\leq$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$\leq$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, finDeLaPlusGrandeSsSequence : **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
∃ 1 ≤ i ≤ n tel que t[i] = element
estTrieEnOrdreCroissant(t)

Déclaration g,d,m : **Naturel**

debut
g ← 1
d ← n
tant que g ≠ d **faire**
m ← (g + d) div 2
si t[m] ≥ 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.

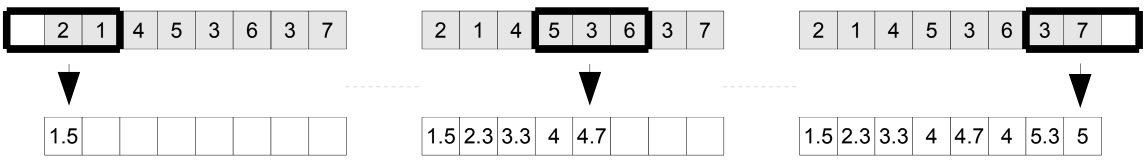


FIGURE 2.1 – Lissage d'un signal avec une fenêtre de taille 3

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 ≤ b
                    strictementMonotone(f,a,b)
  Déclaration  m : Reel
debut
  m ← (a + b) / 2
  si (b - a)≤ 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

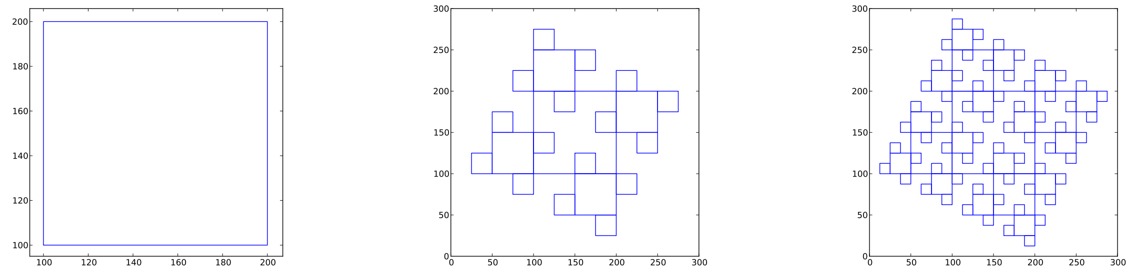
- AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple
- CD201 : Identifier et résoudre le problème des cas non récursifs
- CD202 : Identifier et résoudre le problème des cas récursifs
- CD203 : Identifier une récursivité terminale et non terminale et ce que cela implique

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)**

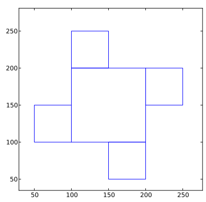


(a) $carres(g, 100, 100, 100, 1)$ (b) $carres(g, 100, 100, 100, 3)$ (c) $carres(g, 100, 100, 100, 4)$

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

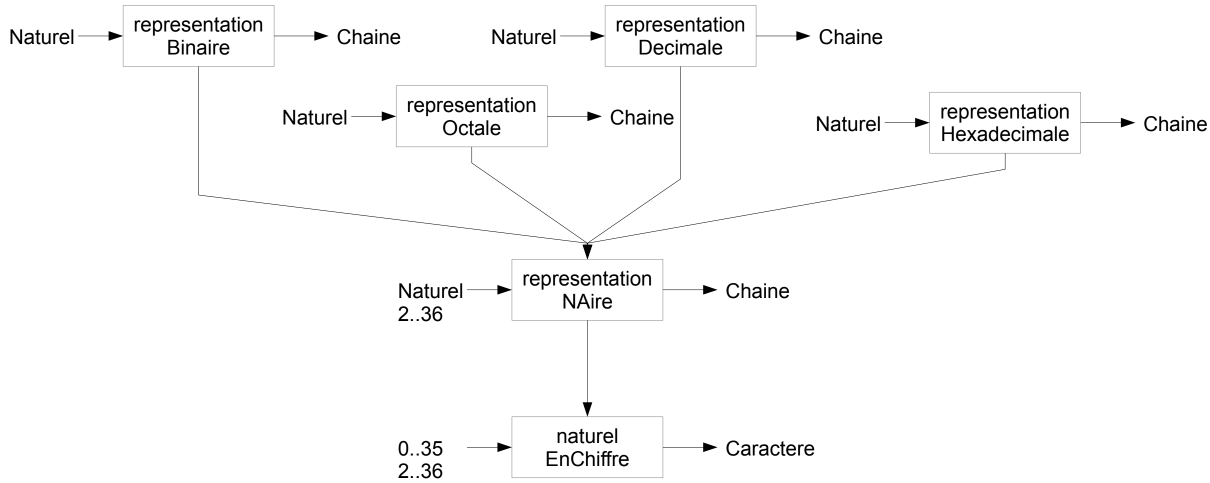
L’objectif de cet exercice est de concevoir quatre fonctions permettant de représenter un naturel en chaîne de caractères telles que la première fonction donnera une représentation binaire, la deuxième une représentation octale, la troisième une représentation décimale et la dernière une représentation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

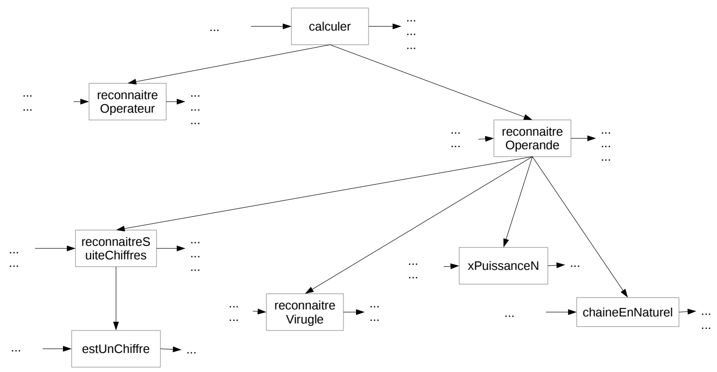
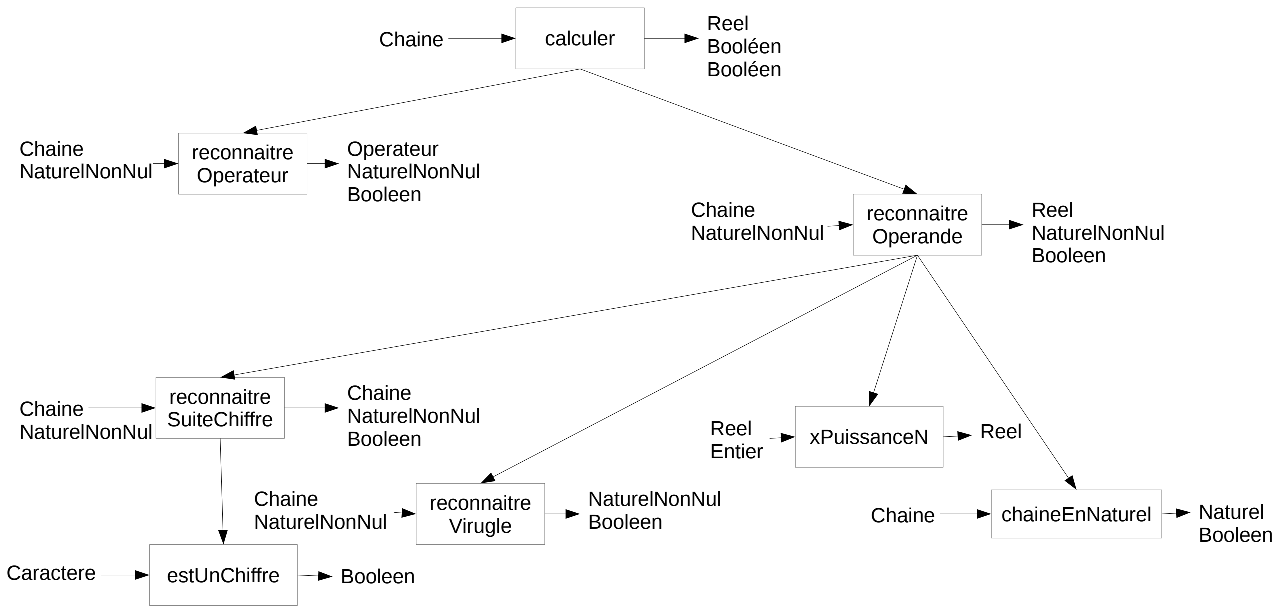


FIGURE 5.1 – Analyse descendante d'une calculatrice simple

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 ≤ 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 ≤ 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 $\leq$ 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$\leq$ 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** × **Reel** → Point2D
obtenirX: Point2D → **Reel**
obtenirY: Point2D → **Reel**
distanceEuclidienne: Point2D × Point2D → **ReelPositif**
translater: Point2D × Point2D → Point2D
faireRotation: Point2D × Point2D × **Reel** → 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

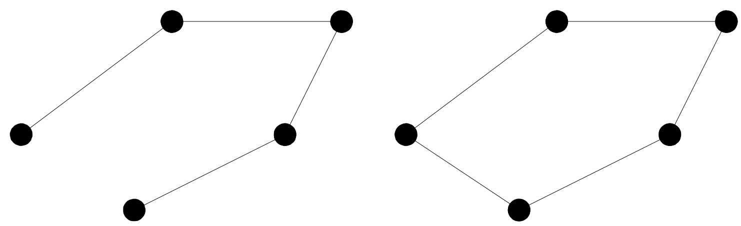
≪ Une ligne polygonale, ou ligne brisée (on utilise aussi parfois polyligne par traduction de l'anglais *polyline*) est une figure géométrique formée d'une suite de segments, la seconde extrémité de chacun d'entre eux étant la première du suivant.[…] Un polygone est une ligne polygonale fermée. ≫ (Wikipédia)

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).

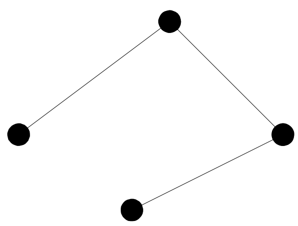


FIGURE 6.2 – Supression d'un point

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 $|\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \alpha_5| = 360$.

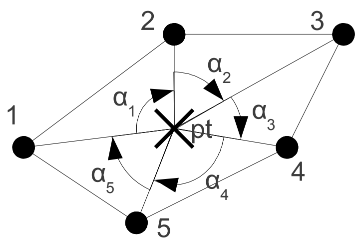


FIGURE 6.3 – Point à l'intérieur d'une polyligne

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 :

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

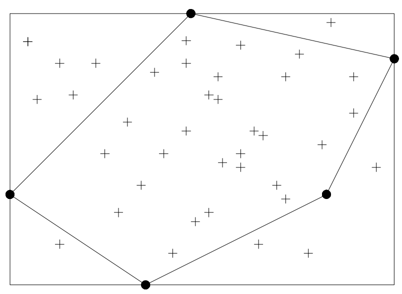


FIGURE 6.4 – Calcul de la surface d'une polyligne par la méthode de Monté-Carlo

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 × **Naturel** → **Reel**

- **rectangleEnglobant** Polyligne → Point2D × Point2D
- **surfaceRectangle** Point2D × Point2D → **Reel**
- **pointAleatoireDansRectangle** Point2D × Point2D → 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
```