Algorithmique avancée et programmation C
Exercices de TD
3.2

N. Delestre

Table des matières

Avant propos

Évaluation par compétences

Depuis l'année universitaire 2018-2019, la validation du cours « Algorithique avancée et programmation C » utilise une évaluation par compétences. Le référentiel de compétences 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 compétences.

Quelque soit l'exercice les compétences suivantes sont évaluées :

- AN001 : Savoir désigner les choses (identifiant significatif)
- AN002 : Savoir être précis quant aux types de données utilisés
- AN003 : Connaître le rôle de l'analyse
- CP001 : Savoir ce que représente le paradigme de programmation impératif
- CP002 : Savoir ce que représente le paradigme de programmation structuré
- CP006 : Connaître le rôle de la conception préliminaire
- CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
- CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
- CD006 : Savoir choisir la bonne itération
- CD007 : Savoir quelles catégories de paramètres effectifs peut être utilisées pour un passage de paramètre donné
- CD009 : Savoir 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 compétences - exercices

Compétences	Exercices
AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple	3.4, 7, 12, 13
AN101 : Savoir identifier les entrées et sorties d'un problème	1.3, 2.4, 4, 5
AN102 : Savoir décomposer logiquement un problème	2.4, 4
AN103 : Savoir généraliser un problème	4
AN104 : Savoir si un problème doit être décomposé	2.4
AN201 : Savoir identifier les dépendances d'un TAD	6, 8, 12

Compétences	Exercices
AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier	6, 8, 12
AN204 : Savoir formaliser des opérations d'un TAD	6, 12
AN205 : Savoir formaliser les préconditions d'une opération d'un TAD	6, 8
AN206 : Savoir formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD	6, 12
AN301 : Savoir lister les collections usuelles	8
CP003 : Savoir choisir entre une fonction et une procédure	1.3, 4, 5, 6, 8, 12
CP004 : Savoir 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 : Savoir choisir un passage de paramètre (E, S, E/S)	2.2, 5, 6, 12
CD001 : Savoir dissocier les deux rôles du développeur : concepteur et utilisateur	6
CD002 : En tant qu'utilisateur, savoir respecter une signature	1.1, 1.2
CD003 : Savoir utiliser le principe d'encapsulation	6, 8
CD101 : Savoir estimer la taille d'un problème (n)	1.4, 4
CD102 : Savoir calculer une complexité dans le pire et le meilleur des cas	1.4, 4, 7
CD104 : Savoir écrire un algorithme d'une complexité donnée	2.3, 3.2, 3.3
CD201 : Savoir identifier et résoudre le problème des cas non récursifs	3.1, 3.2, 3.3, 3.4, 3.5, 7, 8, 10, 12
CD202 : Savoir identifier et résoudre le problème des cas récursifs	3.1, 3.2, 3.3, 3.4, 3.5, 7, 8, 10, 12
CD203 : Savoir identifier une récursivité terminale et non terminale et ce que cela implique	3.1, 3.2, 3.3, 3.4, 3.5
CD301 : Savoir identifier un problème qui se résout à l'aide d'un algorithme dichotomique	2.3
CD302 : Savoir définir l'espace de recherche d'un algorithmique dichotomique	1.4, 2.3
CD303 : Savoir diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)	1.4, 2.3
CD403 : Savoir concevoir et utiliser des arbres (binaires, n-aires)	10
CD501 : Connaître les algorithmes des différents tris et leurs complexités	7
CD601 : Savoir concevoir des collections à l'aide de SDD	10
CD602 : Connaître les algorithmes d'insertion et de suppression (naïfs et AVL) dans un arbre binaire de recherche	10
CD701 : Savoir définir la programmation dynamique	13

Compétences	Exercices
CD702 : Savoir appliquer la programmation dynamique pour des cas simples	13
CD801 : Savoir concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)	12
CD804 : Connaître des algorithmes de recherche du plus court chemin : Dijkstra et A*	12
CD901 : Savoir 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 :

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

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)

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

— CD006 : Savoir 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 ≫).

1.2 Palindrome

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)

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

— CD006 : Savoir 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.

1.3 Position d'une sous-chaîne

Compétences évaluées

- AN101 : Savoir identifier les entrées et sorties d'un problème
- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir 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.

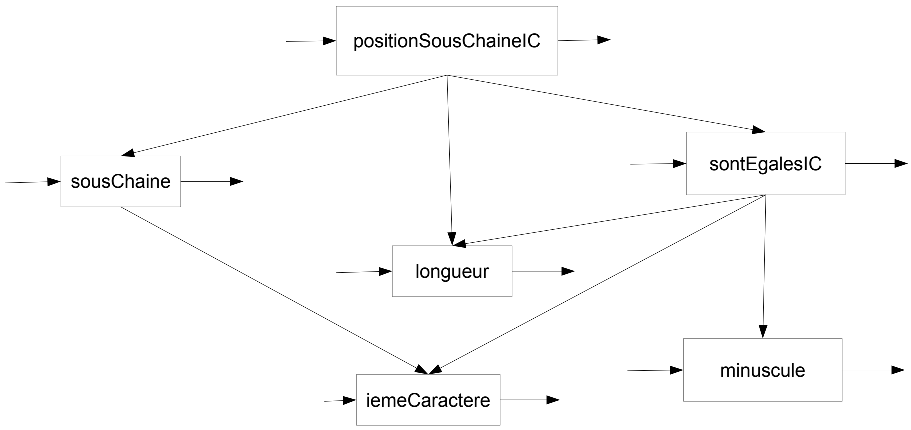


FIGURE 1.1 – Une analyse descendante

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`

1.4 Racine carrée d'un nombre : recherche par dichotomie

Compétences évaluées

— CD302 : Savoir définir l'espace de recherche d'un algorithmique dichotomique

— CD303 : Savoir diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)

— CD101 : Savoir estimer la taille d'un problème (n)

— CD102 : Savoir 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 ?

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

Compétences évaluées
— CP004 : Savoir 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.

2.2 Sous-séquences croissantes

Compétences évaluées
— CP003 : Savoir choisir entre une fonction et une procédure
— CP004 : Savoir concevoir une signature (préconditions incluses)
— CP005 : Savoir choisir un passage de paramètre (E, S, E/S)
— CD005 : Savoir 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.

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

Compétences évaluées

- CP004 : Savoir concevoir une signature (préconditions incluses)
- CD104 : Savoir écrire un algorithme d'une complexité donnée
- CD301 : Savoir identifier un problème qui se résout à l'aide d'un algorithme dichotomique
- CD302 : Savoir définir l'espace de recherche d'un algorithmique dichotomique
- CD303 : Savoir 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.

2.4 Lissage de courbe

Compétences évaluées

- AN101 : Savoir identifier les entrées et sorties d'un problème
- AN102 : Savoir 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.

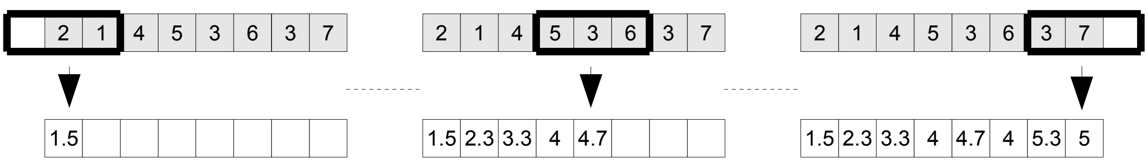


FIGURE 2.1 – Lissage d'un signal avec une fenêtre de taille 3

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)

Chapitre 3

Rappels : récursivité

3.1 Palindrome

Compétences évaluées

- CP004 : Savoir concevoir une signature (préconditions incluses)
- CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- CD202 : Savoir identifier et résoudre le problème des cas récursifs
- CD203 : Savoir 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 ?

3.2 Puissance d'un nombre

Compétences évaluées

- CP004 : Savoir concevoir une signature (préconditions incluses)
- CD104 : Savoir écrire un algorithme d'une complexité donnée
- CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- CD202 : Savoir identifier et résoudre le problème des cas récursifs
- CD203 : Savoir 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 $\Omega(n)$.

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

Compétences évaluées

- — CP004 : Savoir concevoir une signature (préconditions incluses)
- — CD104 : Savoir écrire un algorithme d'une complexité donnée
- — CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- — CD202 : Savoir identifier et résoudre le problème des cas récursifs
- — CD203 : Savoir 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 ne s'annule qu'une seule et unique fois sur $[a, b]$.

3.4 Dessin récursif

Compétences évaluées

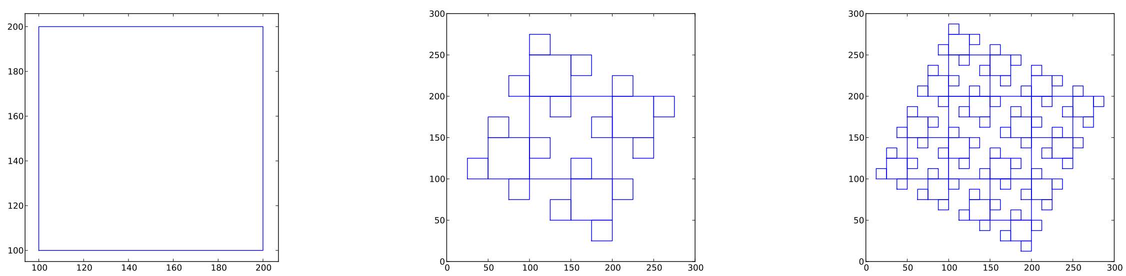
- — AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple
- — CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- — CD202 : Savoir identifier et résoudre le problème des cas récursifs
- — CD203 : Savoir identifier une récursivité terminale et non terminale 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) $carres(100, 100, 100, 1)$ (b) $carres(100, 100, 100, 3)$ (c) $carres(100, 100, 100, 4)$

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`.

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

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)
— CD201 : Savoir identifier et résoudre le problème des cas non récursifs
— CD202 : Savoir identifier et résoudre le problème des cas récursifs
— CD203 : Savoir 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

Chapitre 4

Représentation d'un naturel

Compétences évaluées

- AN101 : Savoir identifier les entrées et sorties d'un problème
- AN102 : Savoir décomposer logiquement un problème
- AN103 : Savoir généraliser un problème
- AN104 : Savoir si un problème doit être décomposé
- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir concevoir une signature (préconditions incluses)
- CD001 : Savoir dissocier les deux rôles du développeur : concepteur et utilisateur
- CD002 : En tant qu'utilisateur, savoir 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.

4.2 Conception préliminaire

Donnez les signatures des fonctions ou procédures identifiées précédemment.

4.3 Conception détaillée

Donnez les algorithmes de ces fonctions ou procédures

Chapitre 5

Calculatrice

Compétences évaluées

- AN101 : Savoir identifier les entrées et sorties d'un problème
- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir concevoir une signature (préconditions incluses)
- CP005 : Savoir 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.

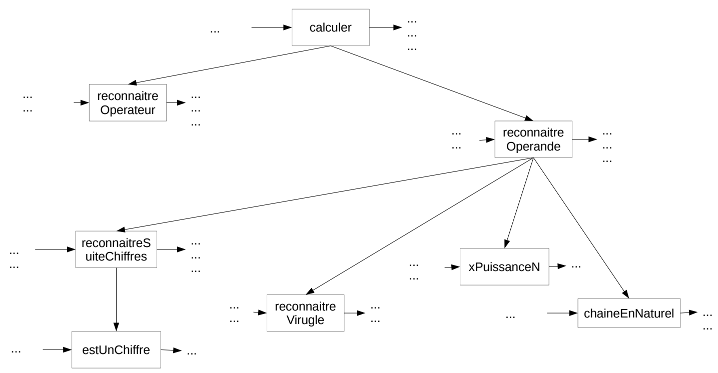


FIGURE 5.1 – Analyse descendante d'une calculatrice simple

5.2 Conception préliminaire

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

5.3 Conception détaillée

Donnez les algorithmes des fonctions et procédures identifées.

Chapitre 6

Un peu de géométrie

Compétences évaluées

- AN201 : Savoir identifier les dépendances d'un TAD
- AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
- AN204 : Savoir formaliser des opérations d'un TAD
- AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
- AN206 : Savoir formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD
- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir concevoir une signature (préconditions incluses)
- CP005 : Savoir choisir un passage de paramètre (E, S, E/S)
- CD003 : Savoir 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`)
2. Conception préliminaire : Donnez les signatures des fonctions et procédures des opérations de ce TAD

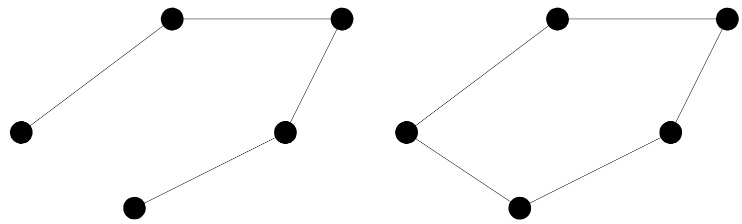
Remarque(s) :

- 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.



(a) polyligne ouverte (b) polyligne fermée

FIGURE 6.1 – Deux polylignes

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;
— 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).

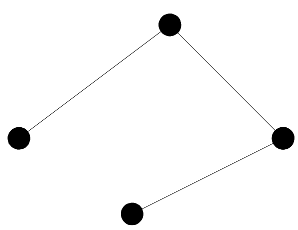


FIGURE 6.2 – Supression d'un point

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.

6.2.2 Conception préliminaire

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

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.

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

Proposez le code de la fonction suivante :`estALInterieur`

fonction estALinterieur (p : Polyligne, pt : Point2D) **: Booleen**
  ⌊**précondition(s)** estFerme(p) et non estSurLaFrontiere(pt,p)

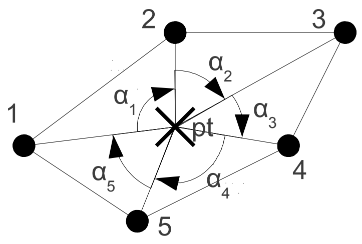


FIGURE 6.3 – Point à l'intérieur d'une polyligne

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 :

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

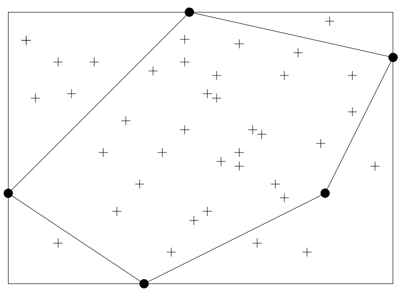


FIGURE 6.4 – Calcul de la surface d'une polyligne par la méthode de Monté-Carlo

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

— **fonction** reelAleatoire (borneMin,bornneMax : **Reel) : Reel**

1. Proposez l'analyse descendante pour le calcul d'une surface d'une polyligne à l'aide de la méthode de Monté-Carlo.
2. Donnez les signatures des procédures et fonctions de votre analyse descendante.
3. Donnez l'algorithme de l'opération principale (au sommet de votre analyse descendante).

Chapitre 7

Tri par tas

Compétences évaluées

— AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple
— CD102 : Savoir calculer une complexité dans le pire et le meilleur des cas
— CD201 : Savoir identifier et résoudre le problème des cas non récursifs
— CD202 : Savoir identifier et résoudre le problème des cas récursifs
— CD501 : Connaître 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

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

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.

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.

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.

Chapitre 8

Sudoku

Compétences évaluées

- AN201 : Savoir identifier les dépendances d'un TAD
- AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
- AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
- AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
- AN301 : Savoir lister les collections usuelles
- CP003 : Savoir choisir entre une fonction et une procédure
- CD003 : Savoir utiliser le principe d'encapsulation
- CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- CD202 : Savoir 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.

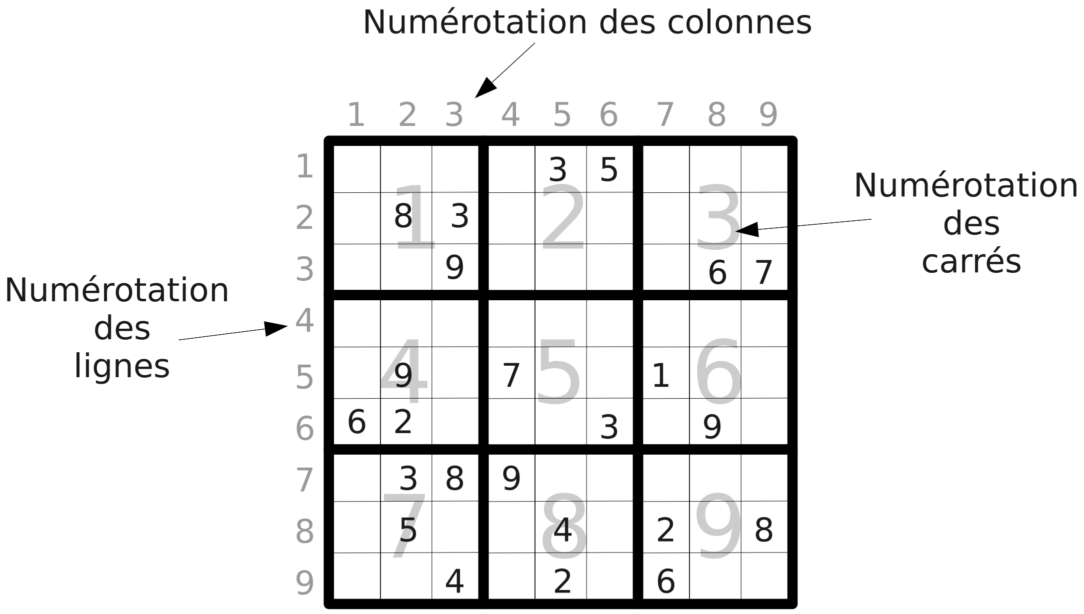


FIGURE 8.1 – Exemple de grille de Sudoku

	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.

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.

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**

Chapitre 9

Liste

Compétences évaluées

— AN301 : Savoir lister les collections usuelles
— CP003 : Savoir choisir entre une fonction et une procédure
— CP004 : Savoir concevoir une signature (préconditions incluses)
— CP005 : Savoir choisir un passage de paramètre (E, S, E/S)
— CD003 : Savoir utiliser le principe d'encapsulation
— CD201 : Savoir identifier et résoudre le problème des cas non récursifs
— CD202 : Savoir identifier et résoudre le problème des cas récursifs
— CD401 : Savoir concevoir et utiliser des listes chaînées
— CD901 : Savoir 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.

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.

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

9.3 Utilisation : Liste ordonnée d'entiers

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

Chapitre 10

Arbre Binaire de Recherche (ABR)

Compétences évaluées

- CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- CD202 : Savoir identifier et résoudre le problème des cas récursifs
- CD403 : Savoir concevoir et utiliser des arbres (binaires, n-aires)
- CD601 : Savoir concevoir des collections à l'aide de SDD
- CD602 : Connaître les algorithmes d'insertion et de suppression (naïfs et AVL) dans un arbre binaire de recherche
- CD901 : Savoir 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.

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`.

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`.
2. Montrer que les simples et doubles rotations conservent la propriété d'un ABR (en considérant que l'arbre ne contient pas de doublons).
3. Expliciter la procédure d'équilibrage d'un arbre qui aurait deux sous-arbres équilibrés mais qui pourrait ne pas être équilibré.
4. Expliciter la procédure d'insertion : `inserer`.
5. Expliciter la procédure de suppression : `supprimer`.

Chapitre 12

Graphes

Compétences évaluées

- AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple
- AN201 : Savoir identifier les dépendances d'un TAD
- AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
- AN204 : Savoir formaliser des opérations d'un TAD
- AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
- AN206 : Savoir formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD
- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir concevoir une signature (préconditions incluses)
- CP005 : Savoir choisir un passage de paramètre (E, S, E/S)
- CD201 : Savoir identifier et résoudre le problème des cas non récursifs
- CD202 : Savoir identifier et résoudre le problème des cas récursifs
- CD801 : Savoir concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)
- CD804 : Connaître 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`.

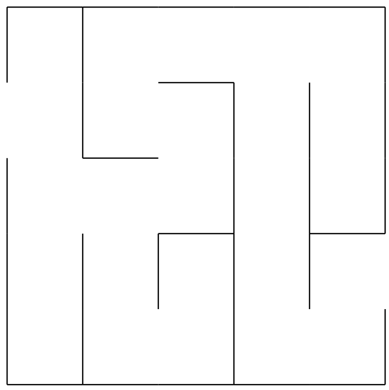


FIGURE 12.1 – Un labytinthe

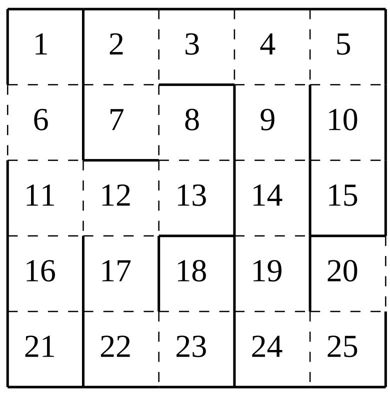


FIGURE 12.2 – Un labytinthe composé de cases

Le TAD labyrinthe

Les opérations disponibles sur un labyrinthe sont les suivantes :

- 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`
2. Donnez le TAD `Labyrinthe`

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>**)**

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.

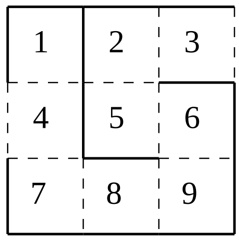


FIGURE 12.3 – Un labytinthe composé de 9 cases

Représentation du graphe

Proposez la matrice d'adjascence du graphe précédent.

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.

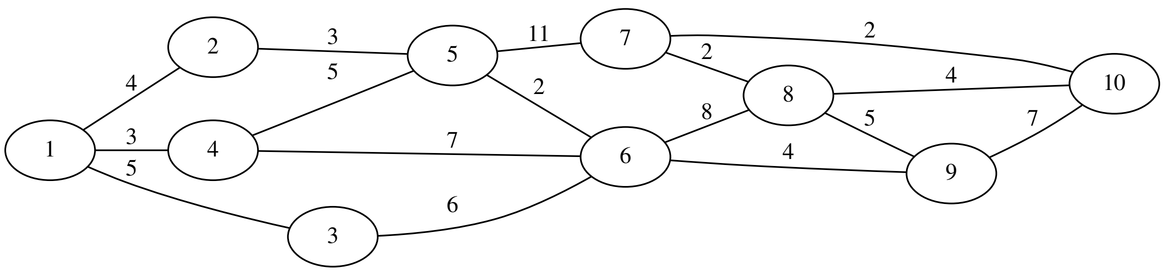


FIGURE 12.4 – Un graphe valué positivement

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.

Algorithme

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

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)

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 **??** 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

,,

,,,,

Chapitre 13

Programmation dynamique

Compétences évaluées

- AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple
- CD701 : Savoir définir la programmation dynamique
- CD702 : Savoir appliquer la programmation dynamique pour des cas simples
- CD801 : Savoir concevoir des graphes (matrice d'adjacence, matrice d'incidence, liste d'adjacence)

13.1 L'algorithme de Floyd-Warshall

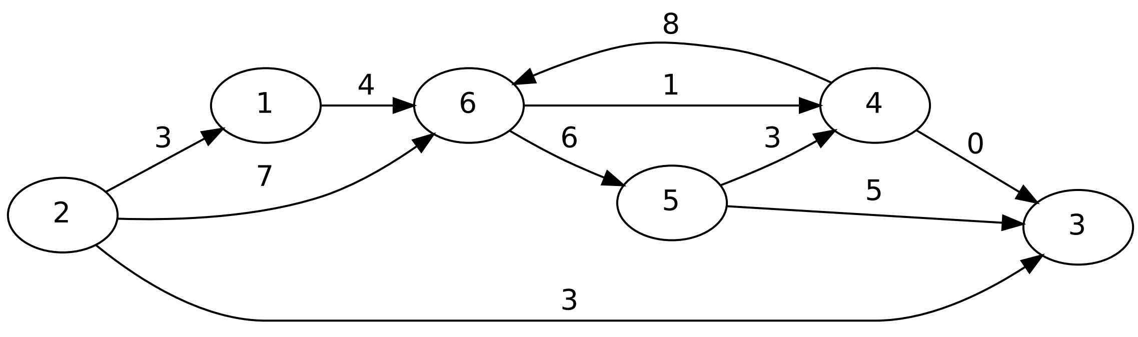


FIGURE 13.1 – Un graphe orienté valué

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 :

$$M_{i,j}^k = min(M_{i,j}^{k-1}, M_{i,k}^{k-1} + M_{k,j}^{k-1}). \gg [1]$$

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) ;

— $M_{i,i}^0 = 0$;

— $M_{i,j}^0 = +\infty$ 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.

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"`.

$$m = \begin{array}{c} \\ \llcorner \\ t \\ o \\ i \\ l \\ e \end{array} \begin{array}{c} \begin{array}{cccccccc} \llcorner & v & o & i & t & u & r & e \end{array} \\ \left(\begin{array}{cccccccc} \; & \; & \; & \; & \; & \; & \; & \; \\ \; \\ \; \\ \; \\ \; \\ \; \end{array} \right) \end{array}$$

2. À quel paradigme de conception appartient cet algorithme ? Justifiez.
3. Donnez l'algorithme de la fonction qui permet de calculer la distance de Levenshtein entre deux mots.

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