PREAMBULE

L'essentiel du cours d'algorithmique est noté dans ce document. Composé de 12 chapitres principaux, celui-ci est accompagné d'une partie complémentaire nommé Chapitre 0 qui concerne une (pseudo) convention d'écriture algorithmique retenue pour ce cours.
L'écriture des langages informatiques est normalisée, mais l'écriture algorithmique ne dispose pas d'une convention officielle, et chaque auteur de publications du domaine opte pour telle ou telle formalisme.
Les enseignants en font autant, nous vous présentons un formalisme dans ce chapitre 0, accompagné des arguments explicites qui nous ont amenés à prendre ces décisions, tant sur le fond que sur la forme.
Il est, par conséquent, vivement recommandé de commencer la lecture de ce chapitre avant même de débuter le cours par le chapitre 1. Les enseignants ne le présenteront pas en cours, mais celui-ci pourra servir de support à des explications complémentaires ou bien pour répondre à des questions complémentaires. Ensuite, au fur et à mesure de l'avancé du cours, sera intéressant de revenir sur ce chapitre 0 afin de compléter les informations acquises.
Pour les élèves qui ont déjà des connaissances en programmation, qui maîtrisent un langage, … ou qui ne désirent pas programmer plus tard, ce cours est malgré tout un concentré de connaissances fondamentales pour un informaticien, quelle que soit sa spécialité.

Bon courage …

Généralités sur l'Algorithmique

Définitions

Algorithmique

Suite finie de règles à appliquer, dans un ordre déterminé, à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données (sources : E. Universalis).
Suite d'opérations nécessaires et suffisantes à l'accomplissement d'une tâche.
Ces opérations se nomment en informatique des « INSTRUCTIONS ».

Algorithme d'assemblage d'un produit fini

Algorithme d’Euclide (calcul du PGCD)
Algorithme de résolution d’une équation …

Historique

Historiquement, les langages font parties des sciences très anciennes visant à transmettre des moyens efficaces pour obtenir des résultats en partant d’éléments donnés.

Par la suite, cela représentait tout procédé de calcul systématique.
Inventé par le grand mathématicien Abu Ja'far Muhammad ibn Musa al-Khawarizmi.
Vraisemblablement né en 780 à Bagdad, il est mort vers 850, D'autres sources voient son lieu de naissance en Ouzbékistan, au sud de la mer d'Aral, dans la ville de Khawarizm
En informatique moderne : on le désigne par un procédé automatique (autonome) et effectif, comportant une description (finie) des entrées, des sorties, et des tâches élémentaires à réaliser.

Réalisation d’un programme

PROCEDURE de REALISATION d'un PROGRAMME INFORMATIQUE :

L'écriture algorithmique est une phase intermédiaire et indispensable pour réaliser un programme.

La qualité du développement final dépend aussi de cette phase cruciale.

La suppression de celle-ci, perçue comme un “gain de temps" est généralement la cause d’un accroissement de la durée de développement, du fait de la non prise en compte de la complexité du problème au bon moment.

[La taille des « pavés » permet de connaître le niveau d’importance (le volume) des phases de développement. Il faut aussi noter l’ordre des phases, l’incidence des premières phases sur les suivantes montre bien que tout projet doit être engagé en ne négligeant aucune phase.]

Notion d’instruction

Notion d'instruction et notion de donnée

Une instruction est un ordre élémentaire - au sens algorithmique -, que peut exécuter un programme. Les données manipulées par les instructions sont : des variables proprement dites (x, y, …) des « variables » constantes (pi, …) des valeurs littérales ("bonjour", 45, VRAI) des expressions booléennes ou complexes (combinaisons de variables, constantes et valeurs littérales avec des opérateurs)
Notion de PRIMITIVE, ou fonction élémentaire

Deux primitives d'entrée-sortie :
LIRE() : lecture de la frappe au clavier
ECRIRE() : affichage à l'écran

L'AFFECTATION, pour une "mise en mémoire" :
le symbole d'affectation : <--

Notion de séquences

Suite d'opérations élémentaires (ou ACTIONS) dont l'exécution est séquentielle.
Chaque pavé représente ici une instruction.

La lecture d'un algorithme s'effectue de haut en bas.

Attention, les représentations graphiques du type “organigramme” sont limitées aux seules structures algorithmiques et ne sont pas employées pour modéliser des ensembles modulaires (programmes).

Notion de ruptures de séquences

L'exécution des opérations élémentaires ACTIONS 2a et ACTIONS 2b n'est pas systématique.
Chacune d’elle dépend de la "condition" de la rupture de séquence.
La réalisation des ruptures de séquence s'effectue en fonction des éléments transmis à la "condition".

L'exécution de l’opération élémentaire ACTION 2 peut éventuellement faire l’objet d’une répétition.
L’action 2 peut avoir une occurrence nulle, égale à 1 ou encore une occurrence n.
Elle dépend d'une "condition" qui permet ou non la répétition.

Conventions d’écriture

Les commentaires sont précédés de : //
ou /* */
Les mots clés et Primitives sont écrits en majuscule : VAR:
Les chaînes de caractères sont encadrés par : "
Les caractères uniques sont encadrés par : '
Les accents sont seulement autorisés :
dans les lignes de commentaire
dans les chaînes de caractères : "élève"
pour les caractères uniques : 'à'

Présentation

Concevoir un programme, c'est mettre en œuvre des mécanismes intellectuels complexes qui nécessitent la maîtrise des concepts intégrés à l'utilisation de l'ordinateur sur lequel le logiciel doit fonctionner.

Ces concepts sont nombreux et situés à différents niveaux : du plus concret au plus abstrait.

Mais, c'est aussi : appliquer avec rigueur un nombre important de règles essentielles pour parvenir à notre fin ; un logiciel qui fonctionne dans tous les cas de figure, et qui répond le plus justement au problème posé.

Pour cela, il faudra structurer la technique de résolution en fonction de trois éléments de base en informatique …

Les trois structures

La structuration des données

Un programme nécessite l'emploi systématique de données ou d'ensembles de données :
Des données élémentaires (structures les plus basiques)
[Modules 2]
Des structures évoluées (structures complexes)
[Modules 4 - 6 - 7 - 9]

La structuration algorithmique

La suite d'opérations à réaliser pour résoudre un problème nécessite un minimum d'organisation :
Des séquences (suite successive d'opérations réalisées sans condition)
[Module 3]
Des ruptures de séquences (opérations conditionnées réalisées ou non suivant un contexte inconnu lors de la programmation)
[Module 3 - 5]

La structuration du programme
[Module 8]

Module 0: Convention d'écriture algorithmique

Intérêt d'une convention
Conventions de mise en forme
Conventions liées à la sémantique des programmes
Conventions liées aux concepts avancés

Module 1: Généralités sur l'Algorithmique

Présentation
Les trois structures
Présentation des modules
Objectifs du cours

Module 2: Données et Structures de données
Types de données
Opérations élémentaires
Opérateurs associés à chaque type
Accès aux données en mémoire

Module 3: Structures algorithmiques élémentaires
Présentation
Structures séquentielles
Structures de sélection
Structures de répétition

Module 4: Structures linéaires de données I
Présentation
Structures indexées à une dimension
Structures indexées à plusieurs dimensions
Structures indexées dynamiques

Module 5: Structures algorithmiques imbriquées

Présentation
Structures de sélection imbriquées
Structures de répétition imbriquées

Module 6: Structures non linéaires de données I
Présentation
Structure abstraite de données: Enregistrement simple
Structure abstraite de données: Enregistrement imbriqué

Module 7: Structures linéaires de données II
Présentation
Structure abstraite de données: Liste
Structure abstraite de données: Pile
Structure abstraite de données: File

Module 8: Structure de programme
Présentation
Programme principal, fonction et procédure
Récursivité

Module 9: Structures non linéaires de données II
Présentation
Structure abstraite de données: Graphe
Structure abstraite de données: Arbre

Module 10: Programme et Système d'exploitation
Présentation
Flux d'entrées/sorties
Structures de données: Fichiers
Système d'exploitation: Gestion des erreurs

Module 11: Complexité Algorithmique: Introduction
Complexité d'un algorithme
Ecriture en O()
Classes de complexité

Module 12: Résolution d'un problème
Approche de résolution globale d'un problème
Résoudre un problème en informatique
Qualités d'un programme informatique

Données et structures de données

Présentation

Les structures de données élémentaires sont des informations de base dont on a besoin en informatique, et qui seront stockées en mémoire, et traitées par des algorithmes.
Il faudra distinguer :
les structures élémentaires qui sont des types concrets de données,
des structures avancées (simples ou complexes) qui sont de véritables structures de données (suite organisées de données élémentaires) ou types abstraits de données.

Correspondances de type

Les types élémentaires de données ont une définition proche de celle étudiée en mathématique.
En algorithmique, on limite volontairement les types élémentaires à la liste suivante :
R : Le type réel, noté :REEL

Z : Le type entier (relatif), noté :ENTIER
Le type booléen, noté :BOOLEEN
Le type caractère (unique), noté :CARACTERE
Dans les langages, on retrouve les types correspondants :
float, real, double, int, integer, bool, char, …

Cinq types élémentaires

Lors de l'emploi de ces différents types de données, au coeur des algorithmes, on ne se préoccupera pas :
Du domaine des valeurs affectées aux données, contrairement aux langages, pour lesquels il est possible de définir si une donnée ne sera que positive ou nulle.
De la taille des valeurs affectées, contrairement aux langages, pour lesquels il est impératif de connaître la plus grande valeur affectée en cours d'utilisation de l'algorithme pour préciser son type.
Par ailleurs, un type élémentaire est ajouté à la liste présentée ci-dessus, dans le but de travailler sur des chaînes de caractères, extension du type CARACTERE :
Le type suite de caractères :CHAINE

Les données élémentaires utilisées en algorithmique :
ENTIER
REEL
CARACTERE
CHAINE (compléments d'informations, module 5)
BOULEEN

Les structures avancées :
à suivre [Module 3 et Module 5]

Le type ENTIER

ENTIER
nombres entiers
négatifs
positifs
nuls
Exemples :
33
- 13
[pas de limite de taille, ni de définition de domaine relatif ou non, (contrairement aux langages)]

Le type REEL

REEL
nombres entiers
fractionnés
négatifs

positifs
nuls
Exemples :
3,26
-10,5
[pas de limite de taille, ni de définition de domaine relatif ou non, (contrairement aux langages)]

Le type CARACTERE

CARACTERE (ou CAR)
caractère unique
Exemples :
'F'
'j'
'!'
'' représente aucun caractère
[référence : table des codes ASCII]
apostrophe simple

Le type CHAINE

CHAINE
caractère unique
suite de caractères
Exemples :
"Bonjour"
"586"
"ENSET"
"" représente aucun caractère
[référence : table des codes ASCII]
apostrophe double
*le type chaîne est une concaténation de caractères du type CARACTERE

Le type BOOLEEN

BOOLEEN* **
VRAI
FAUX

*pour simplifier, le type BOOLEEN est une représenté à l'aide des deux termes ci-dessus, en
réalité, le compilateur fait correspondre un 1 pour VRAI et un 0 pour FAUX, (voire 0 pour FAUX et tout autre valeur représente VRAI) **même s'il semble logique de trouver une correspondance avec un élément « bit », sa taille en mémoire vaut, en générale, un octet !

Les structures de données évoluées

A l'aide des structures de données élémentaires, il va être possible de construire de

véritables structures de données complexes et organisées. La construction du type CHAINE, sera représentée comme une suite de caractères : une LISTE. Ce type abstrait est une structure linéaire.
D'autres constructions seront des structures hiérarchisées (des ENREGISTREMENTS ou des ARBRES).
D'autres structures représenteront des relations plus complexes (des GRAPHES) :Ces deux derniers types abstraits sont des structures non linéaires.

Parmi les structures linéaires :
Les listes (structure construite à l'aide de chaînages)
Les tableaux (structures indicées)
Les piles et les files (listes dont les accès sont limités)

Parmi les structures non linéaires :
Les enregistrements (structure contenant des champs)
Les arbres (structure hiérarchisée)
Les graphes (structure organisée par des relations)

Présentation

Termes les plus courants employés dans cette partie du cours pour définir les opérations élémentaires sur les données :
Déclaration (réservation d'emplacement mémoire)
Initialisation (écriture initiale)
Affectation (écriture avec perte de l'ancienne donnée)
Utilisation (lecture)
Réaffectation (terme spécialement employé ici afin de présenter la possibilité de modifier une donnée à l'aide d'elle-même)

[Le terme initialisation est employé lorsque la donnée est affectée la toute première fois.]
Opérateur commun à tous les types
Les structures élémentaires de données Un symbole commun à toutes les structures de données permettra d'affecter, de réaffecter, d'initialiser les différents types de données étudiés dans ce chapitre. à suivre : partie 3 de ce module, Opérations élémentaires

Le type ENTIER

Déclaration
nom_de_la_donnee :ENTIER
Initialisation (avec déclaration)
nom_de_la_donnee <-- valeur :ENTIER
Affectation (hors déclaration)
nom_de_la_donnee <-- valeur
Utilisation
autre_donnee <-- nom_de_la_donnee

Réaffectation
nom_de_la_donnee <-- nom_de_la_donnee + 1

Le type REEL

Déclaration
nom_de_la_donnee :REEL
Initialisation (avec déclaration)
nom_de_la_donnee <-- valeur :REEL
Affectation (hors déclaration)
nom_de_la_donnee <-- valeur
Utilisation
autre_donnee <-- nom_de_la_donnee
Réaffectation
nom_de_la_donnee <-- nom_de_la_donnee * 0,5

Le type CARACTERE

Déclaration
nom_de_la_donnee :CARACTERE
Initialisation (avec déclaration)
nom_de_la_donnee <-- ‘a’ :CARACTERE
Affectation (hors déclaration)
nom_de_la_donnee <-- ‘a’
Utilisation
autre_donnee <-- nom_de_la_donnee
Réaffectation
nom_de_la_donnee <-- nom_de_la_donnee + 1

* Cas particulier d’une réaffectation pour une variable de type CARACTERE. Celui-ci est considéré par le compilateur comme une variable entière d’un octet correspondant à la table
du codage ASCII. Ainsi, ‘a’ + 1 correspond à ‘b’.

Le type CHAINE

Déclaration
nom_de_la_donnee :CHAINE
Initialisation (avec déclaration)
nom_de_la_donnee <-- “valeur” :CHAINE
Affectation (hors déclaration)
nom_de_la_donnee <-- “valeur”
Utilisation
autre_donnee <-- nom_de_la_donnee

Le type BOOLEEN

Déclaration
nom_de_la_donnee :BOOLEEN

Initialisation (avec déclaration)
nom_de_la_donnee <-- VRAI :BOOLEEN
nom_de_la_donnee <-- FAUX
Utilisation
autre_donnee <-- nom_de_la_donnee
Réaffectation
nom_de_la_donnee <-- nom_de_la_donnee + 1

Constantes ou variables

Les Constantes : CONST
Déclaration avec initialisation obligatoire
Utilisation (lecture seule)

Les Variables : VAR
Déclaration
Initialisation dans la déclaration
Affectation (écriture)
Utilisation
Réaffectation

Opérateurs pour le type ENTIER

Les structures élémentaires de données
Un critère de priorité est associé à chaque opérateur. Afin de faciliter la lecture et d'éviter les erreurs, nous utiliserons systématiquement des parenthèses. (voir le référentiel des langages de programmation normalisés)

Opérateurs pour le type REEL

Les structures élémentaires de données
Un critère de priorité est associé à chaque opérateur. Afin de faciliter la lecture et d'éviter les erreurs, nous utiliserons systématiquement des parenthèses. (voir le référentiel des langages de programmation normalisés)

Opérateurs pour le type CARACTERE

Les structures élémentaires de données
Un critère de priorité est associé à chaque opérateur. Afin de faciliter la lecture et d'éviter les erreurs, nous utiliserons systématiquement des parenthèses. (voir le référentiel des langages de programmation normalisés)

Opérateurs pour le type CHAINE

Les structures élémentaires de données
Un critère de priorité est associé à chaque opérateur. Afin de faciliter la lecture et d'éviter les erreurs, nous utiliserons systématiquement des parenthèses. (voir le référentiel des langages de programmation normalisés)

Opérateurs pour le type BOOLEEN

Les structures élémentaires de données
Un critère de priorité est associé à chaque opérateur. Afin de faciliter la lecture et d'éviter les erreurs, nous utiliserons systématiquement des parenthèses. (voir le référentiel des langages de programmation normalisés)

Présentation

Accès à une donnée en mémoire, son type et sa taille
La déclaration d'une donnée élémentaire en mémoire :
un_entier <-- 5720 :ENTIER
a pour effet de réserver et d'initialiser la valeur associée à cette instruction à un emplacement
mémoire de taille prédéfinie par le compilateur employé.
Cette opération, effectuée par le système d'exploitation, exploite la mémoire par le biais d'adresse, normalement écrite en hexadécimal.
Si l'accès « direct » à la donnée est le moyen présenté jusque là, l'accès « indirecte », via l'adresse, reste une technique courante (avec ses avantages et ses inconvénients).

Le type POINTEUR

Un type particulier de donnée est introduit ici : Le Pointeur.
Le pointeur est une variable qui contient une adresse mémoire :
La déclaration s'effectue sous la forme :
*ptr_entier :ENTIER //pointeur sur un entier
Le symbole * signifie ici, pour la déclaration simple, que l'identificateur noté
en suivant est du type : pointeur.
La déclaration avec initialisation s'effectue sous la forme :
*ptr_entier <-- 7 :ENTIER
Le symbole * signifie toujours que l'identificateur noté en suivant est du type : pointeur.
Mais l'initialisation opérée est bien la valeur pointée, et non l'adresse. Lors d'une déclaration d'un pointeur, l'opérateur * indique au compilateur la présence d'une variable pointeur, ce qui ne signifie pas la même chose que pour une affectation,
…
L'accès à la variable se fait grâce à l'opérateur * :
*ptr_entier <-- 7 //écriture
nombre_entier <-- *ptr_entier //lecture
Le symbole * signifie ici, dans le cas de l'utilisation de la variable, que l'accès correspond à la valeur et non à l'adresse.
L'accès aux adresses s'effectue à l'aide de l'opérateur & :
&ptr_entier
&nombre_entier
Mémoire Vive (RAM), Rappel :
Les programmes ne fonctionnent qu'avec cette mémoire.
Elle est découpée en octets référencés par des adresses.

Identificateur, adresse et mémoire

Accès à une donnée en mémoire, son type et son adresse

Déclaration
un_entier :ENTIER

Affectation
un_entier <-- 5720

Valeur de l'adresse
&un_entier //04DFF5C2

Utilisation
autre_entier :ENTIER
autre_entier <-- un_entier

Un pointeur, le type associé et sa valeur, ses adresses

Déclaration
*ptr_entier :ENTIER

Affectation
*ptr_entier <-- 5720

Valeur des adresses
ptr_entier //04DFF5C2
&ptr_entier //04DFF5C8

Utilisation
un_entier :ENTIER
un_entier <-- *ptr_entier

Accès direct à une donnée en mémoire ou via un pointeur

Déclaration
un_entier <-- 5720:ENTIER
*ptr_entier :ENTIER

Affectation
&un_entier //04DFF5C2
ptr_entier <-- &un_entier
ptr_entier //04DFF5C2
&ptr_entier //04DFF5C8
*ptr_entier //5720

Affectation indirecte

*ptr_entier <-- 1258
un_entier //1258

Réservation de la mémoire

Rappel:
En algorithmique, on ne se préoccupe pas de la taille réservée en mémoire par une donnée. En revanche, ce travail doit être impérativement réalisé lors de la phase de traduction dans un langage de programmation.

Portée d'une variable

Historiquement, les variables d'un programme étaient accessibles par tout ou partie d'un logiciel.
Cela représentait un avantage important pour développer un programme, car le concepteur n'avait qu'à définir une série d'identificateurs de variables, leur type, et le processus de compilation était très simple.
Mais les inconvénients sont nombreux :
La lecture des données, et leur modification étaient permises depuis n'importe quelle partie des programmes.
Les logiciels conséquents qui exploitent un très grand nombre de données deviennent extrêmement difficiles à modifier.
Les langages ont donc évolué en intégrant des modèles qui protègent les données grâce à une
très forte structuration des ensembles de données et l'introduction de la programmation modulaire.

Lors de la déclaration des données qui seront utilisées par le programme, il y a réservation de l'espace mémoire pour les variables dédiées au fonctionnement du programme principal ou à chacun des sous programmes. Mais il est possible, aussi, d'effectuer une réservation, non dédiée, de variables dites : GLOBALES.
Les variables globales ont la possibilité d'être exploiter par l'ensemble du programme et les modules appelés, (c'est-à-dire en lecture et en écriture), on parle de la portée des données ou d'accessibilité.
L'emploi de ce type de données est déconseillé, car il permet de passer au travers de la hiérarchisation de la structure du programme, ce qui est contradictoire avec des objectifs de qualité et de sécurité.
Chaque programme et sous-programme a son propre espace de variables « temporaires »,inaccessibles aux autres parties, ces variables sont dites : LOCALES.
L'accès aux valeurs enregistrées peut se faire via adressage (utilisation d'un pointeur), ce qui permet le limiter ou non ces accès à tous les constituants du programme.

Types de données

Quels types de donnée élémentaire peuvent recevoir chacune des informations suivantes ? (l'affectation se fait à l'aide d'une unique variable)

1. La taille d’un individu.
2. Le numéro de téléphone, y compris les points de séparation pour chaque couple de chiffre.
3. Le prix d’un timbre.
4. Un code postal.
5. La réponse relative à la parité d’un nombre entier.
6. La date de naissance constituée au maximum de six chiffres.
7. Le résultat d’une soustraction de deux variables entières.
8. L’identité d’une carte à jouer.
9. Le nom et le prénom d’un individu.
10. Le volume, en litres, d’une piscine.
11. Le numéro d’un département français.
12. La réponse à une question du type : « Voulez-vous continuer ? ».
13. Le résultat d’une division quelconque de deux variables entières.
14. Les initiales d’une personne.
15. Le numéro d’un dé à jouer.

Opérations élémentaires

Quelle est l’opération élémentaire réalisée pour chaque instruction suivante ou groupe d’instructions ?
Et/Ou
Quels types de donnée élémentaire peuvent recevoir chacune des informations suivantes ?
(ATTENTION, des erreurs peuvent se cacher derrière ces opérations !)

1. `nom_de_la_donnee <-- 1024`

2. `nom_de_la_donnee :REEL`

3. `nom_de_la_donnee <-- 10,5 :ENTIER`

4. ```
   autre_donnee :ENTIER
   nom_de_la_donnee <-- 3,85 :REEL
   autre_donnee <-- nom_de_la_donnee
   ```

5. `nom_de_la_donnee <-- nom_de_la_donnee + 1`

6. `nom_de_la_donnee <-- FAUX`

7. `nom_de_la_donnee <-- ‘ ’`

8. `autre_donnee <-- nom_de_la_donnee`
   ```

9.
```
nom_de_la_donnee <-- 5 :REEL
nom_de_la_donnee <-- nom_de_la_donnee * 0,5
```

10.
```
nom_de_la_donnee :CHAINE
nom_de_la_donnee <-- "2008"
```

11.
```
autre_donnee :CHAINE
nom_de_la_donnee <-- 'n' :CARACTERE
nom_de_la_donnee <-- 'N'
autre_donnee <-- nom_de_la_donnee
```

12.
```
autre_donnee :BOOLEEN
nom_de_la_donnee :BOOLEEN
autre_donnee <-- nom_de_la_donnee
```

13.
```
nom_de_la_donnee <-- VRAI :BOOLEEN
```

14.
```
nom_de_la_donnee <-- 'A':CARACTERE
nom_de_la_donnee <-- nom_de_la_donnee + 1
```

Opérateurs associés à chaque type

Quelle est l'opération élémentaire réalisée pour chaque instruction suivante ou groupe d'instructions ?
Et/Ou
Quels types de donnée élémentaire peuvent recevoir chacune des informations suivantes ?
(ATTENTION, des erreurs peuvent se cacher derrière ces opérations !)

1.
```
autre_donnee :ENTIER
nom_de_la_donnee <-- 103,55 :REEL
autre_donnee <-- DIV( nom_de_la_donnee, 8)
```

2.
```
autre_donnee :BOOLEEN
nom_de_la_donnee <-- 15 :ENTIER
autre_donnee <-- ( nom_de_la_donnee > 12 )
```

3.
```
autre_donnee :ENTIER
nom_de_la_donnee <-- 27 :ENTIER
autre_donnee <-- MOD( nom_de_la_donnee, 2)
```

4.
```
autre_donnee :BOOLEEN
nom_de_la_donnee <-- "ENSET" :CHAINE
autre_donnee <-- ( nom_de_la_donnee = "enset" )
```

5. ```
   autre_donnee <-- "ENSET" :CHAINE
   nom_de_la_donnee :CARACTERE
   nom_de_la_donnee <-- EXTRACTION( autre_donnee, 1)
   ```

6. ```
   autre_donnee <-- "ENSET" :CHAINE
   nom_de_la_donnee <-- 15 :ENTIER
   nom_de_la_donnee <-- LONGUEUR( autre_donnee)
   ```

7. ```
   nom_de_la_donnee <-- CONCATENER("ENSET","2009")
   ```

Accès aux données en mémoire

Quelle est l'opération élémentaire réalisée pour chaque instruction suivante ou groupe d'instructions ?
Et/Ou
De quel type est la donnée pour chacune des informations suivantes ?
(ATTENTION, des erreurs peuvent se cacher derrière ces opérations !)

1. ```
   *nom_de_la_donnee <-- f
   ```

2. ```
   *nom_de_la_donnee :REEL
   ```

3. ```
   &nom_de_la_donnee :ENTIER
   ```

4. ```
   *nom_de_la_donnee :ENTIER
   autre_donnee <-- 117 :ENTIER
   *nom_de_la_donnee <-- autre_donnee
   ```

5. ```
   autre_donnee <-- *nom_de_la_donnee + 1
   ```

6. ```
   *nom_de_la_donnee <-- '?' :CARACTERE
   ```

7. ```
   &nom_de_la_donnee <-- "ENSET"
   ```

Structures algorithmiques élémentaires

Présentation

Structure algorithmique

La structure algorithmique concerne la succession et les imbrications de séquences et/ou ruptures de séquences.
Elle forme ainsi une partie d'un programme.
C'est aussi la partie la plus importante du programme d'algorithmique.

Les différentes structures
Les catégories et sous-catégories de structures algorithmiques
On distingue deux grandes familles de structures :
Les structures dont le mode opératoire est purement séquentiel. Elles seront nommées ici : les séquences.
Les structures dont le mode opératoire est conditionné. Elles seront nommées ici : les ruptures de séquences.
Parmi les ruptures de séquences, on distingue encore deux grandes catégories :
Les ruptures de séquences "de sélection" (elles ne font pas appel à la répétition):
SI
CASPARMI
Les ruptures de séquences "répétitives" :
TANTQUE
REPETER
POUR

Séquences : suite d'instructions exécutées dans l'ordre d'écriture

Ruptures de séquences : toutes les ruptures de séquences nécessitent une condition ordonnant
ou non l'exécution des instructions situées dans le corps de l'instruction de rupture de séquence

Rupture de séquence de sélection (non répétitive):
SI
CASPARMI
Rupture de séquence répétitive :
TANTQUE
REPETER
POUR

Les instructions de sélection

Instruction de sélection (non répétitive):

SI..ALORS..FINSI
SI..ALORS..SINON..FINSI
CAS..PARMI..
CAS..PARMI..PARDEFAUT
Les instructions de répétition

Instruction de répétition :

TANTQUE..FAIRE..FINTANTQUE
REPETER..JUSQU'A..
REPETER..TANTQUE..
POUR..FAIRE..FINPOUR

Notion de séquences

Suite d'instructions traitées dans l'ordre d'écriture
(aucune rupture de séquences):

Exemple d'algorithme séquentiel

Notion de ruptures de séquences

L'exécution des opérations élémentaires ACTIONS 2a et ACTIONS 2b n'est pas systématique.
Chacune d'elle dépend de la "condition" de la rupture de séquence.La réalisation des ruptures de séquence s'effectue en fonction des éléments transmis à la "condition".

Instructions de sélection et variantes

Ruptures de séquences :

Instruction de sélection (non répétitive):

SI..ALORS..FINSI
SI..ALORS..SINON..FINSI
CAS..PARMI..
CAS..PARMI..PARDEFAUT

Instruction de sélection « Si »

```
ALGORITHME: compteur_rebours
//BUT : affiche la valeur d'un compteur à rebours
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : la valeur du compteur à rebours
VAR:
```

```
  x   :ENTIER

DEBUT
  x <-- LIRE() //affectation d'une valeur par l'util.
  SI( x < 0 )  //si cette valeur est négative
  ALORS        //alors cette condition est vraie
    x <-- -x   //et la variable se trouve réaffectée
  FINSI
  ECRIRE( "La val.abs. du nombre saisi vaut : ",x)
FIN
```

```
ALGORITHME: max_deux_nombres
//BUT : recherche la valeur max. parmi 2 valeurs saisies
//ENTREE : deux réels saisis par l'utilisateur
//SORTIE : le maximum des deux valeurs
VAR:
  a, b, max  :REEL

DEBUT
  a <-- LIRE()
  b <-- LIRE()
  SI( a > b )
  ALORS
    max <-- a
  SINON
    max <-- b
  FINSI
  ECRIRE( "Le Max entre ",a," et ",b," est : ",max)
FIN
```

Instruction de sélection « Cas parmi »

```
ALGORITHME: nom_mois
//BUT : affiche le nom du mois en fonction du numéro
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : le nom du mois correspondant au chiffre saisi
VAR:
  nombre  :ENTIER

DEBUT
  mois <-- LIRE()
  CAS( mois  )PARMI:
    CAS1:  1  ECRIRE( "Janvier" )
    CAS2:  2  ECRIRE( "Février" )
    …
    CAS12:  12  ECRIRE( "Décembre" )
```

```
  FINCASPARMI
FIN
```

```
ALGORITHME: nom_mois
//BUT : affiche le nom du mois en fonction du numéro
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : le nom du mois correspondant au chiffre saisi
VAR:
  nombre  :ENTIER

DEBUT
  LIRE( mois )
  CAS( mois  )PARMI:
    CAS1:  1  ECRIRE( "Janvier" )
    CAS2:  2  ECRIRE( "Février" )
    …
    CAS12:  12  ECRIRE( "Décembre" )
    PARDEFAUT:  ECRIRE( "Erreur de saisie !" )
  FINCASPARMI
FIN
```

Notion de ruptures de séquences

L'exécution de l'opération élémentaire ACTION 2 peut éventuellement faire l'objet d'une répétition.
L'action 2 peut avoir une occurrence nulle, égale à 1 ou encore une occurrence n.
Elle dépend d'une "condition" placée en début de bloc qui permet ou non la répétition.

L'exécution de l'opération élémentaire ACTION 2 est effectuée une fois et peut éventuellement faire l'objet d'une répétition.
L'action 2 peut avoir une occurrence égale à 1 ou encore une occurrence n.
Elle dépend d'une "condition" placée en fin de bloc qui permet ou non la répétition.

Instructions de répétition et variantes

Ruptures de séquences :

Instruction de répétition :

TANTQUE..FAIRE..FINTANTQUE
POUR..FAIRE..FINPOUR
REPETER..JUSQU'A..
REPETER..TANTQUE..

Instruction de répétition « Tantque »

```
ALGORITHME: compteur_rebours
//BUT : affiche la valeur d'un compteur à rebours
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : la valeur du compteur à rebours
VAR:
  nombre  :ENTIER

DEBUT
  nombre <-- LIRE()
  TANTQUE( nombre >=0 )
  FAIRE
    ECRIRE( "Valeaur du compteur à rebours: ", nombre )
    nombre <-- nombre - 1
  FINTANTQUE
FIN
```

Instruction de répétition « Pour »

```
ALGORITHME: compteur_rebours2
//BUT : affiche la valeur d'un compteur à rebours
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : la valeur du compteur à rebours
VAR:
  nombre  :ENTIER

DEBUT
  nombre <-- LIRE()
  POUR( compteur <-- nombre :ENTIER a 0 au pas de -1 )
  FAIRE
    ECRIRE("Valeur du compteur à rebours: ", compteur)
  FINPOUR
FIN
```

Instruction de répétition « Répéter »

```
ALGORITHME: compteur_rebours3
//BUT : affiche la valeur d'un compteur à rebours
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : la valeur du compteur à rebours
VAR:
  nombre  :ENTIER

DEBUT
  nombre <-- LIRE()
```

```
  REPETER
    ECRIRE("Valeur du compteur à rebours: ", nombre)
    nombre <-- nombre -1
    JUSQU'À( nombre < 0)
FIN



ALGORITHME: compteur_rebours4
//BUT : affiche la valeur d'un compteur à rebours
//ENTREE : un entier saisi par l'utilisateur
//SORTIE : la valeur du compteur à rebours
VAR:
  nombre  :ENTIER

DEBUT
  nombre <-- LIRE()
  REPETER
    ECRIRE("Valeur du compteur à rebours: ", nombre)
    nombre <-- nombre -1
    TANTQUE( nombre >= 0 )
FIN
```

Répondre à chacun des problèmes posés ci-dessous et construire un algorithme en respectant les contraintes et autres informations fournies.

Structure algorithmique imposée : SI..ALORS

Un élève entre la valeur d'une moyenne pour un examen, le programme indique s'il a réussi lorsqu'il obtient une note supérieure ou égale à 12.

Structure algorithmique imposée : SI..ALORS..SINON

En fonction du prix d'un produit HT et de la catégorie de celui-ci (luxe: TVA=19,6 % ; autre: TVA=5,5 %), le programme calcule sa valeur TTC.

Remarque sur le choix des variables (leur type) lorsque celles-ci sont affectées à l'aide d'informations provenant d'utilisateur, ou bien lorsque l'on effectue une opération arithmétique dont le résultat n'est pas du même type que la donnée initialement saisie.

Structure algorithmique imposée : CAS..PARMI

Lorsque que l'utilisateur entre le choix d'un menu, pour 1, il affiche : "Nom du fichier : Algo1.txt"; pour 2 : "Nom du répertoire : C:"; pour 3 : "Nom complet : C:\Algo1.txt".

Structure algorithmique imposée : CAS..PARMI..PARDEFAUT

En fonction du numéro du mois choisi par l'utilisateur, le programme affiche le nombre de

jours correspondant (les années bissextiles ne sont pas prises en compte).

Structure algorithmique imposée : TANTQUE

Le programme affiche systématiquement chaque touche que l'utilisateur frappe jusqu'à ce que la touche 'q' soit entrée ('q' correspond à « QUITTER »).

Remarque sur l'emploi d'une boucle afin d'exécuter un programme un nombre de fois non défini dans le code. (par exemple, pour le test d'un algorithme)

Structure algorithmique imposée : POUR

Le programme affiche un compteur à rebours pour indiquer le nombre de photocopies restantes, avant d'avoir lancé l'impression d'un fichier: "C:\Rapport.txt". (L'impression est réalisée après l'affichage. Noter seulement en commentaire la fonction d'impression).

Remarque sur l'emploi d'une boucle POUR qui, dans la pratique, ne devrait être employé que pour un domaine préalablement défini.

Structure algorithmique imposée : REPETER..JUSQU'A

Le programme affiche systématiquement chaque mot que l'utilisateur frappe jusqu'à ce que le mot "fin" soit saisi. Le programme s'arrête à ce moment-là, le mot "fin" étant le dernier à apparaître sur l'écran.

Structure algorithmique imposée : REPETER..TANTQUE

Le programme affiche systématiquement chaque mot que l'utilisateur frappe jusqu'à ce que le mot "fin" soit saisi. Le programme s'arrête à ce moment-là, le mot "fin" étant le dernier à apparaître sur l'écran.

Conversion EURO/FRANC et FRANC/EURO

L'utilisateur entre un prix et indique si celui-ci est en euro ou en franc.
Le programme effectue la conversion et l'affiche à l'écran.

Recherche du minimum entre deux nombres

L'utilisateur entre deux nombres et le programme indique lequel des deux est le plus petit.

Calcul de la somme ou du produit entre deux nombres

Cet algorithme affiche la somme ou le produit de 2 nombres, suivant le choix de l'utilisateur.

Celui-ci doit saisir deux nombres ainsi qu'un caractère représentant l'opération à effectuer (s : somme, p : produit).

Recherche le(s) mois correspondants

Cet algorithme affiche les mois (en lettres) qui correspondent au nombre de jours qui a été saisi par l'utilisateur.

Par exemple, lorsque l'utilisateur entre la valeur : 31, le programme affiche « Janvier, Mars, Mai, … », soit tous les mois de 31 jours.

Calcul de la puissance

Cet algorithme affiche le résultat du calcul de la puissance de 2 entiers, le second, entré au clavier, est l'exposant.

Table de multiplication inverse

Cet algorithme affiche la table de multiplication du chiffre 5 dans l'ordre inverse (c'est-à-dire en commençant par 10 et terminant par 0).

Calcul de l'aire d'un cercle

Cet algorithme affiche le résultat issu du calcul de l'aire d'un cercle.
L'utilisateur entre une valeur de longueur correspondant au rayon du cercle.

Convertisseur binaire/décimal

Cet algorithme affiche le résultat de la conversion d'une suite binaire en décimale.
L'utilisateur entre la valeur de 8 variables binaires (bit0 à bit8), et le programme se charge de retrouver la valeur décimale correspondante.

Exemple :
bit0 = 1 --- 1
bit1 = 0 --- 2
bit2 = 1 --- 4
bit3 = 0 --- 8
bit4 = 0 --- 16
bit5 = 0 --- 32
bit6 = 0 --- 64
bit7 = 0 --- 128

Ce qui correspond à : bit0 x 1 + bit2 x 4 = 5 (décimal)

Convertisseur des angles décimaux en degré/minute/seconde

Cet algorithme affiche le résultat de la conversion de la valeur d'un angle décimal en degré/minute/seconde.

Convertisseur des angles degré/minute/seconde en décimaux

Cet algorithme affiche le résultat de la conversion de la valeur d'un angle degré/minute/seconde en valeur décimale.

Calcul de la masse molaire moléculaire

Cet algorithme affiche la valeur de la masse molaire moléculaire d'un composant constitué de carbone, d'hydrogène et d'oxygène.
C = 12 mg/mol
H = 1 mg/mol
O = 16 mg/mol
Exemple :
Saccharose = C12 H22 O11

Suite de nombres

Cet algorithme affiche la suite de 10 nombres (entiers) à partir du nombre défini par l'utilisateur.
Ce dernier entre au clavier une valeur et le programme affiche dans un ordre croissant les nombres qui le suivent, sans afficher la valeur de départ.

Suite de nombres pairs

Cet algorithme affiche une suite de nombres pairs (entiers) dans une plage déterminée par l'utilisateur.
Ce dernier entre au clavier une borne basse, ainsi qu'une borne haute, et le programme affiche dans un ordre croissant les nombres pairs compris dans ce domaine, sans afficher les bornes définies.

Somme d'une suite de nombres entiers

Cet algorithme affiche le résultat d'une somme de nombres (entiers).
L'utilisateur entre au clavier une valeur entière et le programme calcule la somme des entiers jusqu'à ce nombre y compris (en partant de zéro).

Somme d'une suite de nombres entiers et pairs

Cet algorithme affiche le résultat d'une somme de nombres (entiers) et pairs.
L'utilisateur entre au clavier une valeur entière (paire ou impaire) et le programme calcule la somme des entiers pairs jusqu'à ce nombre y compris (en partant de zéro).

Calcul d'une fraction

Cet algorithme affiche le résultat d'une fraction de deux nombres entiers.
L'utilisateur entre au clavier la valeur du numérateur ainsi que la valeur du dénominateur.
Le résultat s'affiche et le programme recommence la même opération. L'arrêt de celui-ci est provoqué uniquement lorsque l'utilisateur entre la valeur zéro pour le dénominateur (calcul impossible).

Structures linéaires de données I

Les tableaux

Les tableaux, spécificités et contraintes d'utilisation :

Un tableau est une structure linéaire dans laquelle un certain nombre de données vont pouvoir être insérées.

Pourquoi "linéaire" ? :

Parce cette structure intègre une série de données, représentées pour l'utilisateur de façon linéaire, et en général stockées dans la mémoire les unes dernières les autres.

Comme pour une structure de donnée élémentaire, le tableau porte un nom unique (identificateur) choisi par le programmeur.

Particularités des tableaux

Cette structure nécessite une déclaration afin de faire connaître au programme l'existence des données dans le but d'effectuer une réservation en mémoire.

Lors de la déclaration, un type de donnée élémentaire est affecté au tableau (typage statique). Les données insérées ou modifiées ne peuvent être que du type déclaré initialement dans chacune des cases mémoire réservées.

Tous les éléments d'un tableau sont du même type !

Le principe d'indexation

L'accès aux données d'un tableau se fait par le biais d'un indice (principe d'indexation) dans une dimension propre.

La principale caractéristique des tableaux est d'avoir la possibilité de représenter un domaine à une, deux, voire trois dimensions (ou plus). Mais tous les langages ne disposent pas de ce type de structure avec la même souplesse d'utilisation. En algorithmique : aucune limite; cependant, il est courant de s'adapter au langage de programmation dans lequel le programme est traduit.

Le tableau est une structure essentielle dont l'utilisation est déclinée suivant les diverses formes possibles de programmation dynamique (taille, type) et statique. Le principe d'indexation, noté avec des crochets [ ] est commun à tous les langages.

Tableau et organisation de la mémoire

Représentation en mémoire des tableaux

Exemple d'emploi d'un tableau de caractères :

Dans cet exemple, la taille de chaque case est d'un octet, format attribué à la mise en mémoire d'un caractère seul.

Rappel :

L'octet est la plus petite valeur adressable sur un ordinateur.

Par conséquent, l'adresse de chaque case mémoire pointée à l'aide du tableau indexé, est distante de l'élément suivant (ou précédent) d'un unique octet.

Aussi, tous les éléments d'un tableau construit en statique sont placés en mémoire de manière consécutive.

Tableaux et dimensions d'un tableau

Représentation graphique des tableaux

Opérations élémentaires

Catégories de données et opérations associées propres aux tableaux :

Les Constantes : CONST

Initialisation globale (avec la déclaration)
Utilisation unitaire

Les Variables : VAR

Déclaration
Initialisation globale (avec la déclaration)
Initialisation/affectation unitaire (hors de la déclaration)
Utilisation unitaire
Réaffectation unitaire

Déclaration

L'indice minimum correspond à l'indice du premier élément du tableau (la première case mémoire).

L'indice maximum correspond à l'indice du dernier élément du tableau (la dernière case mémoire). L'indice minimum et l'indice maximum sont toujours séparés par 2 points " . . "

Le type élémentaire (ENTIER, REEL, CARACTERE, CHAINE, BOOLEEN), est toujours précédé du signe :

Initialisation globale (avec la déclaration)

Les indices minimum et maximum correspondent aux indices du premier et du dernier (respectivement) élément du tableau.

Les valeurs affectées initialement sont notées entre accolades et séparées par une virgule. Le signe d'affectation est inscrit avant la première accolade.

Le type élémentaire (ENTIER, REEL, CARACTERE, CHAINE, BOOLEEN), est noté après les valeurs affectées.

Initialisation/affectation unitaire

L'indice courant est une valeur entière permettant d'adresser la case mémoire pour y insérer une valeur. L'indice courant peut être sous la forme d'un chiffre compris entre les indices min et max, une variable entière, une opération arithmétique comprenant une ou plusieurs variables et/ou des valeurs entières.

Valeur affectée dans le tableau à l'indice courant. L'affectation est indiquée grâce au symbole situé à la gauche de la valeur.

Utilisation unitaire

Nom du tableau dont est extraite la valeur affectée à l'autre donnée, grâce à l'adressage de la case mémoire obtenue à l'aide de l'indice courant.

L'indice courant est une valeur entière permettant d'adresser la case mémoire pour y lire la valeur affectée. L'indice courant peut être sous la forme d'un chiffre compris entre les indices min et max, une variable entière, une opération arithmétique comprenant une ou plusieurs variables et/ou des valeurs entières.

Réaffectation unitaire

L'indice courant du tableau dont la valeur est extraite peut être différent de celui de la réaffectation.

ATTENTION ! Ce formalisme n'est pas commun à toutes les publications d'algorithmique ou même les langages de programmation. Cependant, à quelques détails près, il n'y a pas de risque de mauvaise compréhension.

Exemple

Tableau à une dimension

Opérations élémentaires

TABLEAU A DEUX DIMENSIONS

Déclaration

Les indices minimum et maximum correspondent ici à la première dimension du tableau. L'indice minimum et l'indice maximum sont toujours séparés par 2 points " . . "

Les indices minimum et maximum correspondent ici à la seconde dimension du tableau.

Le type élémentaire (ENTIER, REEL, CARACTERE, CHAINE, BOOLEEN), est toujours précédé du signe :

Initialisation globale (avec la déclaration)

Les indices minimum et maximum correspondent ici à la première dimension du tableau pour les premiers crochets, à la seconde pour les suivants.

La première accolade signifie l'ouverture de la première dimension. Il y a autant d'ouverture d'accolades pour la seconde dimension que indice_max1d.

Les éléments sont ordonnés dans la seconde dimension, entre accolades.

Initialisation/affectation unitaire

L'indice courant 1d est ici l'indice de la première dimension.

L'indice courant 2d est ici l'indice de la seconde dimension.

Utilisation unitaire

L'indice courant 1d est ici l'indice de la première dimension.

L'indice courant 2d est ici l'indice de la seconde dimension.

Réaffectation unitaire

Nom du tableau dont la valeur adressée à l'aide des indices courants est réaffecté. Les indices courants de la valeur affectée peuvent être différents de ceux qui servent à la réaffectation.

Nom du tableau dont la valeur est utilisé pour la réaffectation.

Les indices courants du tableau dont la valeur est extraite peuvent être différents de ceux de la réaffectation.

TABLEAU A TROIS DIMENSIONS

Déclaration

Les indices minimum et maximum correspondent ici à la première, puis à la seconde dimension du tableau.

Les indices minimum et maximum correspondent ici à la troisième dimension du tableau.

Le type élémentaire (ENTIER, REEL, CARACTERE, CHAINE, BOOLEEN), est toujours précédé du signe :

Initialisation globale (avec la déclaration)

Les indices minimum et maximum correspondent ici à la première dimension du tableau pour les premiers crochets, à la seconde pour les suivants.

Les deux premières accolades signifient l'ouverture de la première, puis de la seconde

dimension.

Les éléments sont ordonnés dans la troisième dimension, entre accolades.

Initialisation/affectation unitaire

L'indice courant 1d est ici l'indice de la première dimension.

L'indice courant 2d est ici l'indice de la seconde dimension.

L'indice courant 3d est ici l'indice de la troisième dimension.

Utilisation unitaire

L'indice courant 1d est ici l'indice de la première dimension.

L'indice courant 2d est ici l'indice de la seconde dimension.

L'indice courant 3d est ici l'indice de la troisième dimension.

Réaffectation unitaire

Nom du tableau dont la valeur adressée à l'aide des indices courants est réaffecté. Les indices courants de la valeur affectée peuvent être différents de ceux qui servent à la réaffectation.

L'indice courant de la troisième dimension du tableau dont la valeur est extraite peuvent être différents de ceux de la réaffectation.

Exemple

Tableau à deux dimensions

Tableau à trois dimensions

Remarque

En langage de programmation, l'indice minimum est en général la valeur zéro. Celui-ci correspond au premier élément.

Vous pouvez utiliser 0 comme premier indice : Tableau[0..11] (Pour 12 éléments)

En langage de programmation, la déclaration nécessite un unique chiffre. Celui-ci correspond au nombre d'éléments désiré.

Pour initialiser un tableau, si les valeurs numériques utilisent la virgule, pour éviter les confusions, l'opérateur de séparation entre les éléments d'initialisation sera un point-virgule « ; ».

Présentation

Les structures abordées dans ce cours ne sont pas toutes issues d'une même catégorie.

Certaines structures, comme les piles, les files et les listes simulées à l'aide de tableaux, sont des structures statiques, dont la longueur (ou la taille), étant prédéfinie à l'avance, ne peuvent ni être redéfinie, ni évoluer durant l'exécution du programme.

Le défaut majeur d'une structure fixe ou statique, est l'obligation de "prévoir" la taille maximum de sa structure, car cela n'est pas toujours possible !

Aussi, lorsque la taille maximum prévue n'est que très rarement utilisée entièrement durant l'exécution du programme, une quantité de mémoire aura été réservée pour "rien", alors que celle-ci aurait été utile, pour d'autres programmes !

Structures statique et dynamique

Les structures élémentaires (entier, réel, caractère, chaîne, booléen) sont partagées en deux catégories :

En effet, les entiers, les réels, les caractères et les booléens font partis d'une catégorie : statique.

Les chaînes, en revanche, n'existent pas comme il a été présenté (pour des raisons de simplicité). Les chaînes sont, le plus souvent, implémentées de façon dynamique.

Les structures évoluées peuvent (ou doivent) être implémentées aussi de façon dynamique.

La structure linéaire de type : tableau, et la structure non linéaire de type : enregistrement, peuvent être montées de façon statique ou dynamique en fonction des contraintes de programmation.

Les structures linéaires de type : liste, pile, file, et les structures non linéaires : graphe et arbre sont quasi systématiquement implémentées de façon dynamique.

Si la programmation des structures de façon dynamique a pour objectif principal de limiter la taille des "pré-réservations" en mémoire, il faut aussi analyser ce choix afin de connaître

le réel intérêt d'une implémentation dynamique, (les temps d'accès, de recherche et de mise à jour des structures, le gain en capacité mémoire, les risques, …).

Notion de structure dynamique

Les différentes "formes" d'une structure de données dynamique :

Le terme "dynamique" cache divers aspects de la réservation de la mémoire.

L'emploi de pointeurs "typés", permet la réservation d'une variable de type pointeur (par exemple : un pointeur vers un entier) :

Le type de la donnée pointée est fixe,
Le référencement via l'adresse est dynamique.

L'emploi de pointeurs "typés", permet un accès à une structure complexe par son point d'entrée, ou le premier élément de sa structure :

Le type des données pointées est fixe,
Le référencement via l'adresse est dynamique,
La taille de la structure est extensible dynamiquement.

L'emploi de pointeurs non "typés", permet la réservation d'une variable de type pointeur, dont :

Le type de la donnée pointée n'est pas connu à la compilation, celui-ci est renseigné lors de l'exécution du programme, le typage est dynamique,
Le référencement via l'adresse est dynamique.
Attention aux types employés (conversions).

[Explications détaillées dans le chapitre suivant … à suivre]

Les pointeurs : Rappel – Précisions

Le pointeur est l'outil essentiel permettant la programmation dynamique (accès aux éléments placés en mémoire, de façon "indirecte", via des adresses).

La déclaration de variables élémentaires, sous la forme :

entier :ENTIER

est un mode de réservation mémoire statique. La capacité mémoire prise en compte est limitée au strict minimum, soit le nombre d'octets pour un entier (pour l'exemple, fixé à 4

octets*).

Dans le cas d'un pointeur sur un entier :

*ptr_entier :ENTIER

Seul un emplacement mémoire est réservé, celui-ci est du type variable pointeur :

A ce stade, la valeur de ce pointeur n'est pas valide, puisque la donnée à cet emplacement n'est pas initialisée.

La capacité de mémoire réservée pour ce pointeur est la taille d'une adresse (4 octets*).

Afin d'effectuer des tests sur la variable pointeur, celui-ci est forcé à NUL par convention (soit la valeur 0, il n'existe aucun élément enregistré).

*ptr_entier <-- NUL :ENTIER

La zone mémoire dédiée à l'affectation de l'entier n'est pas réservée !

L'affectation :

*ptr_entier <-- 7

Maintenant, le pointeur n'est plus NUL.
L'emplacement mémoire est effectivement réservé, pour un entier unique.
L'affectation est réalisée.
La capacité de mémoire réservée pour ce pointeur et sa variable entière est la taille d'une adresse à laquelle est ajoutée la taille d'une variable de type entier, (soit 8 octets*).

Après une ultime utilisation, (comme par exemple) :

ECRIRE(*ptr_entier)

Il est indispensable de libérer la mémoire de cette variable (seul le pointeur reste réservé), noté ici :

ptr_entier <-- NUL

Pourquoi "dynamique" ?

Parce que la zone mémoire où est affectée la variable, n'est pas déterminée lors de la réservation de son pointeur. Ainsi, entre deux référencements, si le pointeur est forcé à NUL, il y a de fortes chances pour que l'adresse mémoire de l'enregistrement effectif de la

valeur entière change !

Avantages

Les avantages sont nombreux, essentiellement face à un environnement lui-même dynamique.

On note :

Une adaptation de la capacité de la mémoire nécessaire au déroulement d'un programme, permanente et autonome,

La fin de la programmation où : "tout est défini et fixé entièrement par avance" dans les moindre détails,

Une adaptation à l'environnement liée à l'utilisation du logiciel : tous les ordinateurs ne sont pas identiques (capacités de mémoire, …).

Mais aussi :

Il existe aussi une gestion "native" de certaines structures dynamiques de données (comme les tableaux), implémentée au cœur des compilateurs, et dont le niveau de performances est élevé.

Inconvénients

Des inconvénients existent malgré tout.

On note :

Le manque de simplicité en comparaison avec la gestion des structures statiques,
Une nécessaire maîtrise des pointeurs,
Le suivi des allocations de la mémoire sur l'ensemble des structures dynamiques, durant tout le déroulement du processus, ainsi que la libération de la mémoire au terme de son utilisation (sinon risque de "fuites"),
La recopie d'une structure : plus exigeante que pour une structure statique (en code et en ressources),
La gestion des exceptions lors de la demande d'allocation.

Mais aussi :

Un manque d'uniformité de la gestion "native" implémentée dans les compilateurs entraîne des problèmes de transportabilité.

Dangers

Les structures dynamiques, très courantes par la praticité de leur emploi, sont aussi reconnues comme étant à l'origine de certains risques.

Les objets implémentent la plupart du temps, des données de façon dynamique (principalement de taille extensible). Il convient alors de configurer ou contrôler (via une borne) une telle structure afin d'éviter les problèmes de saturation de la mémoire ou les accès hors zone réservée.

On note essentiellement les dangers du type :

Arrêt prématuré du processus utilisant une structure dynamique (mauvaise implémentation, pas de protection),

Blocage du système d'exploitation sur lequel le processus est en cours de fonctionnement, …

Cela ne semble pas, a priori, représenter un réel "danger" ! Cependant, la sur-exploitation des structures dynamiques (parfois non contrôlées), fait apparaître la nécessité de prévenir ce risque par :

Une analyse systématique du besoin d'implémenter une telle structure.

La mise en place du contrôle permanent de la structure jusqu'à sa destruction.

Structures dynamiques : conseils

Il est aussi fortement conseillé :

De toujours initialiser un pointeur à NUL lors de sa déclaration,

De mettre à "zéro" la donnée pointée dès que celle-ci n'est plus utilisée,

Puis de réinitialiser à NUL le pointeur dès que celui-ci n'est plus utile, et/ou avant l'arrêt du programme ou sa destruction.

Le type CHAINE : tableau de caractères

Dans les langages de programmation, les chaînes sont des structures dynamiques* (en algorithmique, il est courant d'employer un type élémentaire statique nommé CHAINE pour faciliter les premiers pas vers la programmation).

Les chaînes sont constituées d'une suite de caractères. Le type de données de base est donc : le CARACTERE associé à un tableau.

Soit :

la structure de type tableau est plus grande que le mot lui-même :

la longueur du mot est stockée dans une variable distincte : dans ce cas, la suite de caractères peut être traitée en tant qu'une chaîne.

la longueur du mot n'est pas connue et la chaîne occupe tout ou partie de la structure : dans ce cas, le mot doit se terminer par un caractère spécial (caractère d'échappement nul) '\0'.

la structure est définie en fonction de la taille du mot lui-même :

la longueur du mot n'est pas connue et la chaîne occupe toute la structure : dans ce cas, le mot doit se terminer par un caractère spécial (nul) '\0'.

Les deux derniers cas sont les plus courants. Le caractère spécial '\0' représente un caractère unique : longueur_chaîne = longueur_mot + 1

Les structures de type tableau déclaré en statique, permettent aussi de créer des suites de caractères.

Dans l'exemple précédent, "un_mot[indice]" correspond à l'accès de chaque caractère pris en compte individuellement.

L'affectation d'une chaîne constante n'est possible qu'à l'aide d'une opération de changement de type :

Car, ce que la déclaration réalise, en fait, c'est l'attribution d'une adresse à "un_mot" et une réservation pour un nombre d'éléments fini.

Le nom attribué au tableau "un_mot" est en réalité une variable pointeur sur le premier élément du tableau, dont l'adresse peut être obtenue d'une autre manière, à l'aide de : "&un_mot[1]".

Le compilateur traite par conséquent, de la même façon, les déclarations écrites sous les formes :

Attention à l'affectation d'une chaîne constante lorsque la structure est de taille déterminée (changement de type nécessaire).

Les structures de type tableau déclaré en dynamique, via un pointeur, permettent de créer des véritables chaînes.

Une des primitives employées dans les premiers exercices du cours permet de connaître la longueur d'un mot.

Une primitive pour l'extraction des caractères.

Tableau dynamique à une dimension

Les structures de type tableau déclaré en dynamique, via un pointeur, sont des collections indexées d'objets identiques.

La gestion de la taille de façon dynamique est adaptée de l'écriture en langage C++.

Tableau dynamique à deux dimensions

Identifier au moins trois exercices du chapitre III dont l'utilisation de tableaux aurait été possible.

Modifier les algorithmes produits pour chacun de ces trois exercices afin d'y intégrer l'emploi de cette structure linéaire de données.

Structures algorithmiques imbriquées

Construction par imbrication

Résoudre un problème grâce à la conception d'un programme informatique doit être envisagé à l'aide des quelques éléments abordés jusqu'à maintenant (structures de données et structures algorithmiques).

En théorie, tout peut être traité à l'aide de ces deux structures.

La combinaison des éléments de ces structures est infinie. Et c'est pour cette raison qu'une solution algorithmique peut, sauf cas exceptionnel, tout résoudre. C'est aussi la raison pour laquelle il peut exister plusieurs solutions algorithmiques pour un même problème.

La condition : une expression booléenne

Les instructions de rupture de séquence nécessitent l'emploi d'une condition.

Cette condition peut être simple ou multiple (à suivre).

La condition, ne pouvant prendre que deux valeurs : VRAI ou FAUX (comme le type booléen), un programme sera valide si la construction algorithmique est valide pour un ensemble de conditions valides !

Dit autrement, un grand nombre d'erreurs de programmation se trouvent au niveau des conditions des instructions de sélection ou de répétition.

Ruptures de séquences et risques

A ce stade, il faut distinguer le risque d'erreurs de programmation au niveau des conditions des instructions de sélection ou de répétition.

Pour une instruction de sélection, le risque est minime, car le programme ne répond pas à l'attente que l'on en a.

Pour une instruction de répétition, le risque est majeur, car le programme peut ne pas répondre à l'attente que l'on en a (cas où la répétition ne s'effectue pas), mais surtout, la répétition risque de ne pas s'arrêter.

Conclusion : Dans le cas d'emploi des instructions de répétition, le programmeur devra redoubler de vigilance afin de contrôler que chacune d'elles se termine !

Imbrications par l'exemple

Imbrication : SI..ALORS

Énoncé : Groupe sanguin et filiation*

*ATTENTION, ces critères ne sont valables que pour 95% de la population, voire moins !

La filiation, la relation via les groupes sanguins (critères d'impossibilité) :

Deux parents O ne peuvent avoir d'enfant d'un autre type que O,

Deux parents A, ou un parent A et un parent O, ne peuvent avoir d'enfant B, ni AB,

Deux parents B, ou un parent B et un parent O, ne peuvent avoir d'enfant A, ni AB,

Si l'un des parents est AB, l'enfant ne peut être O.

```
ALGORITHME: filiation
VAR:    mere, pere  :CHAINE
DEBUT
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
de la mère [en majuscule]")
    mere <-- LIRE()
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
du père [en majuscule]")
    pere <-- LIRE()
    SI (mere = "O")
    ALORS
        SI(pere = "O")
        ALORS
            ECRIRE("L'enfant sera de type O")
        FINSI
    FINSI
    SI (mere = "A")
    ALORS
        SI(pere = "A")
        ALORS
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        FINSI
    FINSI
    SI (mere = "A")
    ALORS
        SI(pere = "O")
        ALORS
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        FINSI
    FINSI
    SI (mere = "O")
    ALORS
        SI(pere = "A")
        ALORS
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        FINSI
    FINSI
    SI (mere = "B")
    ALORS
        SI(pere = "B")
        ALORS
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        FINSI
    FINSI
```

```
    SI (mere = "B")
    ALORS
        SI(pere = "O")
        ALORS
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        FINSI
    FINSI
    SI (mere = "O")
    ALORS
        SI(pere = "B")
        ALORS
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        FINSI
    FINSI
    SI(mere = "AB")
    ALORS
        ECRIRE("L'enfant ne peut être O")
        ECRIRE("L'enfant peut être du groupe A ou B ou AB")
    FINSI
    SI(pere = "AB")
    ALORS
        ECRIRE("L'enfant ne peut être O")
        ECRIRE("L'enfant peut être du groupe A ou B ou AB")
    FINSI
FIN
```

Imbrication : SI..ALORS..SINON

Énoncé : Indice de Masse Corporelle*

*ATTENTION, ces critères ne sont valables que pour estimation générale et nécessitent une interprétation élaborée par des spécialistes !

IMC, présentation :

L'indice de masse corporelle permet d'estimer la quantité de masse grasse de l'organisme à partir de deux paramètres : le poids et la taille. Il permet d'évaluer la relation entre le poids et la santé en déterminant la corpulence de la personne, de voir s'il y a obésité ou maigreur et il en détermine la sévérité. L'IMC ne doit pas être utilisé comme seule mesure de la composition corporelle et/ou de la condition physique.

Bien que l'IMC fournisse des points de repère indicatifs des niveaux de poids chez les adultes (de 18 à 65 ans), il ne s'applique pas aux nourrissons, aux enfants, aux femmes enceintes ou celles qui allaitent, aux personnes gravement malades, aux athlètes, ni aux

adultes de plus de 65 ans.

IMC, calcul :

L'indice de masse corporelle est le rapport du poids (exprimé en Kilogrammes) sur le carré de la taille (exprimée en mètres). Il se calcule donc avec une formule mathématique simple :

IMC = poids / (taille x taille)

IMC, interprétation possible des résultats :

inférieur à 25 : NORMAL
de 25 à 29 : SURPOIDS
de 30 à 34 : OBESITE MODEREE
de 35 à 39 : OBESITE SEVERE
supérieur à 39 : OBESITE MASSIVE

[Enoncé élaboré et adapté à partir des sources suivantes : Magazine de la santé, France5 - 2006]

```
ALGORITHME: imc
VAR:
    imc :REEL
    poids :ENTIER
    taille :REEL
DEBUT
    ECRIRE("Entrez votre poids en kilogramme, puis votre taille en mètre :")
    poids <-- LIRE()
    taille <-- LIRE()
    imc <-- poids / (taille * taille)
    SI(imc <= 25)
    ALORS
          ECRIRE("Poids normal")
    SINON
        SI(imc <= 30)
        ALORS
            ECRIRE("Surpoids")
        SINON
            SI(imc <= 35)
            ALORS
                ECRIRE("Obésité modérée")
            SINON
                SI(imc <= 40)
                ALORS
                    ECRIRE("Obésité sévère")
                SINON
```

```
                    SI(imc > 40)
                    ALORS
                      ECRIRE("Obésité massive")
                FINSI
            FINSI
        FINSI
    FINSI
FIN
```

Imbrication : CAS..PARMI

Énoncé : Groupe sanguin et filiation

Afin de comparer le déroulement des instructions conditionnelles SI..ALORS et CAS..PARMI, l'exemple traité dans ce chapitre est identique au premier exemple vu dans ce module.

ATTENTION ! L'instruction équivalente dans les langages compilables possède des spécificités qu'il faudra prendre en compte en phase d'analyse pour la construction d'algorithmes (expression en valeur entière, chaque cas testé est une constante, emploi d'un « break »). Pour ces raisons, en algorithmique, l'instruction de sélection la plus employée sera le « SI ».

```
ALGORITHME: filiation2
VAR :
    mere, pere  :CHAINE
DEBUT
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
de la mère [en majuscule]")
    mere <-- LIRE()
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
du père [en majuscule]")
    pere <-- LIRE()
    CAS( mere )PARMI:
    CAS1: "O"
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant sera de type O")
        CAS2: "A"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS3: "B"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        FINCASPARMI
```

```
    CAS2: "A"
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS2: "A"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS3: "B"
            ECRIRE("Cas indéterminé")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        FINCASPARMI
CAS3: "B"
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS2: "A"
            ECRIRE("Cas indéterminé")
        CAS3: "B"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        FINCASPARMI
    CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
    FINCASPARMI
FIN
```

Imbrication : CAS..PARMI..PARDEFAUT

Énoncé : Groupe sanguin et filiation

La structure du CAS..PARMI..PARDEFAUT possède l'avantage de donner une alternative non formalisée en fin de structure conditionnelle.

Les langages de programmation possèdent des spécificités de fonctionnement pour cette structure (break, expression de comparaison constante). Nous éviterons, par conséquent de l'employer !

Lors de la traduction du langage algorithmique au langage évolué, il sera nécessaire, parfois, d'adapter l'écriture algorithmique aux particularités du langage de développement

utilisé.

```
ALGORITHME: filiation2
VAR :
    mere, pere  :CHAINE
DEBUT
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
de la mère [en majuscule]")
    mere <-- LIRE()
    ECRIRE("Veuillez entrer la(les) lettre(s) relative(s) au groupe sanguin
du père [en majuscule]")
    pere <-- LIRE()
    CAS( mere )PARMI:
    CAS1: "O"
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant sera de type O")
        CAS2: "A"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS3: "B"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        PARDEFAUT:
            ECRIRE("Veuillez vérifier les informations entrées pour le groupe
sanguin du père !")
        FINCASPARMI
    CAS2: "A"
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS2: "A"
            ECRIRE("L'enfant ne sera ni B, ni AB")
            ECRIRE("L'enfant peut être du groupe A ou O")
        CAS3: "B"
            ECRIRE("Cas indéterminé")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        PARDEFAUT:
            ECRIRE("Veuillez vérifier les informations entrées pour le groupe
sanguin du père !")
        FINCASPARMI
    CAS3: "B"
```

```
        CAS( pere )PARMI:
        CAS1: "O"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS2: "A"
            ECRIRE("Cas indéterminé")
        CAS3: "B"
            ECRIRE("L'enfant ne sera ni A, ni AB")
            ECRIRE("L'enfant peut être du groupe B ou O")
        CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
        PARDEFAUT:
            ECRIRE("Veuillez vérifier les informations entrées pour le groupe
sanguin du père !")
        FINCASPARMI
    CAS4: "AB"
            ECRIRE("L'enfant ne peut être O")
            ECRIRE("L'enfant peut être du groupe A ou B ou AB")
    PARDEFAUT:
            ECRIRE("Veuillez vérifier les informations entrées pour le groupe
sanguin de la mère !")
    FINCASPARMI
FIN
```

Imbrication d'instructions diverses

Énoncé : Tendance à l'obésité

Calcul de l'indice de tendance à l'obésité :

Prise de mesure au tour de taille.

Pour un homme, tendance à l'obésité lorsque la mesure dépasse 100 cm.

Pour une femme, tendance à l'obésité lorsque la mesure dépasse 90 cm.

```
ALGORITHME: tendance_obesite
CONST:
homme <-- 100 :ENTIER
femme <-- 90 :ENTIER
sexe_masculin <-- 'M' :CARACTERE
sexe_feminin <-- 'F' :CARACTERE
VAR:
mesure :ENTIER
sexe :CARACTERE
```

```
DEBUT
    ECRIRE("Veuillez renseigner le programme sur votre sexe: M pour masculin,
et F pour féminin.")
    sexe <-- LIRE()
    ECRIRE("Veuillez entrer la valeur entière relative à votre tour de
taille, SVP.")
    mesure <-- LIRE()
    CAS(sexe)PARMI
    CAS1: sexe_masculin
        SI(mesure > homme)
        ALORS
            ECRIRE("Veuillez consulter votre médecin pour une possible
tendance à l'obésité")
        SINON
    ECRIRE("Vous n'avez pas, a priori, de tendance à l'obésité")
        FINSI
    CAS2: sexe_feminin
        SI(mesure > femme)
        ALORS
    ECRIRE("Veuillez consulter votre médecin pour une possible tendance à
l'obésité")
        SINON
    ECRIRE("Vous n'avez pas, a priori, de tendance à l'obésité")
        FINSI
    FINCASPARMI
FIN
```

Rappel

Problèmes liés aux instructions répétitives (ou boucles)

Il est parfois nécessaire, en fonction d'une condition particulière, de traiter un certain nombre de fois une opération ou suite d'opérations.

Lorsque l'on détecte ce besoin, une structure répétitive doit être employée afin d'éviter une écriture séquentielle de structures conditionnelles bien trop lourde à mettre en place et incompatible avec des informations saisies par l'utilisateur, c'est à dire inconnues lors de la compilation du programme.

Les langages permettent de programmer des structures répétitives “sans réelles limites” liées au langage. Certains exemples de ce module montrent les possibilités offertes. Toutefois, la validation de ces algorithmes est délicate, et la recherche de structuration simple est fortement conseillée !

Puisqu'il est impératif de définir une condition pour "rentrer" dans certaines boucles (dont la condition est en entrée), ou pour "effectuer la répétition", il peut arriver que celle-ci ne soit pas "VRAI" lors du traitement de l'instruction répétitive ! Dans ce cas, l'instruction suivante

est prise en compte et ainsi de suite ..., en revanche, dans le cas contraire, où la répétition s'opère normalement, il faut impérativement "sortir" de cette boucle ! (Sinon, le programme ne se termine jamais).

Conclusion : les opérations répétitives nécessiteront un examen minutieux des conditions de sortie de la boucle.

Imbrication : TANTQUE

Énoncé : Agenda journalier évolué

L'objectif de ce programme est de faire évoluer le programme d'affichage de l'agenda vu précédemment.

L'utilisateur a maintenant le choix du nombre de jours à afficher.

```
ALGORITHME: agenda_journalier4
CONST:
VAR:
    nombre_jours :ENTIER
    jour <-- 1 :ENTIER
    heure <-- 8 :ENTIER
DEBUT
    ECRIRE("Veuillez entrer le nombre de jours d'édition de l'agenda.")
    nombre_jours <-- LIRE()
    TANTQUE( jour <= nombre_jours )
    FAIRE
        ECRIRE( "Jour numéro: ", jour )
        TANTQUE( heure <= 18 )
        FAIRE
            ECRIRE( heure, "H00 : ")
            heure  <-- heure + 1
        FINTANTQUE
        jour  <--  jour + 1
        heure <-- 8
    FINTANTQUE
FIN
```

Imbrication : POUR

A l'identique des autres instructions répétitives, l'instruction POUR risque de boucler indéfiniment !

L'instruction POUR est une instruction répétitive "tout en un". C'est à dire qu'elle a la possibilité d'intégrer une déclaration avec initialisation, la définition de la borne de sortie,

ainsi que la valeur du pas (et de son signe).

Il est, par conséquent, conseillé de ne pas modifier les éléments contenus dans cette instruction au coeur de la boucle ! Lors de l'optimisation d'algorithmes, il est courant de voir ce conseil levé, dans les programmes ainsi que dans la littérature du domaine.

Énoncé : Tables de multiplication complètes

L'objectif de ce programme est d'afficher les tables de multiplication de 0 à 10.

```
ALGORITHME: table_multiplication
VAR:
DEBUT
    POUR( chiffre1 <-- 0 :ENTIER a 10 au pas de +1 )
    FAIRE
        ECRIRE( "Table de multiplication de: ", chiffre1 )
        POUR( chiffre2 <-- 0 :ENTIER a 10 au pas de +1)
        FAIRE
            ECRIRE( chiffre1, " x ", chiffre2, " = ", chiffre1 * chiffre2 )
        FINPOUR
    FINPOUR
FIN
```

Imbrication : REPETER..JUSQU'A

Énoncé : Agenda journalier

Afin de comparer le déroulement des instructions répétitives TANTQUE et REPETER, l'exemple traité dans ce chapitre est identique au premier exemple vu dans ce module.

```
ALGORITHME: agenda_journalier5
VAR:
    nombre_jours :ENTIER
    jour <-- 1 :ENTIER
    heure <-- 8 :ENTIER
DEBUT
    ECRIRE("Veuillez entrer le nombre de jours d'édition de l'agenda.")
    nombre_jours <-- LIRE()
    REPETER
        ECRIRE( "Jour numéro: ", jour )
        REPETER
            ECRIRE( heure, "H00 : ")
            heure  <-- heure + 1
        JUSQU'A( heure > 18 )
        jour  <--  jour + 1
        heure <-- 8
```

```
    JUSQU'A( jour > nombre_jours )
FIN
```

Imbrication : REPETER..TANTQUE

Énoncé : Agenda journalier

Il existe deux variantes pour l'écriture de l'instruction répétitive REPETER. Elles sont présentées dans le but de lire sans problème ces mêmes structures dans la littérature du domaine.

Ses deux variantes sont complémentaires.

Certains langages n'implémentent qu'une seule de ces instructions.

Lors de l’utilisation d’un langage de programmation, la traduction depuis le langage algorithmique peut nécessiter l’emploi d’une écriture algorithmique adaptée aux particularités du langage (typage des données, …).

```
ALGORITHME: agenda_journalier6
VAR:
    nombre_jours :ENTIER
    jour <-- 1 :ENTIER
    heure <-- 8 :ENTIER
DEBUT
    ECRIRE("Veuillez entrer le nombre de jours d'édition de l'agenda.")
    nombre_jours <-- LIRE()
    REPETER
        ECRIRE( "Jour numéro: ", jour )
        REPETER
            ECRIRE( heure, "H00 : ")
            heure  <-- heure + 1
        TANTQUE( heure <= 18 )
        jour  <--  jour + 1
        heure <-- 8
    TANTQUE( jour <= nombre_jours )
FIN
```

Année bissextile

Le problème suivant correspond à la détermination du nombre de jour d’un mois quelconque, et quelque soit l’année (bissextile ou non).

Une année est bissextile si le millésime correspondant vérifie l'une des 2 propriétés suivantes :

soit le millésime est divisible par 4 et n'est pas divisible par 100 ;

soit le millésime est divisible par 400.
(Exemple : 1900 n'était pas une année bissextile, 1996 et 2000 étaient bissextiles.)

Compteur de points des lettres sur un pupitre du jeu de Scrabble

Au Scrabble, lorsque l'on recherche un mot, avant de le poser sur la grille du jeu, on tente d'obtenir un mot dont le nombre de points est le plus important.

Dans cette optique, et sans prendre en compte l'endroit où sera posé le mot sur le grille (incidence sur la valeur du mot : mot compte double ou triple, lettre compte double ou triple), ni le fait qu'une lettre déjà posée servira à constituer le mot, on va calculer la valeur du mot, ce qui permettra de comparer les diverses propositions de mots entre elles.

La valeur des lettres est la suivante :
a, e, i, l, n, o, r, s, t, u = 1 point
d, g, m = 2 points
b, c, p = 3 points
f, h, v = 4 points
j, q = 8 points
k, w, x, y, z = 10 points

Détecteur de palindrome

Un palindrome est un mot qui peut se lire dans les deux sens, de gauche à droite comme de droite à gauche. Par exemple, « ici » est un palindrome à trois lettres.

Le programme à réaliser devra donc, après avoir saisi un mot, d'indiquer si celui-ci est un palindrome ou non, ainsi que le nombre de lettres.

Détecteur d'anagramme

Ce programme recherche la présence d'anagramme entre les deux mots saisis par l'utilisateur.
L'anagramme est un ensemble de lettres qui permettent d'écrire au moins deux mots différents.

Par exemple : « AIMER » et « MARIE » sont des anagrammes.

Durée d'une plongée avec bouteille

Pour une profondeur inférieure ou égale à 100 m, sachant que le plongeur doit faire un palier de 100 secondes à 30 m puis de 30 m à 0, des paliers tous les 3 m, le temps diminuant de 10 secondes à chaque fois.

Les paliers au-dessus de 30 m étant par contre augmentés de 25 secondes tous les 10 m, (ainsi p.ex.: 150 s à 50 m).

La vitesse de remontée est fixée à 1 m/s.

Pour une profondeur donnée, on cherche à calculer le temps global d'attente aux paliers, afin de connaître les capacités des bouteilles, et de déterminer aussi le temps de travail à une profondeur donnée.

Structures non linéaires de données I

Une structure composée

Structure abstraite non linéaire de données :

Structure de données complexe composée de données de types identiques ou différents, voire constituée d'autres structures complexes.

Comme pour une instruction de définition d'un nom de type personnalisé (en langage de programmation : « typedef »), ici, le nouveau type déclaré est complexe car il représente une donnée composée (« struct »).

Ces données sont appelées des "champs", (ou "données membres" en programmation objet). Le même terme est employé dans les tables de base de données.

Une structure hiérarchisée

Structure de données : ENREGISTREMENT

L'accès aux champs ne peut se faire que par le biais de la variable déclarée pour cette structure, (principe d'imbrication ou de hiérarchisation ce qui interdit l'accès direct aux variables membres).

Une structure quelconque imbriquée dans une autre doit être déclarée avant celle qui

l'utilise afin d'être reconnue.

Ce modèle de structure de données est une représentation proche de celle employée pour la modélisation des tables de bases de données.

[En programmation objet, il existe trois grands types de structures complexes de même format : les classes, les structures et les unions. Elles fonctionnent chacune avec leurs spécificités, mais elles utilisent le même principe d'imbrication, autorisant, en plus, la déclaration de fonctions membres qui n'est pas abordé dans ce cours.]

Structure : ENREGISTREMENT

Déclaration des structures de type Enregistrement

```
//structure d'enregistrement simple
ENREGISTREMENT: date
DEBUTENREGISTREMENT
        ... //détail du contenu de l'enregistrement
FINENREGISTREMENT

//autre écriture:
STRUCTURE: date
DEBUT
        ... //détail du contenu de l'enregistrement
FIN
```

Soit, par exemple une structure constituée de deux variables de même type.

```
//Exemple de structure:
ENREGISTREMENT: point //Coordonnées d'un point
DEBUTENREGISTREMENT
        x :ENTIER //champ 1: abscisse
        y :ENTIER //champ 2: ordonnée
FINENREGISTREMENT
```

L'accès aux champs est réalisé à l'aide de l'opérateur "." employé derrière la variable déclarée à l'aide de cette structure.

Utilisation des structures de type Enregistrement

```
//Exemple de structure:
ENREGISTREMENT: point //Coordonnées d'un point
DEBUTENREGISTREMENT
        x :ENTIER //champ 1: abscisse
        y :ENTIER //champ 2: ordonnée
```

```
FINENREGISTREMENT
//------------------------------------------------------------
//déclaration d'une variable à l'aide de cette structure
VAR:
        un_point :point
DEBUT
        un_point.x <-- LIRE() //affectation directe du champ
        un_point.y <-- LIRE() //affectation directe du champ
FIN
```

Une structure de type abstrait : Enregistrement

A l'aide de la structure "point", il n'est pas possible d'accéder à x (ou y).

```
//déclaration d'une variable à l'aide de cette structure
VAR:
        un_point :point
DEBUT
        x <-- LIRE() //affectation directe impossible
        point.y <-- LIRE() //variable "point" non déclarée
FIN
```

L'emploi du nom de la structure en dehors d'une déclaration n'a aucun sens.

Imbrication des structures de type Enregistrement

```
//structures d'enregistrements imbriqués
ENREGISTREMENT: date
DEBUTENREGISTREMENT
        ... //détail du contenu de l'enregistrement
FINENREGISTREMENT
    ... //autres variables
//------------------------------------------------------------
ENREGISTREMENT: personne
DEBUTENREGISTREMENT
        ...
        date_naissance :date //déclaration à l'aide
        //d'une autre structure d'enregistrement:date
        ...
FINENREGISTREMENT
```

Enregistrement simple

Déclaration

```
VAR:
ENREGISTREMENT: nom_enregistrement
```

```
DEBUTENREGISTREMENT
               nom_champ1 :<TYPE>
        ...
               nom_champn :<TYPE>
FINENREGISTREMENT
    enr1, enr2, enr3, enr4, enr5 :nom_enregistrement
```

Représentation de la structure

Représentation des champs sous la forme d’une table

Opérations élémentaires

Affectation unitaire

```
VAR:
ENREGISTREMENT: personne
DEBUTENREGISTREMENT
               nom     :CHAINE
        prenom     :CHAINE
        nni    :ENTIER
FINENREGISTREMENT
    pers1, pers2, pers3, pers4, pers5 :personne
DEBUT
    ...
    pers1.nom <-- "PARSER"
    pers1.prenom <-- "Kevin"
    pers1.nni <-- 1700833023126
    ...
FIN
```

Utilisation unitaire

```
VAR:
ENREGISTREMENT: personne
DEBUTENREGISTREMENT
               nom     :CHAINE
        prenom     :CHAINE
        nni    :ENTIER
FINENREGISTREMENT
    pers1, pers2, pers3, pers4, pers5 :personne
    variable_chaine1, variable_chaine2 :CHAINE
    variable_entiere :ENTIER
DEBUT
    ECRIRE (pers1.nom, pers1.prenom, pers1.nni)
```

```
    variable_chaine1 <-- pers1.nom
    variable_chaine2 <-- pers1.prenom
    variable_entiere <-- pers1.nni
FIN
```

Ré-affectation unitaire

```
VAR:
ENREGISTREMENT: famille
DEBUTENREGISTREMENT
               nom_pere     :CHAINE
        nom_mere     :CHAINE
        nb_enfants     :ENTIER
FINENREGISTREMENT
    couple1, couple2, couple3     :famille
DEBUT
    ...
    couple1.nb_enfants <-- couple1.nb_enfants + 1
    ...
FIN
```

Emploi d'un tableau d'enregistrements

Structure linéaire indexée d'enregistrements :

```
VAR:
ENREGISTREMENT: personne
DEBUTENREGISTREMENT
               nom     :CHAINE
        prenom     :CHAINE
        nni    :ENTIER
FINENREGISTREMENT
TABLEAU:    pers[1..5] :personne
DEBUT
    ...
    pers[1].nom <-- "PARSER"
    pers[1].prenom <-- "Kevin"
    pers[1].nni <-- 1700833023126
    ...
FIN
```

Emploi d'un tableau dans un enregistrement

Structure linéaire indexée dans un enregistrement :

```
VAR:
ENREGISTREMENT: eleve
DEBUTENREGISTREMENT
```

```
               nom      :CHAINE
        prenom     :CHAINE
        notes[1..12]    :REEL
FINENREGISTREMENT
TABLEAU:    pers :eleve
DEBUT
    ...
    pers.notes[1] <-- 17
    pers.notes[2] <-- 13
    pers.notes[3] <-- 11,5
    ...
FIN
```

Enregistrement imbriqué

Déclaration

```
VAR:
ENREGISTREMENT: date
DEBUTENREGISTREMENT
               jour      :ENTIER
        mois      :ENTIER
               annee      :ENTIER
FINENREGISTREMENT
ENREGISTREMENT: personne
DEBUTENREGISTREMENT
               nom      :CHAINE
        prenom     :CHAINE
        date_naissance    :date
FINENREGISTREMENT
    pers1, pers2, pers3, pers4, pers5 :personne
```

Représentation de la structure

Opérations élémentaires

Affectation unitaire

```
VAR:
    // avec les structures "date" et "personne"
    // des diapositives précédentes
    pers1, pers2, pers3, pers4, pers5 :personne
DEBUT
    ...
    pers1.nom <-- "PARSER"
```

```
    pers1.prenom <-- "Kevin"
    pers1.date_naissance.jour <-- 23
    pers1.date_naissance.mois <-- 8
    pers1.date_naissance.annee <-- 1970
    ...
FIN
```

Utilisation unitaire

```
VAR:
    // avec les structures "date" et "personne"
    // des diapositives précédentes
    pers1, pers2, pers3, pers4, pers5 :personne
    date_du_jour :date
DEBUT
    ...
    ECRIRE (pers1.nom, pers1.prenom)
    ECRIRE (pers1.date_naissance.jour,                         '/',
pers1.date_naissance.mois,                  '/', pers1.date_naissance.annee)
    date_du_jour.annee <-- 2007
    ECRIRE ("Age: ", date_du_jour.annee -
 pers1.date_naissance.annee)
    ...
FIN
```

Imbrication des données avec des enregistrements

Calcul de la moyenne d'un élève

Le programme doit permettre de calculer la moyenne de trois matières pour un unique élève, en prenant en compte un seul contrôle QCM comptant pour 40 %, et deux travaux pratiques de 30 % chacun.

Les matières sont les suivantes (elles ont le même poids sur la note finale) :
ALGO
C
JAVA
L'élève doit renseigner tout d'abord, son nom, son prénom, son numéro de cin, sa date de naissance et les dates de chaque examen.

Le programme doit ensuite, en fonction de la moyenne obtenue par l'élève, attribuer une mention en fonction de la liste suivante :

- moyenne supérieure ou égale à 16 et inférieure ou égale à 20 : mention très bien
- moyenne supérieure ou égale à 14 et inférieure à 16 : mention bien
- moyenne supérieure ou égale à 12 et inférieure à 14 : mention assez bien
- moyenne supérieure ou égale à 10 et inférieure à 12 : mention passable

- moyenne supérieure ou égale à 5 et inférieure à 10 : mention insuffisant
- moyenne supérieure ou égale à 0 et inférieure à 5 : mention très insuffisant

Structures linéaires de données II

Une structure linéaire chaînée

Les structures linéaires de type tableau indexé :

Parmi les structures dites linéaires, le tableau (à une ou deux dimensions) est une structure de données très simple à programmer et aussi très facile à utiliser, (donc très employée).

Cependant, certains aspects d’une telle structure rendent inefficaces les algorithmes qui les utilisent. Par exemple, la durée d’exécution d’un algorithme de tri d’un tableau de grande taille peut s’avérer extrêmement long.

On préfèrera monter des structures plus « souples » d’emploi dans ce cadre : des listes (chaînées), dont les éléments s’enchaînent comme des maillons.

Un maillon

Qu’est-ce qu’un maillon ?

Un maillon est une structure d’enregistrement.

Chaque maillon est constitué (par exemple) :

Une variable pour contenir une information unique :

valeur :CARACTERE

Une variable pointeur pour accéder au prochain maillon de la chaîne (suite de maillon):

*ptr_suivant <-- NUL :maillon

Un maillon peut contenir un second pointeur pour accéder au maillon précédent (principe

d’une chaîne double). Si nécessaire, la variable « valeur » peut être accompagnée d’autres variables, de type identique ou différent.

Chaînage simple

Chaînage simple et circulaire

Chaînage double

Chaînage double et circulaire

Gestion du pointeur

Positionnement du pointeur courant

La gestion de la position courante du pointeur est très importante.

Suivant l’utilisation qui est faite de la structure chaînée et des contraintes associées (pile, file, liste, …), l’utilisateur se trouve face à deux problèmes de programmation :

Lors de l’utilisation en vue d’une insertion :

le choix de l’emplacement de l’élément à ajouter, par rapport à la position courante.

Lors de la modification de la structure :

le choix de la technique d’insertion ou de suppression.

Le choix de ces techniques permet de garantir :

un fonctionnement sûr des structures dynamiques,

et d’identifier les traitements les moins coûteux.

Chaînage simple : Création de la chaîne

Première étape :

Déclaration d’un pointeur d'entrée

*ptr_d_entree <-- NUL :maillon

La déclaration s’effectue en zone « VARIABLE » et ne permet de réserver qu'un emplacement mémoire de type variable pointeur.

Le pointeur est NUL.

Seconde étape :

Déclaration d’une structure d’enregistrement nommée « maillon », constituée de :

valeur :CARACTERE
*ptr_suivant <-- NUL :maillon

*ptr_temp :maillon
ptr_temp <-- NOUVEAU :maillon

Troisième étape :

Liaison puis mise à jour de la valeur de « maillon » :

ptr_d_entree <-- ptr_temp
(*ptr_d_entree).valeur <-- ‘S’
(*ptr_d_entree).ptr_suivant <-- NUL

Stratégie de gestion : Généralités

Problème d'ordre des traitements, ajout et insertion

Dans l'exemple précédent, la mise à jour des valeurs du maillon est effectuée via le pointeur d'entrée, et non directement, lors de la création du maillon (voir exemple suivant).

Lors de l’utilisation d’une chaîne simple ou double en vue d’une insertion, la gestion du pointeur courant doit permettre de traiter l’insertion :

soit après l’élément courant,
soit avant.

Ce choix est primordial afin de traiter l’ensemble des insertions/retraits à l’aide d'une stratégie cohérente.

Dans le cas d’une suppression, c’est l’élément courant qui est supprimé. Le problème de positionnement avant l'action ne se pose pas, contrairement à l'insertion. En revanche, la

nouvelle position courante doit être déterminée à l'avance : soit le maillon suivant devient la position courante, soit c'est le précédent, (dans le cas où ceux-ci existent).

Chaînage simple : Option d'ajout (ajout à droite)

Première étape :

Déclaration d'un nouveau maillon :

*ptr_temp :maillon
ptr_temp <-- NOUVEAU :maillon

Seconde étape :

Mise à jour de la valeur du nouveau maillon avant la liaison :

(*ptr_temp).valeur <-- 'I'
(*ptr_temp).ptr_suivant <-- NUL

Troisième étape :

Liaison du nouveau maillon avec le maillon d'entrée :

(*ptr_d_entree).ptr_suivant <-- ptr_temp

Chaînage simple : Option d'insertion (ajout à gauche)

Première étape :

Déclaration d'un nouveau maillon :

*ptr_temp :maillon
ptr_temp <-- NOUVEAU :maillon

Seconde étape :

Mise à jour de la valeur du nouveau maillon :

(*ptr_temp).valeur <-- 'I'

Troisième étape :

Mise à jour du pointeur sur le maillon suivant :

(*ptr_temp).ptr_suivant <--ptr_d_entree

Quatrième étape :

Mise à jour du pointeur d'entrée sur le premier maillon :

ptr_d_entree <-- ptr_temp
ptr_temp <-- NUL

Stratégie de gestion : Conclusion

Modification de la structure

Quelle que soit la stratégie employée, le pointeur d’entrée du chaînage ne doit pas être perdu (effacé ou écrasé).

Lorsque cela est possible (par rapport aux objectifs définis pour utiliser cette structure):

Soit : les modifications du pointeur d’entrée restent « interdites » (excepté lors de la libération de la mémoire).

Soit : avant toute modification d'un pointeur d'accès (par exemple: en entrée, tête, queue), celui-ci est sauvegardé à l’aide d’un pointeur temporaire afin de traiter les exceptions (via le mécanisme "try…catch") et de reconstruire une structure correcte.

Présentation

Les listes, spécificités et contraintes d'utilisation :

Une liste est une structure complexe et linéaire pour laquelle les opérations élémentaires ne sont pas limitées à l’ajout et au retrait d'éléments, comme pour les piles et les files.

Les limites liées aux spécificités et contraintes d'utilisation des autres structures linéaires font que les listes sont en général complémentaires aux tableaux, piles et files.

La notion de liste se distingue des autres structures linéaires par le fait que l’accès aux données ne se fait pas à l’aide d’un indice, ni grâce à un point d’accès unique.

Leur implémentation peut être simulée à l’aide d’un tableau. Mais le plus courant reste l’emploi de la technique de chaînage (simple ou double) associée à la création de maillons dans lesquels sont insérées les données.

Tous ces concepts reposent sur les principes de réservation dynamique de la mémoire et de

gestion de points d'accès aux données.

La technique de chaînage est aussi la solution la plus employée dans la création de piles ou de files. [à suivre …]

Le concept fondamental permet une gestion dynamique de la structure et de son contenu, par exemple : gestion d'une file en liste chaînée avec la possibilité de modifier l'ordre des priorités des éléments insérés ; gestion d'une pile dont le retrait d'éléments est autorisé en dehors du sommet ; …

Les structures pile et file, aux chapitres suivants, peuvent être construites sur la base d'une liste (chaînée ou non). Leur implémentation doit garantir les spécificités et contraintes d'utilisation.

Liste simplement chaînée

Chaque case mémoire est composée de deux champs. Le premier, à gauche, reçoit la donnée et le second, à droite, correspond au point d'accès (pointeur) vers la prochaine case mémoire.

Le pointeur de fin de structure est NUL, puisque la case mémoire suivante n'existe pas (cas du dernier maillon de la liste).

Rôle d'une liste simplement chaînée

Le chaînage simple est une structure simple et flexible, nécessitant un accès séquentiel et orienté (à cause de la gestion d'un unique "pointeur vers…" pour chaque maillon).

La capacité mémoire est réduite au minimum grâce à l'unique pointeur pour chaque maillon.

Cela permet aussi d'intervenir sur la liste aisément et rapidement, (au détriment de la sécurité, en comparaison avec une structure obtenue à l'aide d'un double chaînage, à suivre …).

Sur le plan « organisation de la mémoire », chaque maillon, étant pris individuellement, ils viennent « boucher des trous » de mémoire lorsque la capacité de ceux-ci est adaptée.

En revanche, la recherche nécessite un balayage de tous les éléments de la liste à partir du premier (contrairement aux tableaux indicés), ce qui représente un processus relativement long lorsque la liste l'est aussi.

Liste doublement chaînée

Chaque case mémoire est composée de trois champs. Le champ central reçoit la donnée. Le

premier, à gauche, correspond à un point d'accès vers la case mémoire précédente et le troisième, à droite reçoit le point d'accès vers la prochaine case mémoire. Chaque pointeur terminal est NUL (début ou fin de la liste).

Rôle d'une liste doublement chaînée

Le chaînage double est une structure complexe, efficace et flexible, nécessitant un accès séquentiel et bidirectionnel (grâce à la gestion des deux "pointeurs vers…" pour chaque maillon).

La capacité mémoire n'est pas optimale (rapport entre la taille de la donnée dans maillon et la taille cumulée des deux pointeurs).

Le doublement des points d'accès permet d'intervenir sur la liste aisément et rapidement (malgré les réaffectations de pointeurs) sans recourir à un balayage systématique depuis le début de la chaîne.

Sur le plan « sécurisation des données », les processus qui partagent des ressources mémoire sont susceptibles de « couper » un lien entre deux maillons. Dans ce cas, un processus de contrôle peut détecter une rupture et reconstituer le chaînage sans perte.

Opérations élémentaires

Opération d'ajout et de retrait

Trois opérations élémentaires modifient la structure de la liste, l'ajout, l'insertion et le retrait :

AJOUTER : opération d'ajout en queue de liste le nouvel élément

INSERER : opération d'insertion à la position courante et décale l'ancien élément pointé comme élément suivant

RETIRER : opération de retrait de l'élément courant

D'autres opérations évoluées permettent d'effectuer des opérations sur la liste. Il n'existe aucune limite, aucune convention.

Primitives associées

Déclaration : ECRITURE SIMPLIFIEE

LISTE: nom_liste :ENTIER(*)

Fonctions élémentaires :

```
AJOUTER(nom_liste,element)//ajout en queue
INSERER(nom_liste,element)//insertion courante
element <-- RETIRER(nom_liste)//retrait courant
ECRIRE(nom_liste,element)//écriture courante
element <-- LIRE(nom_liste)//lecture courante
```

Fonction complémentaire :

```
var_booleen <-- RECHERCHER(nom_liste,element) //recherche un élément particulier
```

*nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Fonctions élémentaires de déplacement* et de test :

```
SUIVANT(nom_liste)//se positionne sur suivant
PRECEDENT(nom_liste)//se positionne sur précédent
var_booleen <-- EST_TETE(nom_liste)//test
var_booleen <-- EST_QUEUE(nom_liste)//test
var_booleen <-- EST_VIDE(nom_liste)//test
var_booleen <-- EST_PLEINE(nom_liste)//test
```

Fonction complémentaire :

```
VIDER(nom_liste) //ordonne le retrait de tous les éléments
```

*nous utilisons ici la notion de position courante de l'élément pointé.

Déclaration : ECRITURE ORIENTEE OBJET

```
LISTE: nom_liste :ENTIER(*)
```

Fonctions élémentaires :

```
nom_liste.AJOUTER(element)//ajout en queue
nom_liste.INSERER(element)//insertion courante
element <-- nom_liste.RETIRER()//retrait courant
nom_liste.ECRIRE(element)//écriture courante
element <-- nom_liste.LIRE()//lecture courante
```

Fonction complémentaire :

```
var_booleen <-- nom_liste.RECHERCHER(element) //recherche un élément particulier
```

*nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Fonctions élémentaires de déplacement* et de test :

nom_liste.SUIVANT()//se positionne sur suivant
nom_liste.PRECEDENT()//se positionne sur précédent
var_booleen <-- nom_liste.EST_TETE()//test
var_booleen <-- nom_liste.EST_QUEUE()//test
var_booleen <-- nom_liste.EST_VIDE()//test
var_booleen <-- nom_liste.EST_PLEINE()//test

Fonction complémentaire :

nom_liste.VIDER() //ordonne le retrait de tous les éléments

*nous utilisons ici la notion de position courante de l'élément pointé.

Les algorithmes des primitives

Exemple

EXEMPLE, Gestion d'une course (quinté) avec une liste :

L'utilisation d'une liste est parfaitement adaptée dans le traitement d'informations sujettes à modification, mise en ordre, suppression, …

AJOUTER (l_quinte,"1")
AJOUTER (l_quinte,"4")
AJOUTER (l_quinte,"5")
AJOUTER (l_quinte,"7")
AJOUTER (l_quinte,"10")
AJOUTER (l_quinte,"15")
AJOUTER (l_quinte,"23")
AJOUTER (l_quinte,"25")

Les chevaux arrivés au-delà de la 5ème place sont retirés de la liste, …

RECHERCHER(l_quinte,"10")
RETIRER(l_quinte)
RECHERCHER(l_quinte,"4")
RETIRER(l_quinte)
RECHERCHER(l_quinte,"25")
RETIRER(l_quinte)

Les chevaux sont arrivés dans l'ordre suivant, un algorithme prend en charge la réorganisation de la liste, en mémoire, cela n'a d'incidence que sur le second champs "pointeur vers …".

premier : ("23")
second : ("1")
troisième : ("7")
quatrième : ("15")
cinquième : ("5")

Implémentation à l'aide de tableaux

Transmission à un terminal avec une liste simplement chaînée, implémentée à l'aide de tableaux :

Liste des lettres : "B", "O", "N", "J", "O", "U", "R".

Gestion d'une course (quinté) avec une liste doublement chaînée, implémentée à l'aide de tableaux :

Liste des chevaux : "1", "4", "5", "7", "10", "15", "23", "25".
Retrait des chevaux arrivés au-delà de la 5ème place.
L'ordre d'arrivée est le suivant : "23", "1", "7", "15", "5".

Présentation

Les piles, spécificités et contraintes d'utilisation :

Une pile est une structure linéaire, en forme de liste ordonnée et basée sur le moment de l'insertion, et pour laquelle les opérations d'ajout et de retrait d'élément ne sont effectuées qu'à une seule extrémité appelée : le sommet de la pile (ou tête).

Une pile est une structure de données du type LIFO (Last In First Out), ce qui correspond à "dernier entré premier sorti". Ce mode de fonctionnement est impératif et aucune dérogation ne peut être autorisée (à part une opération qui consiste à vider la pile).

Une pile est déclarée pour un type d'éléments. Tous les éléments de la pile doivent être du même type ! (hors cas d'une pile hétérogène traitée avec chaînage ou tableau de pointeurs sur pointeurs)

L'implémentation d'une pile se limite (ici) à sa simulation via un tableau. Il existe aussi une technique de chaînage (à suivre …).

Le rôle d'une pile

Une pile est une structure qui permet à un processus informatique de "se souvenir" pendant un certain temps d’une information ou d'une suite d'informations.

L’ordre constitué des éléments est primordial par rapport aux traitements en cours qui utilisent la pile.

Ce qui est "empilé" est considéré comme étant des éléments non encore traités, du travail restant à effectuer :

Pile permettant de mémoriser les résultats intermédiaires d’un calcul.

Gestion des variables locales lors des appels d’une fonction (étudié dans les modules suivants).

Emploi de la récursivité.

Opérations élémentaires

Opération d'ajout et de retrait

Seules deux opérations ont la possibilité de modifier le contenu de la pile : l’ajout et le retrait, qui portent des noms spécifiques pour cette structure :

EMPILER : opération d’ajout

DEPILER : opération de retrait

Les termes EMPILER et DEPILER sont souvent employés, mais aussi AJOUTER et RETIRER…

Primitives associées

Déclaration : ECRITURE SIMPLIFIEE

PILE: nom_pile :ENTIER(*)

Fonctions élémentaires :

```
EMPILER(nom_pile,element) //ajout tête/sommet
element <-- DEPILER(nom_pile) //retrait tête
element <-- TETE(nom_pile) //lecture seule
var_booleen <-- EST_VIDE(nom_pile) //test
var_booleen <-- EST_PLEINE(nom_pile) //test
```

Fonction complémentaire :

VIDER(nom_pile) //ordonne le retrait de tous les éléments

(*)nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Déclaration : ECRITURE ORIENTEE OBJET

PILE: nom_pile :ENTIER(*)

Fonctions élémentaires :

```
nom_pile.EMPILER(element) //ajout tête/sommet
element <-- nom_pile.DEPILER() //retrait tête
element <-- nom_pile.TETE() //lecture seule
var_booleen <-- nom_pile.EST_VIDE() //test
var_booleen <-- nom_pile.EST_PLEINE() //test
```

Fonction complémentaire :

nom_pile.VIDER() //ordonne le retrait de tous les éléments

(*)nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Les algorithmes des primitives

Exemple

EXEMPLE, Suite de calculs à l'aide d'une pile :

L’utilisation d’une pile est parfaitement adaptée dans les évaluation d’expressions avec parenthèses, comme par exemple : 5 + (9 + 8) * 3

```
EMPILER( 5 )
EMPILER( 9 )
EMPILER( 8 )
EMPILER( DEPILER() + DEPILER() )
EMPILER( 3 )
EMPILER( DEPILER() * DEPILER() )
EMPILER( DEPILER() + DEPILER() )
ECRIRE( DEPILER() )
```

Implémentation à l'aide d'un tableau

Suite de calculs gérée avec une pile implémentée à l'aide d'un tableau :

Opération à effectuer : 5 + (9 + 8) * 3

Présentation

Les files, spécificités et contraintes d'utilisation :

Une file est une structure linéaire, en forme de liste ordonnée et basée sur le moment de l'insertion, pour laquelle les opérations d'ajout sont effectuées à une seule extrémité appelée : la queue de la file ; et les opérations de retrait sont effectuées à l'autre extrémité appelée : la tête de la file.

Une file est une structure de donnée du type FIFO (First In First Out), ce qui correspond à "premier entré premier sorti". Ce mode de fonctionnement est impératif et aucune dérogation ne peut être autorisée (à part une opération qui consiste à vider la file).

Une pile est déclarée pour un type d'éléments. Tous les éléments de la pile doivent être du même type ! (hors cas d'une file hétérogène traitée avec chaînage ou tableau de pointeurs sur pointeurs)

La notion de file en informatique, est identique à celle d'une file d'attente (comme au cinéma, …).

Ce concept est courant en mathématiques (théorie des graphes, …). En informatique, le recours à un tel type de structure est très courant, par exemple : gestion de la file d'attente des impressions sur une imprimante en réseau, traitement des transmissions, gestion des ressources mémoire et des processus, des accès disques, ...

Pour résumer, les éléments d'une file sont traités dans leur ordre d'arrivée. L'implémentation d'une file se limite (ici) à sa simulation via un tableau. Il existe aussi une technique de chaînage (à suivre …).

Le rôle d'une file

D'une manière générale, on utilise les files :

Lorsqu'il existe un décalage entre des demandes de traitement et le traitement effectif de ces demandes,

Lorsqu'il y a une notion d'ordre.

Très souvent, lorsqu'il y a une obligation ou une simple notion de priorité, la structure d'une

file est généralement la plus utilisée. La principale raison de ce choix vient du fait qu'il existe une distinction claire et sans ambiguïté entre tête et queue, et que (conjointement) l'élément de tête possède toujours la plus haute priorité.

Le moment d'insertion sert de critère de priorisation des éléments les uns par rapport aux autres. Ensuite, l'ordre ne peut plus changer !

Opérations élémentaires

Opération d'ajout et de retrait

Seules deux opérations ont la possibilité de modifier le contenu de la file : l'ajout et le retrait, qui portent des noms spécifiques pour cette structure :

ENFILER : opération d'ajout

DEFILER : opération de retrait

Les termes ENFILER et DEFILER sont souvent employés, mais aussi AJOUTER et RETIRER…

Primitives associées

Déclaration : ECRITURE SIMPLIFIEE

FILE: nom_file :ENTIER(*)

Fonctions élémentaires :

ENFILER(nom_file,element) //ajout en tête
element <-- DEFILER(nom_file) //retrait queue
element <-- TETE(nom_file) //lecture seule
var_booleen <-- EST_VIDE(nom_file) //test
var_booleen <-- EST_PLEINE(nom_file) //test

Fonction complémentaire :

VIDER(nom_file) //ordonne le retrait de tous les éléments

(*)nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Déclaration : ECRITURE ORIENTEE OBJET

FILE: nom_file :ENTIER(*)

Fonctions élémentaires :

```
nom_file.ENFILER(element) //ajout en queue
element <-- nom_file.DEFILER() //retrait tête
element <-- nom_file.TETE() //lecture seule
var_booleen <-- nom_file.EST_VIDE() //test
var_booleen <-- nom_file.EST_PLEINE() //test
```

Fonction complémentaire :

```
nom_file.VIDER() //ordonne le retrait de tous les éléments
```

(*)nous utiliserons entier ou caractère dans l'exercice de simulation à l'aide d'un tableau.

Les algorithmes des primitives

Exemple

EXEMPLE, Transmission à un terminal à l'aide d'une file :

L’utilisation d’une file est parfaitement adaptée dans le traitement d'informations transmises entre deux terminaux, comme par exemple : "BONJOUR"

```
ENFILER( 'B' )
ENFILER( 'O' )
ENFILER( 'N' )
ENFILER( 'J' )
ENFILER( 'O' )
ENFILER( 'U' )
ENFILER( 'R' )
ECRIRE( DEFILER(), DEFILER(), DEFILER(), DEFILER(), DEFILER(), DEFILER(), DEFILER() )
```

Implémentation à l’aide de tableaux

Transmission à un terminal avec une file implémentée à l'aide de tableaux :

Liste des lettres : "B", "O", "N", "J", "O", "U", "R".

Comme cela a été montré dans le cours, il est fait appel ici, à l’emploi d’un ou plusieurs tableaux afin de simuler le fonctionnement de ces structures linéaires.

L’objectif, ici, réside en l’écriture des structures algorithmiques des principales primitives associées aux structures de données linéaires (pile, file, liste).

Primitives pour une pile, une file et une liste

Algorithmes pour une PILE

EMPILER
DEPILER
EST_VIDE
TETE

Algorithmes pour une FILE

ENFILER
DEFILER
EST_VIDE
TETE

Algorithmes pour une LISTE

LECTURE
ECRIRE
AJOUTER
RETIRER

Algorithmes complémentaires pour une LISTE

EST_VIDE
RECHERCHER
MAX
MIN

Gestion de la taille de chaque structure

La gestion de la taille d'une structure construite dynamiquement est fortement conseillée.
Reprendre les algorithmes ci-dessus et sécuriser ces structures.

Structure de programme

Présentation

Structure d’un programme

Décomposition et organisation en modules dont l’objectif est de prendre en charge de façon autonome des tâches distinctes -parties de programmes (sous-programmes)- contenues dans le programme (principal).

Découpage en modules

Objectifs de la structuration d'un programme

Simplifier le programme global grâce à une encapsulation des tâches distinctes et hiérarchisables.

Réduire la taille du programme (code exécutable) en mémoire (hors cas de la programmation à l’aide de « inline », mise « en ligne » par recopie systématique du sous-programme appelé).

Réduire la taille de l'écriture algorithmique.

Travailler en équipes.

Obtenir, en final, un programme de qualité grâce à une validation systématique de chaque entité modularisée, une maintenance aisée, une lisibilité accrue, la possibilité d'exploiter des modules dans d'autres programmes grâce à la constitution de bibliothèques, …

Méthodologie

En phase d’analyse et de conception, application du principe « Diviser pour régner » :

Analyser le problème à traiter afin de le rendre séquentiel lorsque cela est possible.

Toute tâche répétitive, autonome, hiérarchisable doit être à l’origine de la construction d’un module.

Application récursive :

Répéter ce processus de décomposition au sein des modules.

Paramétrer les liens entre le Programme Principal et les Modules :

Les modules disposent de paramètres leur permettant de communiquer dans le sens (unique) :

programme principal → sous-programme,

sous-programme → sous-sous-programme, etc ...

Spécialiser le sous-programme en accordant une donnée (en retour), à sa propre fonction.

Standardiser le plus possible les développements :

Eviter de refaire une partie de programme qui existe déjà,

Prévoir un développement futur connexe,

Anticiper une évolution naturelle du programme.

Préférer la simplicité et la validité face à la créativité.

Le Programme Principal, peut faire appel à des procédures ou fonctions diverses, ce qui a pour effet de structurer le programme, de réduire sa place en mémoire ainsi que son écriture, et surtout, de faciliter sa maintenance.

Procédure et fonction

Langage algorithmique et langages structurés

En algorithmique, il est courant d'utiliser deux sortes de sous-programme:

les procédures,

les fonctions.

En langage évolué (ou langage de programmation), ces deux formes de sous-programme n'existent pas forcément. [Les procédures ont tendance à ne plus être intégrées dans certains langages structurés et évolués.]

Notion de « classes » :

D'autres formes de programmation (orientée objet, …) exploitent le même principe de décomposition en modules. Ce sont des langages élaborés à partir de sous-programmes évolués dont le concept « objet » accroît le principe de hiérarchisation (les fonctions sont attachées à un objet, …).

Programmation structurée à l'aide de modules :

Un programme structuré à l'aide de modules est composé d'un (et un seul) programme

principal (main).

A l'intérieur de ce programme principal, se trouvent des appels aux modules appartenant au programme ou simplement liés à celui-ci.

Les modules d'un programme :

Un module peut appelé un autre module.

Un module peut s'appeler lui-même, cette particularité se nomme : la récursivité.

Différence entre Procédure et Fonction :

Procédure : ne retourne pas de donnée

Fonction : au terme de son appel, retourne une donnée unique en relation avec sa fonction propre (facultatif)

Langage algorithmique

Chaque procédure est caractérisée par un nom unique (identificateur), des variables locales (facultatives) nécessaires à leur propre fonctionnement (sorte de variable temporaire), une structure algorithmique (interne au module) encadrée par les mots clés : début et fin (comme pour un programme).

Chaque fonction est caractérisée par un nom unique (identificateur), des variables locales (facultatives) nécessaires à leur propre fonctionnement (sorte de variable temporaire), une structure algorithmique (interne au module) encadrée par les mots clés : début et fin (comme pour un programme).

Ces sous-programmes ne peuvent pas s'exécuter indépendamment d'un programme principal.

Représentation graphique

Structure de programme : approche graphique

Exemple de l'initialisation du jeu d'échec

Le Programme Principal "preparer_jeu_echec" appelle trois sous-programmes distincts. Dans cette représentation graphique, on retrouve l'ordre des appels, mais pas le nombre d'appels. C'est l'arbre des relations entre les modules et le programme principal.

Qualités d’un programme informatique

Les principales qualités d'un programme

Parmi les principales qualités d’un programme informatique se trouve la modularité.
Aussi, la décomposition d’un problème en modules facilite l’aboutissement d’autres critères de qualité.

Validité
Convivialité
Transportabilité
Maintenance aisée
Modularité
Sécurité
Sûreté
Immunité
Lisibilité
Optimalité mono ou multi-objectif
Simplicité

Présentation

Programmation modulaire

Un programme modulaire est composé d'un (et un seul) programme principal.

A l'intérieur de ce programme principal, se trouvent des appels aux modules appartenant au programme.

Les modules d'un programme :

Un module peut appeler un autre module.

Un module peut s'appeler lui-même, cette particularité se nomme : la récursivité.

Un module peut appeler un autre module qui appelle le module appelant, cette spécificité se nomme : la récursivité croisée.

Lancement d'un programme (ou logiciel)

Lors du lancement d'un programme, la première partie prise en compte par l'ordinateur est celle relative aux données (dites "statiques"):

L'ordinateur réserve l'espace mémoire nécessaire et suffisant au fonctionnement du programme.

Cette opération est réalisée grâce aux déclarations des données et de leur type.

Ensuite, le programme commence à exécuter les instructions notées après le mot clé "debut".

Programme principal

Structure du programme principal

Procédure et fonction

Une procédure ou une fonction est un sous-programme qui a pour objectif de réaliser une tâche particulière (et une seule) qui lui est propre.

Points communs entre une procédure et une fonction :

Les sous-programmes disposent d'une zone propre de réservation et de stockage des données, en fonction du nombre et du type de celles-ci.

Afin de faire communiquer le programme et le sous-programme, il est possible (mais facultatif) de transmettre un ou plusieurs paramètres au module appelé (c'est à dire, la valeur de certaines variables utiles au module).

Différences fondamentales entre procédure et fonction

Une procédure ne peut pas transmettre de valeur issue de son traitement au programme (ou sous-programme) appelant.

Dans les langages de programmation qui utilisent les procédures, il n’existe pas de restriction d’accès aux données du programme appelant.

Une fonction peut transmettre (facultativement) une et une seule donnée au programme appelant.

Cette donnée, étant unique, peut être simple ou complexe, c’est-à-dire représenter une structure composée de plusieurs données.

Programme principal et procédure

Programme principal et fonction

Organisation : Programme principal et fonctions

Paramètre formel et paramètre réel

Paramètre :

On appelle "paramètres", les noms des variables placés dans la déclaration d'une procédure ou d'une fonction, ou dans l'écriture de son appel.

Paramètres de déclaration : Ce sont des variables locales au module, et à ce titre, ils sont déclarés dans l'entête du sous programme, entre parenthèses : ce sont les paramètres formels.

Paramètres d'appel : Inscrits entre parenthèses, lors de l'appel, ils servent de liens entre l'appelant et le sous programme prédéfini : ce sont les paramètres effectifs appelés communément les arguments.

Paramètre effectif (ou réel, ou argument) :

Lorsque les paramètres sont de type "donnée" (et non un adressage à l'aide d'un pointeur sur la donnée), ils contiennent effectivement les valeurs avec lesquelles sera effectué le traitement du sous-programme.

Lors de l'appel, leur valeur est recopiée dans les paramètres formels correspondants à l'emplacement prédéfini (l'ordre est conservé).

Un paramètre effectif en donnée peut être :

soit une variable (locale) du programme appelant,

soit une valeur littérale (ECRIRE("Bonjour")),

soit le résultat d'une expression (ECRIRE(a+1)).

Passage de paramètre par valeur

Passage de paramètre par adresse

Passage de paramètre par pointeur

Définition

La récursivité

Concept fondamental :

en mathématique :

relation de récurrence :

n! = n(n - 1)!
pour n >= 1 avec 0! = 1

en informatique :

fonctions récursives.

Le principe de la récursivité en algorithmique

Qui s'appelle lui-même.

Appels de plus en plus « simples ».

Solution directe pour le dernier appel.

Respect de la condition d'arrêt.

Les types de récursivité

Récursivité simple : procédure ou fonction qui s'appelle elle-même.

Récursivité croisée : Un sous programme SP1 appelle un sous programme SP2 qui rappelle le sous programme SP1.

Dans tous les cas, il faut s'assurer que le sous programme ne s'appelle pas indéfiniment.

Existence d'une condition de sortie de la boucle récursive.

Instruction répétitive et récursivité

Exemple, Calcul non récursif de la factorielle

Exemple, Calcul récursif de la factorielle

Intérêts et contraintes

Intérêts de la récursivité

Technique efficace de simplification de l'écriture d'un algorithme de résolution d'un problème.

L'algorithme récursif est défini (et élaboré) en fonction de lui-même.

Certaines structures de données peuvent contenir une même structure en interne (un

répertoire d'ordinateur est composé de fichiers et de dossiers, ces dossiers peuvent contenir d'autres fichiers et dossiers, quasiment à l'infini…).

Un problème, dans certains cas, peut se ramener au même problème de taille inférieure et ainsi de suite jusqu'à obtenir un problème élémentaire. Le concept de récursivité est parfaitement adapté avec cette catégorie de problèmes.

Par exemple, pour le calcul du factorielle de N :   N !

Contraintes de la récursivité

S'assurer que la fonction est complètement définie sur tout son domaine d'application.

Etre certain que l'algorithme aboutira sur un cas connu : le cas d'arrêt de la récursivité.

Contrôler que le contexte d'utilisation est parfaitement adapté aux inconvénients de la récursivité en informatique : besoins en ressources systèmes (plus particulièrement en gestion de l'espace mémoire) ; les variables temporaires du programme ou de la fonction qui provoque l'appel sont sauvegardées temporairement dans une structure de type "pile" (vérifier la taille de la pile et le type de la pile : allocation statique ou dynamique).

Exemple

Exemple, Calcul de la factorielle :   fact(3)

Structure de donnée associée

Gestion de la pile en récursivité :   fact(3)

Une pile est utilisée lors des appels récursifs :

EMPILER(3)
EMPILER(2)
EMPILER(1)
DEPILER()
DEPILER()
DEPILER()

Construction et paramétrage des modules

Primitives pour une pile, une file et une liste

Reprendre les exercices du chapitre VII sur les principales primitives associées à une pile, une file et une liste, et construire des procédures.

Calcul de la moyenne d'un élève

Construire les procédures de modification des notes du programme au chapitre VI pour calculer la moyenne ainsi que la fonction qui retourne la moyenne à la fonction qui détermine la mention associée.

Construction d'une fonction récursive

Durée d'une plongée avec bouteille

Reprendre le problème du plongeur et construire une fonction récursive qui retourne la durée totale de la plongée en fonction d'une profondeur donnée.

Structures non linéaires de données II

Les graphes et les arbres

Structure non linéaire de données

En informatique, il est souvent nécessaire d'employer des structures complexes basés sur les éléments de la Théorie des Graphes : les nœuds (ou sommets) et les relations entre ces nœuds (arcs ou arêtes).

Une bonne implémentation de ce type de structures est important, car la performance des algorithmes qui les utilisent s'en trouve directement affectée.

Dans ce module, seront présentés :

Les graphes.
Les arbres.

Les graphes et les arbres ne font pas toujours l'objet d'une implémentation informatique avec le principe de chaînage étudié pour construire une liste.

Les structures complexes qui représenteront des graphes et des arbres multibranches dans un programme, peuvent être construites à l'aide :

Une ou plusieurs matrices (tableau à deux dimensions).
Plusieurs listes (tableau ou liste chaînée).

Dans le cas d'arbres binaires, les structures chaînées (maillons constitués de trois ou quatre champs), seront très efficaces.

Présentation

Les deux catégories de graphes :

Le graphe orienté

Le graphe non orienté

Implémentation informatique

Les moyens d'implémentation des graphes

Graphes orientés :

Matrice :
  matrice d'incidence

Liste :
  liste d'incidence

Graphes non orientés :

Matrice :
  matrice d'adjacence

Liste :
  liste d'adjacence

Matrice d'adjacence (Contexte non orienté)

Graphe non orienté :

Matrice d'adjacence
  sommet/sommet
  sommet/arête

Liste d'adjacence (Contexte non orienté)

Graphe non orienté :

Liste d'adjacence
  sommet/sommet
  sommet/arête

Matrice d'incidence (Contexte orienté)

Graphe orienté :

Matrice d'incidence
  sommet/sommet
    Successeurs
      Prédécesseurs

Matrice d'incidence
  sommet/sommet
    Successeurs {+1}
    Prédécesseurs {-1}

Matrice d'incidence
  sommet/arête
    Successeurs
    Prédécesseurs

Matrice d'incidence
  sommet/arête
    Successeurs {+1}
    Prédécesseurs {-1}

Liste d'incidence (Contexte orienté)

Liste d'incidence
  sommet/sommet
    Successeurs
    Prédécesseurs

Liste d'incidence
  sommet/arête
    Successeurs
    Prédécesseurs

Le choix d’une structure de données

Les critères d'informations à prendre en compte en vue d'une implémentation des graphes (hors traitement).

Les sommets (ou noeuds) :

Le nombre total de sommets.

Les relations entre les sommets :

Le nombre total de relations.
Relation simple ou multiple entre deux sommets.
Relation orientée ou non orientée.
Valeur associée à la relation :
Type de la valeur de la relation (entier, réel, composé, …).

Le rapport entre le nombre de sommets et le nombre de relations.

Présentation

Les deux catégories : arbres et arborescences
Les arbres

Les arborescences

Terminologie

Modélisation sous une forme arborescente
Profondeur
Noeud/Sommet
Racine
Feuille
Arête

Implémentation informatique

Les arbres et variantes sur les arbres :

Les arbres sont des "graphes" spécifiques.

Propriété fondamentale :
Nbre liaisons = Nbre sommets - 1
Leur modélisation en informatique peut être envisagée, à l'aide des structures utilisées pour représenter des graphes (abordées dans le chapitre précédent) :
Les matrices.
Les listes.

Cependant, certaines structures sont mieux adaptées que d'autres. Les matrices ("sommet/sommet") vont nécessiter plus de ressources que des listes. Aussi, la technique de chaînage reste la solution la plus courante et très efficace.

Exemple

L'implémentation des arbres :

Matrice d'adjacence
  sommet/sommet

Matrice "père/fils"
  sommet/sommet
    ascendant de: {+1}
    descendant de: {-1}

Liste "père/fils"
  sommet/sommet
    ascendants et descendants:

Le choix d'une structure de données

Les arbres binaires

Les types d'arbres :

Plusieurs types d'arbres existent. Ils sont liés aux spécificités du cadre d'emploi et aux algorithmes de traitement :
Les arbres génériques ou multi branche.
Les arbres binaires.
Les arbres ordonnés.
Les arbres binaires de recherche.
Les arbres équilibrés.
Les arbres complets.
voire … une association des propriétés ci-dessus.

Les arbres binaires :

Les arbres binaires sont des arbres spécifiques, très souvent implémentés en informatique.

Tout arbre "générique" (ou multi branche), peut être transformé en un arbre binaire.

Le but d'une telle structure est multiple :
Organiser l'arborescence afin d'ordonner sa structure sur son contenu,

Disposer d'une dichotomie directement, avec le choix "fils gauche" ou "fils droit" donné par la structure d'un noeud limité à deux liens maximum,

Employer des algorithmes de recherche simple, des fonctions récursives, sans parcours de listes de noeuds, …),

Effectuer des recherches efficaces.

Les arbres binaires de recherche sont des arbres binaires dont les noeuds sont renseignés à l'aide d'une information "clé", (implémentation plus efficace qu'un arbre multi branche).

Définition :

Un arbre binaire est un arbre multi branche dont le nombre maximum de fils pour un noeud quelconque ne doit pas dépasser 2.

Construction d'un arbre binaire à partir d'un arbre multi branche

Règles de construction :
-1- un père avec fils unique est représenté avec ce fils à gauche.

-2- un père avec deux fils est représenté avec le fils gauche à gauche et le fils droit comme fils droit de ce fils gauche.

-3- chaque fils supplémentaire est représenté comme fils droit du fils droit inséré (voir règle 3).

Les arbres binaires de recherche :

Propriétés d'un arbre binaire de recherche :
L'arbre est constitué d'un ensemble fini d'éléments totalement ordonnés.

Chaque élément est pourvu d'une "clé" (valeur entière).

Les clés sont toutes distinctes.

L'objectif de la construction d'un tel arbre est à recherche d'une clé, à l'aide d'un parcours simple. Règles de base implémentées pour cet arbre : parcours en profondeur d'abord et ordre croissant des clés {3;5;8;10;12;15;16;17}.

Les règles de construction sont :
Pour un fils gauche : le père est de valeur inférieure et le frère de valeur supérieure.

Pour un fils droit : le père est de valeur inférieure et le frère aussi.

L'implémentation des arbres binaires :

Structure "père/fils"
  Sommet (noeud)
    clé:
    fils_gauche:
    fils_droit:

```
//Exemple de structure d'un noeud:
ENREGISTREMENT: noeud
DEBUTENREGISTREMENT
        cle :ENTIER //valeur de la clé du noeud
        *ptr_filsgauche :noeud //pointeur sur un noeud gauche
        *ptr_filsdroit  :noeud //pointeur sur un noeud droit
FINENREGISTREMENT
```

Construction de fonctions pour analyser un graphe matriciel

Soit un graphe non orienté représenté par une structure matricielle (tableau de type booléen, à deux dimensions, relations sommet/sommet).

Construire les fonctions qui permettent de répondre aux questions suivantes :

Existe-t-il une relation entre deux sommets donnés ?

Existe-t-il une relation à partir d'un sommet donné ?

Existe-t-il une relation à destination d'un sommet donné ?

Existe-t-il une relation au moins un sommet isolé (c'est-à-dire sans liaison avec d'autres sommets) ?

Combien de sommets sont en relation avec un sommet donné ?

Construction de fonctions pour analyser un arbre matriciel

Soit un arbre représenté par une structure matricielle (tableau de type booléen, à deux dimensions, relations sommet/sommet).

Construire les fonctions qui permettent de répondre aux questions suivantes :

Ce sommet, est-il une racine ?

Ce sommet, est-il une feuille ?

Cet arbre, est-il un arbre binaire ?

Construction de fonctions récursives pour analyser un arbre chaîné

Soit un arbre binaire représenté par une structure chaînée (enregistrement à trois champs de type pointeur, un champ de type donnée, par exemple un entier).

Construire les fonctions qui permettent de répondre aux questions suivantes :

Trouver et retourner la valeur de la donnée de la racine ?

Trouver et retourner la valeur de la donnée d'une feuille ?

Programme et système d'exploitation

Environnement de programmation

Qu'est-ce que l'environnement de programmation ?

Un programme est destiné à fonctionner sur un ordinateur. Mais tous les ordinateurs sont différents !

Les différences les plus importantes sont :
Au niveau matériel :

Microprocesseur, mémoire, périphériques internes, périphériques externes, …

Au niveau logiciel :

Système d'exploitation.

Le programmeur doit, par conséquent, intégrer les spécificités de ces divers éléments avant de définir l'ensemble des contraintes de programmation liées à cet environnement, (donc, avant de traduire un algorithme !).

Quelles sont les conséquences sur la programmation ?

Les langages, ces dix dernières années, évoluent vers une normalisation. Même si cela permet de réduire les contraintes présentées précédemment, le programmeur est obligé de maîtriser les éléments associés à cette norme.

Plus précisément, le concepteur devra, parmi l'ensemble des contraintes de programmation liées à l'environnement du logiciel à réaliser :
Choisir un langage approprié,

Connaître les bibliothèques associées au langage,

Analyser certaines contraintes afin de construire ses propres bibliothèques,

…

Bibliothèques

Les grandes familles de bibliothèques

Chaque compilateur (ou EDI: Environnement de Développement Informatique) est fourni avec une série de bibliothèques.

Parmi ces bibliothèques, les plus importantes sont :
Les bibliothèques standards dédiées à un langage.

Les bibliothèques spécialisées (comme, par exemple, les bibliothèques de classes MFC: Microsoft Foundation Classes).

D'autres contraintes de développement orienteront le choix d'un type de bibliothèque plutôt qu'un autre. (En général, le développeur se spécialise dans une famille de bibliothèque).

Des bibliothèques algorithmiques !

En algorithmique, on tente de conserver un niveau d'abstraction suffisant pour ne pas alourdir l'objectif de simplification de la structuration algorithmique.

Ainsi, les algorithmes sont accompagnés des seules bibliothèques conçues par le programmeur, soit pour pallier l'inexistence d'une fonction particulière dans les bibliothèques courantes, soit pour en adapter une au besoin spécifique du moment.

C'est donc bien en phase de traduction d'un algorithme en langage de programmation, que le concepteur fait intervenir toutes les particularités du langage utilisé et des bibliothèques choisies.

Flux de données

Qu’est-ce que “les flux de données” ?

En informatique, on ne trouve que très rarement des programmes qui fonctionnent en complète « autarcie » !

Il est souvent nécessaire de traiter des données inconnues lors de l’écriture du programme, de même que l’affichage des résultats s’impose d’une manière ou d’une autre.

Ces informations, qui circulent entre l’ordinateur et son environnement, doivent, par conséquent, être encadrées afin de ne pas faire d’erreurs d’interprétation, sous peine de fournir un résultat faussé, ou bien de bloquer le programme, au pire l’ordinateur lui-même !

Ces données sont appelées des flux (en anglais: stream). Ceux-ci permettent aux programmes de dialoguer avec l’écran, le clavier, une imprimante, mais aussi les fichiers et surtout la mémoire.

Gestion des entrées/sorties

Comment sécuriser “les flux de données” ?

Malgré la diversité des ordinateurs sur la marché, la cohérence des informations captées ou émises doit être garantie.

Sur le plan « écriture algorithmique », la gestion des entrée/sorties, utilisée jusqu’à maintenant, ne prenait pas en compte les risques inhérents à une mauvaise gestion de l’environnement de programmation. [La dernière partie de ce module traite essentiellement de ce problème]

Il existe un moyen de prendre la main sur un risque de ce type et de contrôler la fin de l’exécution du programme qui a rencontré un défaut majeur durant la gestion de son environnement.

Aussi, s’il existe une écriture algorithmique équivalente à ce dispositif de protection des programmes, le concepteur intègrera directement en langage de programmation cet outil, plutôt que dans l’algorithme en pseudo-code.

Gestion des entrées/sorties

Formatage implicite des données

Les informations qui circulent entre l’ordinateur et son environnement, sont réparties en deux groupes :
Les informations qui entrent : elles sont nécessaires et attendues par le programme.

Les informations qui sortent : elles sont les productions issues du déroulement des algorithmes.

Ces informations ont en commun, un point important, en relation avec les données traitées : le type élémentaire* choisi par le programmeur lors de la conception !

Ces données sont représentées suivant un formatage précis et nécessaire pour garantir la correspondance avec la donnée affectée (flux d'entrée) ou la sortie attendue (flux de sortie).

type élémentaire* : n'est pas abordé, ici, le cas très courant où le type élémentaire d'une donnée est choisi dynamiquement !

Flux de données formatés

Formatage explicite des données

Les primitives utilisées depuis le début du cours :
LIRE()
ECRIRE()

(fonctions d'une bibliothèque algorithmique virtuelle), ont été employées comme des flux d'entrée/sortie standard, c'est-à-dire dont les canaux respectifs sont : le clavier et l'écran.

Le formatage des données et le type programmé ont toujours été considérés conformes l'un à l'autre.

Afin d'éviter les erreurs et d'améliorer la gestion des flux d'entrées/sorties, celles-ci peuvent être formatées de façon implicite, grâce à des compléments d'informations transmis au flux.

La primitive ECRIRE() sera enrichie de séquences d'échappement* ANSI pour l'obtention d'un affichage précis.

séquences d'échappement* : se baser sur le référentiel du langage choisi !

Présentation

Regroupement d'informations sur un support non volatile (comme le disque dur de l'ordinateur ou une bande magnétique), permettant la communication -via des données "globales permanentes"- d'un programme à un autre ou pour un même programme.

Les fichiers structurés : fichiers constitués de données de même nature ou de séquences (de données) de même nature.

Les fichiers non structurés : fichiers constitués d'informations de natures diverses, de séquences irrégulières "non structurables".

Deux types d'accès :

Accès séquentiel : nécessite de parcourir tous les éléments du fichier pour effectuer une action de lecture ou d'écriture (bande magnétique). Ce mode d'accès est toutefois permis sur des fichiers créés en accès direct (ex: contrôle d'intégrité, checksum), dans ce cas, le "sens" originel des informations lues reste difficilement identifiable.

Accès direct : permet de se positionner directement sur l'information voulue à partir de son emplacement sur le support (technique d'adressage, découpage en secteurs, pistes) ou d'un numéro (indexation).

Cinq grandes familles de fichiers (principaux modèles) :
[Le type d'accès est défini à la création du fichier.]
Fichier à accès purement séquentiel : soit les séquences sont strictement de même longueur, soit le sens du contenu est donné par le contenu même.

Fichier à accès séquentiel indexé* : nécessite la lecture de l'index ainsi que sa recherche (évolution du modèle purement séquentiel).

Fichier à accès direct et à structure fixe : séquences d'informations de taille prédéfinie et fixe.

Fichier à accès direct et à structure variable* : séquences d'informations de taille variable.

Fichier à accès direct et à structure complexe* : consiste en une combinaison des techniques avancées de la structuration et de liaisons (bases de données).

Dans un programme, un fichier est désigné par une variable, que l'on déclare dans la section des variables.

Pour déclarer un fichier structuré, il faut préalablement déclarer le ou les enregistrements qui composent et structurent ce fichier.

Opérations élémentaires

Un fichier implique un mode d'accès propre.

Avant d'utiliser un fichier, il faut contrôler s'il existe, le créer sinon. Lorsque celui-ci est localisé, et afin de l'utiliser, il est nécessaire de spécifier les modes d'accès à l'ouverture :
mode lecture :

permet de récupérer des données du fichier, sans être autorisé à modifier le contenu.

mode écriture :

écriture par insertion : permet de lire, ajouter et modifier le contenu librement.

écriture par ajout : permet d'ajouter en fin de fichier des nouveaux éléments (pas d'accès au reste).

Après utilisation, un fichier doit être fermé (cela permet de libérer la mémoire virtuelle réservée après la mise à jour des données sur le support physique).

La communication Ram ↔ Fichier est bidirectionnelle.

Lecture : transfert d'une information sur un fichier dans une variable de même type.

Ecriture : recopie le contenu d'une variable, soit à la position courante dans le fichier, soit à la fin du fichier (en fonction du mode d'ouverture).

Les commandes SUIVANT() et PRECEDENT() permettent d'accéder aux éléments successifs, séquentiellement.

[L'emploi de primitives plus évoluées autorise la mise en buffer d'une taille d'informations plus importante.]

Les déplacements sont effectués en contrôlant la position courante (comme pour une structure LISTE). La fin du fichier : variable_booleenne <-- fichier.FIN_DE_FICHIER()

Le début du fichier : variable_booleenne <-- fichier.DEBUT_DE_FICHIER()

Primitives associées

Primitives relatives à l'accès aux fichiers

Primitives relatives à l'utilisation des fichiers

Primitives relatives à l'accès séquentiel

Exemple

Utilisation d'une structure de fichier

Présentation

Les erreurs liées à l'environnement de programmation

Les erreurs liées à l'environnement de programmation peuvent être décomposées en plusieurs catégories :
La gestion de la mémoire.

La gestion des fichiers.

La gestion des périphériques.

Erreurs et mémoire

La gestion de la mémoire :
Une demande d'allocation (ou de réallocation) de la mémoire, peut se révéler infructueuse.

Un accès mémoire peut être réservé.

Une modification d'une partie de la mémoire peut ne pas être autorisée.

…

Erreurs et fichiers

La gestion des fichiers :

Le fichier existe déjà.

Le fichier est introuvable.

Le répertoire est introuvable.

L'accès au fichier est interdit.

L'écriture dans le fichier est réservé.

La structure du fichier est corrompue.

…

Erreurs et périphériques

La gestion des périphériques :

Le périphérique est introuvable.

Le périphérique est occupé.

…

Remarque : Une partie de la gestion complexe de cet environnement est traitée dans une « couche du système d'exploitation » inaccessible pour le programmeur.

Les exceptions

Le contrôle systématique des valeurs retournées lors de l'emploi des fonctions (ou primitives), ne permettent pas de prévenir tous les types d'erreurs au sein du système.

Toutes les exceptions « système » ne sont pas traitées ici !

Il en est de même pour les accès aux fichiers.

Avant de connaître la valeur retournée, et de la tester, un grand nombre d'exceptions peuvent être levées par le système d'exploitation !

Gestion d'une exception

La gestion des erreurs ne peut être abordée de cette façon, sur l'ensemble du programme, sous peine d'alourdir son écriture, mais aussi de réduire les performances lors de son exécution.

Il existe un mécanisme : la gestion des exceptions, qui, introduite au sein du programme, permet de centraliser l'ensemble des erreurs produites durant le déroulement du processus.

La gestion des exceptions utilise une structure algorithmique propre à son fonctionnement. Cette structure ne peut traiter que les problèmes issus des instructions d'un bloc particulier.

La structure algorithmique se présente en deux parties* :
Le premier bloc est celui susceptible de provoquer une exception. Il commence par « ESSAYER » (en C++: try).

Le second bloc correspond au traitement programmé de l'exception. Si une exception se produit au niveau du premier bloc, le second est exécuté, sinon, les instructions qui s'y trouvent ne sont jamais appliquées ! Il commence par « ATTRAPER » (en C++: catch).

deux parties* : dans les langages, la première partie peut se trouver imbriquée dans une autre de même type, la seconde partie est éventuellement multiple et spécialisée pour ne traiter qu'une seule catégorie d'exception.

Structure algorithmique pour la gestion d'une exception

Exemple d'allocation de la mémoire.

Soit: le fichier s'ouvre, alors « GESTION FICHIER SANS ERREUR » s'affiche, et le programme continue l'exécution des instructions qui suivent « FINEXCEPTION ».

Soit: l'ouverture est un échec, alors « ERREUR NON DEFINIE AVEC UN FICHIER » s'affiche, et le programme continue l'exécution des instructions qui suivent « FINEXCEPTION ».

Générer une exception

Le programmeur a la possibilité de créer ses propres gestionnaires d'exception*.

L'instruction « LEVER » (en C++: throw) fait appel au bloc gestionnaire correspondant.

En général, cette instruction se situe dans un bloc d'une instruction de sélection. Lorsque que la condition est valide, il faut dérouler le contenu du gestionnaire, l'exception est appelée.

Le déroulement est identique à une exception générée par le système d'exploitation.

gestionnaires d'exception* : l'instruction qui permet de lever une exception spécifique peut être spécialisée dans le but de définir avec précision le type d'erreur produit. Dans ce cas, un gestionnaire d'erreurs par défaut doit être ajouté après tous les autres blocs ESSAYER spécialisés.

Exemple d'allocation de la mémoire

Algorithmes et méthodes de tri à partir d'un ou plusieurs fichiers

Tri par insertion

A partir d'un fichier contenant un nombre fini d'éléments, construire un tableau trié, et gérer la sécurité des relations avec le système d'exploitation.

Principe : analyse des éléments par paire.
Eléments : insertion progressive des éléments dans la structure.
Type d'opérations : balayage simple et décalage simple.
Tri effectif : insertion directe de l'élément à sa place.
Type de comparaison : quelconque.
Structure : une structure unique.
Composition de la structure : deux ensembles, un trié et un vide.
Gestion de la structure : pointeur de balayage et pointeur de décalage.
Ordre du tri : croissant.

Exemple :

Nous allons insérer les éléments suivants, dans l'ordre de présentation :

```
{8,3,4,9,6,5}
suite[1..6] <-- { , , , , , }

pointeur sur :   1er élément
ajout :          8
suite[1..6] <-- {8, , , , , }
ajout :          3
suite[1..6] <-- {8, , , , , }
pointeur sur :   1er élément
paire analysée : 8,3            8 > 3 VRAI
décalage :      {8, , , , , }
fin décalage :  { ,8, , , , }
pointeur sur :   1er élément
insertion :      3
suite[1..6] <-- {3,8, , , , }

ajout :          4
suite[1..6] <-- {3,8, , , , }
pointeur sur :   1er élément
paire analysée : 3,4            3 > 4 FAUX
suite[1..6] <-- {3,8, , , , }
pointeur sur :   2ème élément
paire analysée : 8,4            8 > 4 VRAI
décalage :      {3,8, , , , }
fin décalage :  {3, ,8, , , }
pointeur sur :   1er élément
paire analysée : 3,4            3 > 4 FAUX
pointeur sur :   2ème élément
insertion :      4
suite[1..6] <-- {3,4,8, , , }

ajout :          9
suite[1..6] <-- {3,4,8, , , }
pointeur sur :   1er élément
paire analysée : 3,9            3 > 9 FAUX
suite[1..6] <-- {3,4,8, , , }
pointeur sur :   2ème élément
paire analysée : 4,9            4 > 9 FAUX
suite[1..6] <-- {3,4,8, , , }
pointeur sur :   3ème élément
paire analysée : 8,9            8 > 9 FAUX
pointeur sur :   4ème élément
insertion :      9
suite[1..6] <-- {3,4,8,9, , }

ajout :          6
suite[1..6] <-- {3,4,8,9, , }
pointeur sur :   1er élément
```

```
paire analysée : 3,6            3 > 6 FAUX
suite[1..6] <-- {3,4,8,9, , }
pointeur sur :   2ème élément
paire analysée : 4,6            4 > 6 FAUX
suite[1..6] <-- {3,4,8,9, , }
pointeur sur :   3ème élément
paire analysée : 8,6            8 > 6 VRAI
décalage :      {3,4,8,9, , }
décalage :      {3,4,8, ,9, }
fin décalage :  {3,4, ,8,9, }
pointeur sur :   3ème élément
insertion :      6
suite[1..6] <-- {3,4,6,8,9, }

ajout :          5
suite[1..6] <-- {3,4,6,8,9, }
pointeur sur :   1er élément
paire analysée : 3,5            3 > 5 FAUX
suite[1..6] <-- {3,4,6,8,9, }
pointeur sur :   2ème élément
paire analysée : 4,5            4 > 5 FAUX
suite[1..6] <-- {3,4,6,8,9, }
pointeur sur :   3ème élément
paire analysée : 6,5            6 > 5 VRAI
décalage :      {3,4,6,8,9, }
décalage :      {3,4,6,8, ,9}
décalage :      {3,4,6, ,8,9}
fin décalage :  {3,4, ,6,8,9}
pointeur sur :   3ème élément
insertion :      5
suite[1..6] <-- {3,4,5,6,8,9}
```

Tri par fusion

Principe : analyse des éléments par paire.
Eléments : insertion progressive des éléments des deux structures dans une troisième.
Type d'opérations : balayage simple.
Tri effectif : insertion directe de l'élément à sa place.
Type de comparaison : quelconque.
Composition des structures : deux structures triées, une structure vide.
Structure : trois structures distinctes.
Gestion de la structure : pointeur de balayage pour chacune des structures.
Ordre du tri : croissant.

Exemple :
Nous allons insérer les éléments suivants, dans l'ordre de présentation :

```
{3,4,8,9}
{1,2,5,6,7}
suite[1..9] <-- { , , , , , , , , }

                {3,4,8,9}
pointeur sur :   1er élément
                {1,2,5,6,7}
pointeur sur :   1er élément
suite[1..9] <-- {_, , , , , , , , }
pointeur sur :   1er élément

paire analysée : 3,1               3 > 1 VRAI

ajout :          1
suite[1..9] <-- {1, , , , , , , , }
                {3,4,8,9}
pointeur sur :   1er élément
                {1,2,5,6,7}
pointeur sur :   2ème élément
suite[1..9] <-- {1,_, , , , , , , }
pointeur sur :   2ème élément

paire analysée : 3,2               3 > 2 VRAI

ajout :          2
suite[1..9] <-- {1,2, , , , , , , }
                {3,4,8,9}
pointeur sur :   1er élément
                {1,2,5,6,7}
pointeur sur :   3ème élément
suite[1..9] <-- {1,2,_, , , , , , }
pointeur sur :   3ème élément

paire analysée : 3,5               3 > 5 FAUX

ajout :          3
suite[1..9] <-- {1,2,3, , , , , , }
                {3,4,8,9}
pointeur sur :   2ème élément
                {1,2,5,6,7}
pointeur sur :   3ème élément
suite[1..9] <-- {1,2,3,_, , , , , }
pointeur sur :   4ème élément

paire analysée : 4,5               4 > 5 FAUX

ajout :          4
suite[1..9] <-- {1,2,3,4, , , , , }
```

```
                    {3,4,8,9}
pointeur sur :       3ème élément
                    {1,2,5,6,7}
pointeur sur :       3ème élément
suite[1..9] <-- {1,2,3,4,_, , , , }
pointeur sur :       5ème élément

paire analysée : 8,5                    8 > 5 VRAI

ajout :              5
suite[1..9] <-- {1,2,3,4,5, , , , }
                    {3,4,8,9}
pointeur sur :       3ème élément
                    {1,2,5,6,7}
pointeur sur :       4ème élément
suite[1..9] <-- {1,2,3,4,5,_, , , }
pointeur sur :       6ème élément

paire analysée : 8,6                    8 > 6 VRAI

ajout :              6
suite[1..9] <-- {1,2,3,4,5,6, , , }
                    {3,4,8,9}
pointeur sur :       3ème élément
                    {1,2,5,6,7}
pointeur sur :       5ème élément
suite[1..9] <-- {1,2,3,4,5,6,_, , }
pointeur sur :       7ème élément

paire analysée : 8,7                    8 > 7 VRAI

ajout :              7
suite[1..9] <-- {1,2,3,4,5,6,7, , }
                    {3,4,8,9}
pointeur sur :       3ème élément
                    {1,2,5,6,7}
pointeur sur :       fin
suite[1..9] <-- {1,2,3,4,5,6,7,_, }
pointeur sur :       8ème élément

                    {3,4,8,9}
pointeur sur :       3ème élément

ajout :              8

suite[1..9] <-- {1,2,3,4,5,6,7,8,_}
pointeur sur :       9ème élément

                    {3,4,8,9}
```

```
pointeur sur :   4ème élément

ajout :          9

suite[1..9] <-- {1,2,3,4,5,6,7,8,9}
pointeur sur :   fin
```

Tri à bulles

Principe : analyse des éléments par paire.
Eléments : déjà présents dans la structure.
Type d'opérations : balayage simple et balayage à effet de bord.
Tri effectif : permutation des éléments de la paire.
Type de comparaison : quelconque.
Structure : une structure unique.
Composition de la structure : deux ensembles, un trié et un non trié.
Gestion de la structure : pointeur de balayage et pointeur d'ensemble non trié.
Ordre du tri : croissant.

Exemple :

suite[1..6] <-- {8,3,4,9,6,5}
pointeur sur : 8
paire analysée : 8,3 8 > 3 VRAI
permutation : 3,8
suite[1..6] <-- {3,8,4,9,6,5}
pointeur sur : 8
paire analysée : 8,4 8 > 4 VRAI
permutation : 4,8
suite[1..6] <-- {3,4,8,9,6,5}
pointeur sur : 8
paire analysée : 8,9 8 > 9 FAUX
suite[1..6] <-- {3,4,8,9,6,5}
pointeur sur : 9
paire analysée : 9,6 9 > 6 VRAI
permutation : 6,9
suite[1..6] <-- {3,4,8,6,9,5} EFFET DE BORD !
pointeur sur : 8
paire analysée : 8,6 8 > 6 VRAI
permutation : 6,8
suite[1..6] <-- {3,4,6,8,9,5} EFFET DE BORD : TERMINE
pointeur sur : 8
paire analysée : 8,9 8 > 9 FAUX
suite[1..6] <-- {3,4,6,8,9,5}
pointeur sur : 9

paire analysée : 9,5 9 > 5 VRAI
permutation : 5,9
suite[1..6] <-- {3,4,6,8,5,9} EFFET DE BORD !
pointeur sur : 8
paire analysée : 8,5 8 > 5 VRAI
permutation : 5,8
suite[1..6] <-- {3,4,6,5,8,9} EFFET DE BORD !
pointeur sur : 6
paire analysée : 6,5 6 > 5 VRAI
permutation : 5,6
suite[1..6] <-- {3,4,5,6,8,9} EFFET DE BORD : TERMINE
pointeur sur : 6
paire analysée : 6,8 6 > 8 FAUX
suite[1..6] <-- {3,4,5,6,8,9}
pointeur sur : 8
paire analysée : 8,9 8 > 9 FAUX
pointeur sur : 9 FIN

Complexité Algorithmique: Introduction

Présentation

L'évaluation des algorithmes est une phase importante de la vie d'un logiciel (ou partie de logiciel).

Ce critère d'efficacité doit faire l'objet d'une analyse approfondie dés le début d'un projet de conception et de développement.

C'est, en général, une analyse comparative qui est engagée. La question posée est : « Parmi ces deux algorithmes, lequel est le plus performant ? »

En termes de performance, toute analyse comparative devra être traitée en donnant la priorité au critère de performance basé :
Soit sur le temps d’exécution de l’algorithme.

Soit sur la capacité mémoire nécessaire.

Mais les décisions d'un analyste-développeur ne se limitent pas à une simple comparaison ...

Comment analyser la performance d'un algorithme ?
Le moyen de mesure le plus courant est la durée d'exécution.

Mais, à ce stade, c'est un processus qui est testé. Cela signifie que d'autres éléments importants sont pris en compte. Interviennent dans ce type d'analyse :
Côté logiciel :
Le langage de programmation
Le compilateur et les options de compilation
Le système d'exploitation
Côté matériel :
Le type de processeur et sa vitesse de traitement
La capacité mémoire et les temps d'accès

L'association matériel/logiciel (problèmes d'incohérence)

Tous ces éléments ont un impact important sur les mesures relevées.

Première approche

Afin d'effectuer une analyse efficace, il faut aborder le problème en amont, et évaluer l'algorithme "sur le papier" !

La question qui se pose semble, au premier abord, surprenante :
Comment mesurer la durée d'exécution d 'un programme à l'état d'algorithme ?

Deux options se présentent :
La première consiste à relever, instruction par instruction, les durées unitaires de chaque opération traitée par la machine et au niveau le plus bas (instructions machines).

> Par exemple, une simple opération d'affectation :
>
> peut avoir une durée en nombre de tops d'horloge de 2 ; une opération 4, … (valeurs données juste à titre d'exemple)

Le cumul des tops d'horloge donne une approximation de la durée d'exécution.

Cette première option pose deux problèmes dans la réalité :
D'une part, cette méthode fait encore intervenir des spécificités relatives à l'environnement (processeur employé).

D'autre part, une évaluation aussi précise est un travail complémentaire conséquent.

Conclusion : cette technique n’est réservée qu’aux langages de programmation de très bas niveau.

La seconde option consiste à s'extraire entièrement du contexte de développement.
Cette méthode est basée sur la prise en compte d'éléments essentiels au bon déroulement de l'algorithme, ce qui le caractérise. Sont pris en compte :
Les traitements : soit, le nombre effectif de calculs.
et, par conséquent, la cause des calculs répétitifs: les boucles.

Approche maximaliste

Définition :
La complexité d'un algorithme informe du nombre d'instructions caractéristiques mis en oeuvre avec un ensemble de données de taille définie.

Précision :
Cette définition ne peut se suffire à elle-même. L'approche complémentaire associée à la complexité algorithmique est une approche maximaliste (ou "au pire des cas" ou pessimiste).

L'approche par une complexité moyenne peut aussi être envisagée. Cependant, son évaluation reste délicate et le résultat des comparaisons n’est pas toujours représentatif.

Représentation :
La complexité d'un algorithme est une fonction.

Objectif :
La recherche d'une complexité algorithmique la plus faible possible.

Temps constant

Temps d'exécution constant :
Dans l'exemple, le temps d'exécution est constant.

Propriété :
Toutes les instructions, de cet ordre, exécutées une (ou plusieurs) fois sont prises en compte avec un temps d'exécution constant.
Par extension :
Si toutes les instructions d'un algorithme sont exécutées en temps constant, l'algorithme est lui-même traité en temps constant.
La notation est : 1

Temps linéaire

Temps d'exécution linéaire : (hors affichage du résultat)

Dans l'exemple :
Quels sont les instructions dont le temps d'exécution est constant ?

Quels sont celles dont le temps d'exécution n'est pas constant ? (ou combien d'instructions non constantes existe-t-il dans le cas où le "mot" de longueur N serait saisi par l'utilisateur ?)

Temps d'exécution linéaire : (hors affichage du résultat)

Dans l'exemple précédent, le temps d'exécution est linéaire.
Propriété :
Toute instruction en charge du traitement d'un ensemble complet en entrée (ou en sortie), constitué de N éléments, possède un temps d'exécution linéaire égal à N.
Particularité :
Lorsque le traitement effectif est celui d'un ensemble fini de N éléments, un temps d'exécution de N correspond à l'optimal (pas d'amélioration possible).
La notation est : N

Temps proportionnel

Temps d'exécution proportionnel : (hors affichage du résultat)

Dans cet exemple, le temps d'exécution est proportionnel à la taille de l'entrée.
Propriété :
Toute instruction en charge du traitement d'un ensemble en entrée (ou en sortie), constitué de N éléments, et dont le traitement dépend directement des éléments en entrée possède un temps d'exécution proportionnel à N.
Particularité :
Lorsque le traitement effectif d'un ensemble fini de N éléments dépend de ses entrées :
Soit, l'exécution est stoppée en cours : le traitement itératif est terminé, (ex: caractère trouvé). L'analyse est basée sur une complexité moyenne : N / 2.

Soit, l'exécution va jusqu'à son terme, (ex: caractère non trouvé). L'analyse est basée sur une complexité au pire des cas : N.

Temps quadratique

Temps d'exécution quadratique : (hors affichage du résultat)

Dans l'exemple ci-dessous, une table des occurrences est montée de telle manière qu'à chaque indice de cette table corresponde l'occurrence de la lettre du mot de même indice.

Dans cet exemple, le temps d'exécution est quadratique.
Propriété :
Toute instruction en charge du traitement d'un ensemble complet en entrée (ou en sortie), constitué de N éléments, et elle-même incluse dans une instruction de même type et de même domaine, possède un temps d'exécution global égal à N^2.
Particularité :
Lorsque chacun des traitements effectifs est celui d'un ensemble fini de N éléments, un temps d'exécution de N correspond à l'optimal (pas d'amélioration possible).

En revanche, le traitement global des instructions imbriquées en N peuvent être à l'origine d'optimisations.
La notation est : N^2

Amélioration du temps quadratique

Temps d'exécution réduit : (hors affichage du résultat)

L'exemple ci-dessous est une version améliorée du précédent dont le temps d'exécution était quadratique.

Dans cet exemple, le temps d'exécution est amélioré par rapport à l'exemple précédent de temps quadratique.
Propriété :
Toute instruction en charge du traitement d'un ensemble réduit d'un élément en entrée (ou en sortie) à chaque itération, constitué à la base de N éléments, et elle-même incluse dans une instruction exploitant le domaine complet (soit N éléments), possède un temps d'exécution global égal à N fois la partie entière de N/2.
Particularité :
Lorsque chacun des traitements effectifs est celui d'un ensemble fini de N éléments, un temps d'exécution est de N.

Le traitement interne à cette partie est diminué d'un élément à chaque itération.

Dans cette configuration, l'algorithme correspond à l'optimal (pas d'amélioration possible).

La notation est : N.[N/2]

Temps logarithmique

Temps d'exécution logarithmique : (hors affichage du résultat)

Dans cet exemple, le temps d'exécution est logarithmique.
Propriété :

Toute instruction en charge du traitement d'un ensemble complet en entrée (ou en sortie), constitué de N éléments, qui partage cet ensemble en deux sous-ensembles de même cardinalité (à un élément près), à chaque itération, possède un temps d'exécution global égal à la valeur entière du logarithme de N.

Particularité :
Lorsque le nombre d'éléments contenus dans l'ensemble est impair, le temps d'exécution global est augmenté de 1.

Lorsque l'élément recherché n'est pas contenu dans l'ensemble, le temps d'exécution global est égal à la valeur entière du logarithme de N.

Lorsque l'élément recherché est retrouvé, le temps d'exécution global peut être optimisé (amélioration possible).

La notation est : log(N)

Ordre de grandeur

Classement et représentation graphique des principales fonctions

Notation en O() « grand O »

L'analyse d'un algorithme, avec une précision « à l'instruction près », est difficilement envisageable.

Seuls les termes les plus grands intéressent l'analyste en charge des comparaisons !

L'outil mathématique employé pour réaliser cela est une fonction.

Sa notation est : O() (dire: « grand O »)

Intérêt d'une notation en O():
Comparer des algorithmes sur la base de la borne maximale de leur temps d'exécution.
La simplification des écritures obtenues à l'aide de la fonction O(), risque d'introduire des erreurs d'approximation, puisque les termes les plus petits et des morceaux de programmes peuvent être ignorés.
Ces éléments seront bornés pour être conservés.

Temps constant

Temps d'exécution constant :

Dans l'exemple,

À l'aide d'une écriture en O(), la notation est : O(1)

Temps linéaire

Temps d'exécution linéaire :

Dans l'exemple,

À l'aide d'une écriture en O(), la notation est : O(N)

Temps proportionnel

Temps d'exécution proportionnel :

À l'aide d'une écriture en O(), la notation est : O(N)

Temps quadratique

Temps d'exécution quadratique :

À l'aide d'une écriture en O(), la notation est : O(N^2)

Temps logarithmique

Temps d'exécution logarithmique :

À l'aide d'une écriture en O(), la notation est : O(log(N))

Première approche

L'analyse de la complexité d'un algorithme permet de distinguer deux grandes classes d'algorithmes.

Souvent, un informaticien parlera de « bon algorithme », et par opposition, de « mauvais algorithme ». Quel critère peut scinder en deux parties distinctes l'évaluation de la complexité algorithmique ?

Au terme de l'analyse d'un algorithme, l'évaluation en O() de celui-ci peut être :
Soit, de l'ordre d'un polynôme :
L'algorithme est dit polynomial.
Soit, exponentiel :
L'algorithme est dit exponentiel.
Les différentes formes d'algorithmes dits « polynômiaux » :
log(N)
N.log(N)

N2
…
Les différentes formes d’algorithmes dits « exponentiels » :
2N
eN
Nlog(N)   (fonction sous-exponentielle)
NN
N!   (fonction factorielle)
…

Algorithme polynomial

Un « bon algorithme » est un algorithme polynomial.

Une fonction exponentielle aura une efficacité bien inférieure à une fonction polynômiale lorsque N est très grand. Aussi, il sera rare de travailler sur des complexités polynômiales dont l’ordre dépasse 4 ou 5 !

Quelque soit le problème traité, il faudra, au regard de l’analyse comparative de la complexité de deux algorithmiques, prendre en compte la dimension de N dans le choix de l’algorithme à implémenter ! En pratique, l’efficacité de l’algorithme dépend étroitement de cet ordre de grandeur.

Problèmes difficiles

Un « bon algorithme » pour la résolution des problèmes de décision (appelés aussi problèmes d’existence), fait parti de la « classe P ».

Un « bon algorithme » pour la résolution des problèmes d’optimisation (ou problèmes d’optimisation combinatoire), fait parti de la famille des « problèmes faciles ».

Un « mauvais algorithme » fait parti de la famille des « problèmes difficiles ».

CHAPITRE XI

Complexité Algorithmique: Introduction

Complexité d'un algorithme

Dans cette partie, il est demandé d’écrire des fonctions dont les algorithmes effectuent des calculs ou des tests élémentaires à l’aide de matrices.

Toutes les productions devront être optimisées pour répondre au critère d'efficacité relatif à la complexité algorithmique.
Le cours de mathématiques, « Algèbre Linéaire », présente de façon plus détaillée, les techniques et les conditions pour réaliser ces opérations.

Addition de deux matrices

Principe :
L'addition de deux matrices ne peut s'effectuer, que lorsque celles-ci sont de même format.
L'addition effective est simplement une addition terme par terme, le résultat produisant une matrice de format identique aux deux autres.

Réaliser une fonction qui produira l'addition de deux matrices implémentées à l'aide de tableaux.

Multiplication d'une matrice par un réel

Principe :
La multiplication d'une matrice avec un réel est réalisée grâce à la multiplication du réel et de chaque terme de la matrice. Le résultat produit une matrice de format identique à celle d'origine.

Réaliser une fonction qui produira la multiplication d'un réel à une matrice implémentée à l'aide d'un tableau.

Multiplication de deux matrices

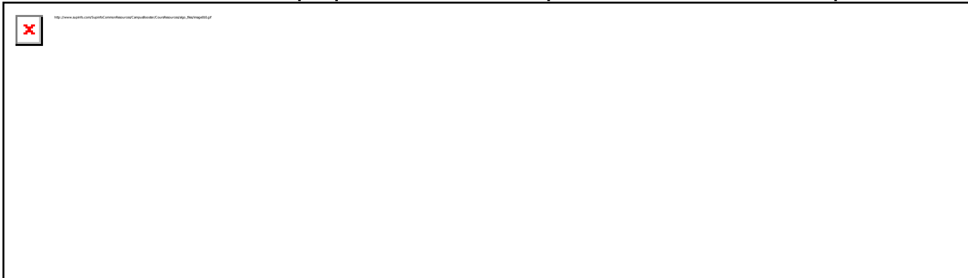
Principe :
La multiplication est une opération plus complexe que les précédentes. Il est impératif de se rapprocher du cours d'algèbre linéaire afin de maîtriser pleinement ce concept de multiplication matricielle.

Réaliser une fonction qui produira la multiplication de deux matrices implémentées à l'aide de tableaux.

Transposé d'une matrice

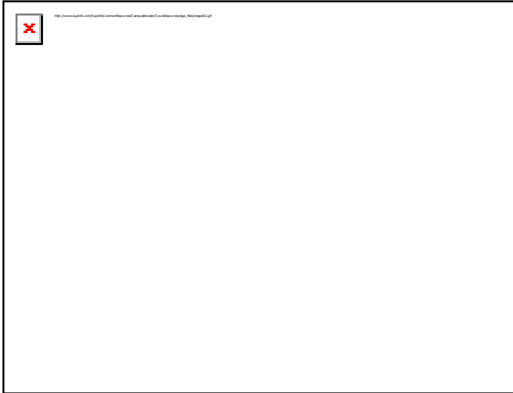
Principe :
La transposition d'une matrice quelconque, modifie la structure de la matrice en permutant les lignes (le nombre de lignes) avec les colonnes (le nombre des colonnes) ainsi que tous les termes qui la composent par rapport à sa diagonale.
La diagonale est toujours « tracée virtuellement » entre le premier terme situé en haut à gauche et le dernier terme en bas à droite.

Réaliser une fonction qui produira le transposé d'une matrice implémentée à l'aide d'un tableau.



Matrice diagonale

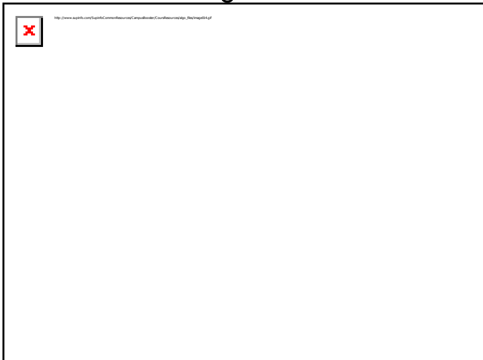
Principe :
Une matrice est dite « diagonale » lorsque tous les termes autres que sur sa diagonale sont nuls.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice diagonale. La matrice doit être implémentée à l'aide d'un tableau.

Matrice scalaire

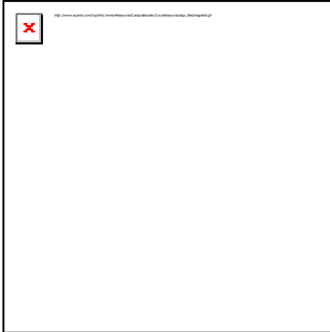
Principe :
Une matrice est dite « scalaire » lorsque tous les termes autres que sur sa diagonale sont nuls et que tous les termes de la diagonale sont de même valeur.
Une autre définition peut être donnée de la sorte : une matrice scalaire est une matrice diagonale dont tous les termes de sa diagonale sont de même valeur.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice scalaire. La matrice doit être implémentée à l'aide d'un tableau.

Matrice unité ou matrice identité

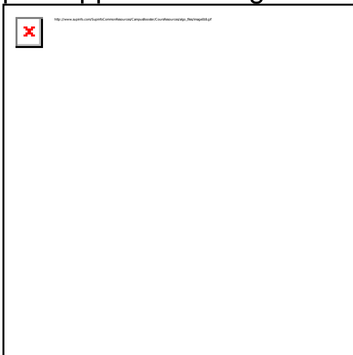
Principe :
Une matrice est dite « unité » ou « identité » lorsque tous les termes autres que sur sa diagonale sont nuls et que tous les termes de la diagonale sont de même valeur égale à 1.
Une autre définition peut être donnée de la sorte : une matrice « unité » ou « identité » est une matrice diagonale dont tous les termes de sa diagonale sont de même valeur égale à 1.
Une dernière définition : une matrice « unité » ou « identité » est une matrice scalaire dont tous les termes de sa diagonale sont égales à 1.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice unité ou identité. La matrice doit être implémentée à l'aide d'un tableau.

Matrice carré symétrique

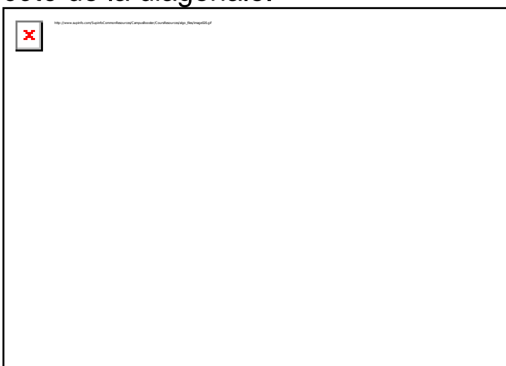
Principe :
Une matrice est dite « symétrique » lorsque tous les termes autres que sur sa diagonale présentent une symétrie par rapport à la diagonale.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice symétrique. La matrice doit être implémentée à l'aide d'un tableau.

Matrice carré antisymétrique

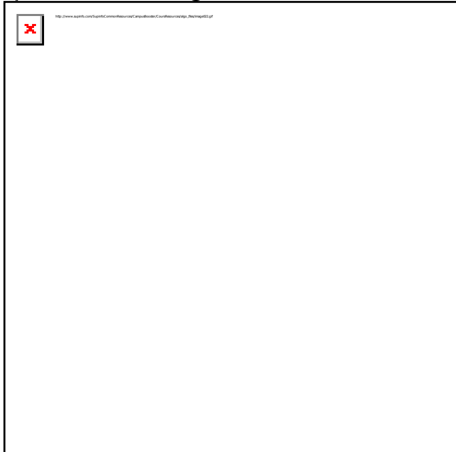
Principe :
Une matrice est dite « antisymétrique » lorsque tous les termes autres que sur sa diagonale présentent une antisymétrie par rapport à la diagonale, c'est-à-dire une inversion du signe pour chacun des termes, de chaque côté de la diagonale.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice antisymétrique. La matrice doit être implémentée à l'aide d'un tableau.

Matrice triangulaire supérieure

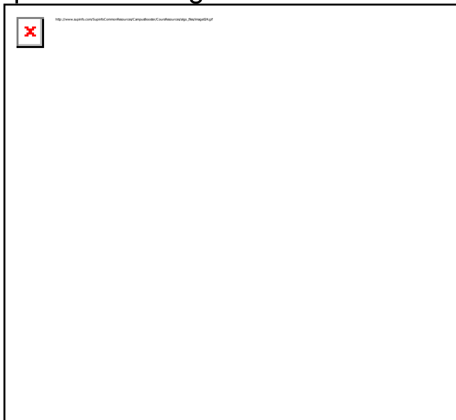
Principe :
Une matrice est dite « triangulaire supérieure » lorsque tous les termes placés en partie inférieure gauche autres que sur sa diagonale sont nuls.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice triangulaire supérieure. La matrice doit être implémentée à l'aide d'un tableau.

Matrice triangulaire inférieure

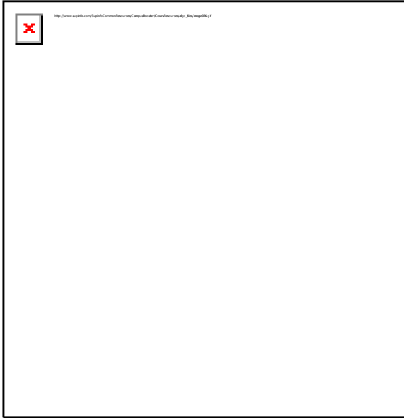
Principe :
Une matrice est dite « triangulaire inférieure » lorsque tous les termes placés en partie supérieure droite autres que sur sa diagonale sont nuls.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice triangulaire inférieure. La matrice doit être implémentée à l'aide d'un tableau.

Matrice triangulaire strictement supérieure

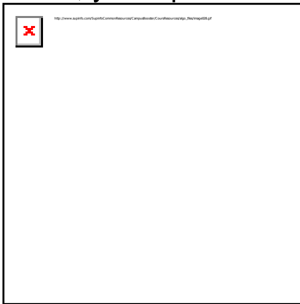
Principe :
Une matrice est dite « triangulaire strictement supérieure » lorsque tous les termes placés en partie inférieure gauche, y compris sur sa diagonale, sont nuls.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice triangulaire strictement supérieure. La matrice doit être implémentée à l'aide d'un tableau.

Matrice triangulaire strictement inférieure

Principe :
Une matrice est dite « triangulaire strictement inférieure » lorsque tous les termes placés en partie supérieure droite, y compris sur sa diagonale, sont nuls.



Réaliser une fonction qui retournera une information de type booléen indiquant que la matrice est bien une matrice triangulaire strictement inférieure. La matrice doit être implémentée à l'aide d'un tableau.

Résolution d'un problème en informatique

Prise en compte d'un problème

Les modèles, les outils d'analyse et de développement

Avant tout, il est nécessaire de préciser que les développements informatiques, comme dans d'autres domaines, sont parfois très complexes.

Pour assurer l'aboutissement de projets conséquents, les analystes et chefs de projet, ciblent des modèles adaptés au contexte étudié afin de faciliter la phase d'études.

Il existe des modèles « génériques » (UML), et d'autres orientés : analyse fonctionnelle (SADT), analyse dynamique (Réseau de Pétri), analyse sémantique, …

Les modélisations permettent, d'une part, de contenir une certaine standardisation du traitement retenu, et d'autre part, de garantir un grand nombre de critères de qualité dont le modèle même se charge.

Chaque modèle dispose de ses propres contraintes de développement. Aussi, ce chapitre n'aborde ni les modèles disponibles, ni leur cadre d'emploi.

Formalisation

Noter et formaliser le plus grand nombre d'éléments, même ceux qui ne semblent pas importants, voire sans intérêt, du moins au début de la démarche.

Noter ce que doit faire, de manière globale, le programme.

Noter les informations demandées en entrée, ainsi que celles en sortie.

Identifier des éléments informationnelles intermédiaires non nécessaires à l'aboutissement du problème, mais éventuellement utiles afin de contrôler l'évolution ou la résolution du problème (très pratique pour valider le programme par étape).

Prendre du recul par rapport au problème donné, par rapport aux définitions (parfois trop étroites) inscrites dans le cahier des charges. Faire cette démarche régulièrement.

Décomposer systématiquement tout ce qui peut l'être (un problème complexe peut très souvent être décomposé en une suite de petits problèmes plus simples, dont la validité peut être atteinte rapidement).

Dans un premier temps, écrire en langage naturel les diverses tentatives de résolution, lorsque si le transfert direct en langage algorithmique est trop compliqué. La forme la plus simple est parfois appelée "Algorithme de principe". Très souvent, son écriture ne prend en compte que la structure algorithmique, en mettant de côté les structures de données complexes à cause des contraintes associées à leur gestion.

Prendre un exemple valide, adapté au contexte, que l'on nomme "échantillon". Il ne doit pas être : ni trop petit pour ne pas rentrer dans un cas simple, ni particulier pour ne pas interpréter que partiellement le programme, ni trop gros pour ne pas perdre de temps en résolution. Celui-ci doit être élaboré pour prendre ne compte un domaine dans son intégralité. (Le calcul de la valeur absolue d'un entier relatif peut être testé à l'aide du domaine : {-7;0;+13} plutôt que {-7}).

Faire tourner "à la main" (principe de traçage) le programme développé (ou les procédures ou fonctions). Décomposer étape par étape l'algorithme et monter le tableau récapitulant l'évolution des données.

Parallèlement à cette démarche, tracer l'arbre des appels effectifs aux modules et contrôler les transferts et les modifications des données de l'ensemble du programme.

Quelles structures de données ?

Recherche des structures de données adaptées

Noter toutes les informations d'entrée et de sortie, ainsi que celles identifiées comme nécessaires pour faire aboutir la résolution du problème.

Identifier les informations qui peuvent évoluer durant la résolution du problème.

Mettre en adéquation les structures de données connues en algorithmique et trouver celles les mieux adaptées aux informations identifiées. Penser que des évolutions notées ou non au cahier des charges peuvent remettre en question une structuration algorithmique à cause d'un choix de données inadapté.

Sélectionner les informations qui n'évolueront jamais (CONSTANTE) de celles qui vont nécessiter une affectation (VARIABLE).

Quelles structures algorithmiques ?

Recherche des structures algorithmiques adaptées

Mettre en évidence la méthode de résolution du problème posé (formule d'ordre mathématique, logique, étapes successives, …).

Mettre en adéquation les structures algorithmiques avec la méthode de résolution déterminée.

Mettre en adéquation les structures algorithmiques avec les structures de données choisies.

Choix des instructions algorithmiques

Opérations séquentielles :
Suite séquentielle d'instructions ou d'appels à des fonctions ou des procédures
Opérations répétitives :
Choix (éventuel) de la récursivité

"bornes" ou "pas" variable :
Préférer TANTQUE

ou REPETER lorsque le corps de la répétition doit être exécuté au moins une fois
"bornes" et "pas" fixes :
Préférer POUR
Opérations de filtrage ou sélection :
Sélection unique :
Préférer SI..ALORS
Sélection unique avec alternative :
Préférer SI..ALORS..SINON
Sélection multiple :
Préférer SI..ALORS..SINON
ou CAS..PARMI (s'adapter au langage de programmation définitif)

Quel découpage modulaire ?

Choix du découpage en modules

Cette partie n'est pas simple.
Elle ne peut être abordée que lorsque l'on a une vision d'ensemble quasi complète d'un projet informatique, ou bien lorsque l'on a une bonne expérience de la programmation.
L'objectif à atteindre rapidement consiste à s'occuper de la modularité le plus tôt possible, en phase de conception.

Une méthode de découpage en modules :
Toujours commencer par écrire le programme sans chercher à optimiser le code d'un point de vue algorithmique ou/et capacité mémoire, (et surtout pas en terme de réduction de l'écriture syntaxique).

Chercher les parties de programme identiques ou similaires et adapter l'écriture pour obtenir une structure générique (c'est à dire adaptée à une transformation en module).

Emploi éventuel de la récursivité

Simplification par le biais de la récursivité

Lors de la phase de découpage, certaines parties du programme peuvent être identifiées comme des structures répétitives autonomes.
Dans ce cas, la récursivité peut être employée. Cependant, elle n'est pas obligatoire. Aussi, il est nécessaire de prendre en compte les contraintes du cahier des charges, afin de contrôler que les ressources sont suffisantes par rapport aux inconvénients générés par l'emploi de la récursivité (taille de la pile).

Dans d'autres situations, c'est la structure de données qui implique l'emploi de la récursivité (arborescence).
Il faut rappeler que la récursivité est un outil de simplification du traitement d'un problème.

Dans tous les cas, lors de la phase d'analyse, si le contexte de développement avec récursivité semble complexe ou moins bien maîtrisé qu'une structure algorithmique répétitive classique, la récursivité doit être abandonnée.

Validation du programme

Validation, qualité essentielle d'un programme

Valider un programme, c'est le contrôler, afin de donner la garantie qu'il réalise exactement ce pour quoi il a été conçu.
Il faut distinguer un programme arrivé à une phase "écriture algorithmique" de celui qui est prêt à être exécuté, que l'on nomme "processus‟.
Contrôler un algorithme, c'est le faire tourner instruction après instruction "à la main", (programme principal, fonctions et procédures) et tracer son fonctionnement de façon exhaustive sur des échantillons valides.

Contrôler un processus, c'est le mettre dans une configuration similaire à un contexte de fonctionnement normal. Une erreur à ce niveau, et tout est à refaire !

Les parties "analyse" et écriture algorithmique sont les phases qui garantissent le mieux la validité d'un programme.

La meilleure méthode consiste à identifier des contextes de fonctionnement de programme par "classe" (ou ensemble de domaines de test), dans le but de contrôler chacune de ces classes.

Plus le problème est complexe et conséquent, plus le nombre de classes peut être important.

Le découpage en modules permet de réduire efficacement le nombre de classes grâce à la validation de chaque module indépendamment.

En phase de test d'un processus (ou test par simulation), il est généralement fait appel à des générateurs de données aléatoires bridés dans les domaines identifiés par les bornes du domaine, les lois applicables, …

Différents types de problèmes

Première approche générale des problèmes difficiles :

En informatique, il existe deux grandes familles de problèmes difficiles à résoudre :
Des problèmes de décisions.

Des problèmes d'optimisation (soit maximisation: recherche d'une meilleure rentabilité ; soit minimisation: réduction des durées de fabrication, …).
Il existe aussi une combinaison des deux types de problèmes. En général, l'un est imbriqué dans l'autre.

Différents types de problèmes

Solution(s) et réponse(s) attendue(s) :

Dans le cas des problèmes de décisions, la réponse attendue est simple :
Réponse : Oui ou Non (expression binaire).
Dans le cas des problèmes d'optimisation, la réponse attendue semble simple, sur un plan purement théorique :
Réponse : une valeur, la meilleure en fonction de l'objectif à atteindre (dans le cas d'un problème d'optimisation mono-objectif).

Résoudre un problème : en théorie

Solution(s) d'un problème :

En informatique, comme en mathématiques, les problèmes peuvent être regroupés en plusieurs catégories :
Une pour laquelle les problèmes ne possèdent pas de solution (ensemble vide).

Une pour laquelle les problèmes possèdent une seule solution.

Une pour laquelle les problèmes possèdent plusieurs solutions (et dont l'objectif est atteint pour chacune d'elles)

Résoudre un problème : en informatique

En informatique, la résolution des problèmes situés dans les deux premières catégories :
Catégorie pour laquelle les problèmes ne possèdent pas de solution (ensemble vide).

Catégorie pour laquelle les problèmes possèdent une seule solution.
est accessible à l'aide d'algorithmes.

En revanche, une classe pour laquelle les problèmes possèdent plusieurs solutions ne peut que très difficilement être traitée à l'aide d'un algorithmique.

Modélisation et implémentation

En informatique :

La recherche d'un modèle approprié au traitement attendu (voire l'adaptation d'un modèle proche) engendre des difficultés de résolution complémentaires.

La structure de données et le type de données employés peuvent avoir pour incidence l'introduction d'une erreur d'approximation sur le résultat.

Le choix de l'algorithme (lorsque l'on opte pour cette solution) a un impact direct sur la solution trouvée.

La complexité algorithmique oriente aussi ce choix.

Par ailleurs, d'autres problèmes n'ont pas de solution juste, à l'aide d'algorithmes, ou encore, leur temps de résolution est démesuré.

Conclusion

La construction de programmes basée sur une écriture algorithme n'est pas toujours la meilleure solution.

L'algorithmique possède ses propres limites, et la recherche de toutes les solutions pour un problème donné peut devenir très complexes, voire impossible.

D’autres façons d’aborder un problème sont nécessaires, d’autres modèles doivent être employés.

Certains langages de programmation sont basés sur une programmation logique dont les concepts sont proches de ceux abordés en algorithmiques, mais dont la structuration générale est radicalement différente.

On ne parle plus d’algorithmes et de compilation, mais de règles de productions et d’interprétation par un moteur d’inférence.

Présentation

Qualité d'un programme

La qualité d'un programme est obtenue grâce à une application permanente et équilibrée des divers critères de qualité présentés ci-après.

Pour un projet de conception de logiciel, leur prise en compte commence dès la réalisation du cahier des charges (phase de formalisation du besoin), … pour se poursuivre durant toute l'écriture algorithmique ... et les différentes phases qui suivent !

Les principales qualités d'un programme
Validité
Convivialité
Transportabilité
Maintenance aisée
Modularité
Sécurité
Sûreté
Immunité
Lisibilité
Optimalité mono ou multi-objectif
Simplicité

Validité

Critère de qualité donné à un programme dont l’exécution correspond exactement à l’objectif prédéfini (en théorie : il ne fait ni plus, ni moins que ce qui a été stipulé dans le cahier des charges).

Démontrer qu’un algorithme est valide, c’est tout d’abord apporter la preuve que celui-ci se termine (condition de terminaison : le nombre d’opérations exécutées est fini).

Ce critère de qualité est une condition obligatoire pour tout algorithme.

Exemple :
Un programme qui calcule l'aire d'un disque dans une précision donnée, doit effectivement fournir le résultat attendu dans la définition spécifiée.

Convivialité

Critère de qualité donné à un programme dont l'exploitation, c'est-à-dire toute la phase d'utilisation (installation, configuration, emploi dans toutes les configurations possibles, suppression), doit être aisée et adaptée aux connaissances de l'utilisateur.

Exemple :
L'interface « homme-machine » doit être bien pensée, ergonomique, afin d'éviter des erreurs graves et irréversibles (comme l'écrasement d'un fichier de version antérieure sans dispositif d'alerte). Enrichir le code d'annonces préventives claires (comme des informations relatives aux données à saisir : « Tapez 'N' ou 'n' pour NON »), …

Transportabilité

Critère de qualité donné à un programme dont l'exploitation est rendue possible dans des environnements différents, au niveau matériel (PC, compatible PC, Mac, …, processeur), et/ou logiciel (systèmes d'exploitation, navigateur, …). Un tel objectif est inscrit dans le cahier des charges. Il conditionne, pour une grande partie, des choix importants (comme le langage de programmation), ce qui a pour conséquence d'augmenter le nombre de contraintes de développement.

Exemple :
Un développement de prototype propriétaire n'a pas les mêmes contraintes de transportabilité qu'un développement distribué via un réseau dont les ressources sont partagées, l'exploitation désirée est multi plateforme, ...

Maintenance aisée

Critère de qualité donné à un programme dont l'évolution, la mise à jour, la mise à niveau (ou la réparation) est prise en compte dès sa conception. Chaque modification se trouve être réalisable sans entraîner un surcroît d'analyses relatives aux autres critères de qualité.

Exemple :
Ce critère est très souvent une contrainte importante en phase de développement. Ne pas prévoir à l'avance que certaines extensions du programme seront nécessaires, peut entraîner une refonte complète du programme ou des structures de données, et par conséquent, une perte de temps et un surcoût financier considérable pour le projet.

Modularité

Critère de qualité donné à un programme qui a une incidence importante sur la maintenance de celui-ci. Aussi, un programme modulaire est un problème bien posé, analysé intégralement, et par conséquent complètement décomposé. A ce stade, l'étude des optimisations algorithmiques envisageables devient aisée, qu'elles soient imposées ou non.

Exemple :
La réalisation de programmes complexes nécessite le partage du projet et l'attribution des tâches bien définies à chacune des équipes de développeurs. Chacun prend en compte une part de l'étude en fonction des spécificités (compétences, langages, contraintes de développement, …) : groupe « moteur de jeu », groupe « stratégie de jeu », groupe « gestion de la base de données », …

Sécurité

Critère de qualité donné à un programme qui a une incidence importante sur la maintenance de celui-ci. Aussi, un programme modulaire est un problème bien posé, analysé intégralement, et par conséquent complètement décomposé. A ce stade, l'étude des optimisations algorithmiques envisageables devient aisée, qu'elles soient imposées ou non.

Exemple :
La réalisation de programmes complexes nécessite le partage du projet et l'attribution des tâches bien définies à chacune des équipes de développeurs. Chacun prend en compte une part de l'étude en fonction des spécificités (compétences, langages, contraintes de développement, …) : groupe « moteur de jeu », groupe « stratégie de jeu », groupe « gestion de la base de données », …

Sûreté

ou sûreté (de fonctionnement) :

Les termes « Sécurité » et « Sûreté » sont souvent employés l'un pour l'autre. Il est important de revenir sur une définition permettant leur distinction.

Le sens donné à la sûreté est celui de la « sûreté de fonctionnement ». Le programme réalisé ne doit pas entraver le fonctionnement d'autres programmes, du système d'exploitation ou de tout autre système lié d'une manière ou d'une autre (matériel ou logiciel).

Ce critère est complémentaire au critère de « validité » présenté précédemment. Aussi, il ne doit pas être omis le fait qu'un dysfonctionnement du monde environnant, directement causé par le programme étudié, peut avoir en retour une incidence sur lui-même.

Exemple :

L'arrêt d'un processus tiers ou de son processus « fils » peut causer des pertes irréversibles. Une mauvaise gestion des allocations de la mémoire, « fuite », entraîne, à terme, une saturation des ressources.

Immunité

Critère de qualité complémentaire à la définition donnée pour la sécurité d'un programme.

Le sens donné à l'immunité est celui de la « prévention ». Le programmeur doit faire appel à des techniques et des concepts, qui permettent de garantir une meilleure insensibilité du programme réalisé par rapport à son environnement (voire, par rapport à lui-même).

Exemple :
La gestion non contrôlée des allocations de la mémoire, écrite systématiquement de façon dynamique, reste un moyen plus risqué qu'une réservation statique (lorsque le cadre de développement l'autorise).

Lisibilité

Critère de qualité donné à un programme dont les choix relatifs à la structure générale du programme, la structuration algorithme, les identificateurs, les techniques et les concepts, … sont précisés de façon suffisamment explicite, sont argumentés, commentés, mis à jour, …

Par ailleurs, ce critère de qualité, perçu comme une contrainte inutile, a pour effet de faciliter grandement la rédaction des documents du projet et des notices associées (à destination des utilisateurs, des développeurs en charge de la maintenance, …).

Exemple :
Le choix des identificateurs facilite la compréhension de l'ensemble (par son concepteur et par autrui : le professeur, par exemple ;-). Les concepts employés, les raisons d'un choix d'une structure plutôt qu'une autre, le domaine associé à des variables, … doivent apparaître dans les programmes afin de faciliter les relectures, la maintenance, mais aussi la correction des erreurs !

Optimalité

Optimalité mono ou multi objectif :

Critère de qualité, systématique ou imposé, donné à un programme dont le niveau d'exigence en termes de « performance », est évalué. Cela correspond à une contrainte imposante en phase de développement.

Il existe deux objectifs distincts d'optimisation.

Le programme concerné doit atteindre un objectif de recherche de l’optimal : soit en terme de « processus » (réduction de la durée relative à la résolution du problème), soit en terme de « mémoire » (réduction de la capacité utilisée par le programme).

Il est très difficile de traiter les deux objectifs conjointement (l’un ayant une incidence sur l’autre et inversement). Par conséquent, le cahier des charges doit stipuler clairement, en fonction du contexte, la priorité donnée à chacun d’eux.

Exemple :
L’efficacité de la recherche d’un élément dans une structure de données, dépend de la structure choisie. Si la structure s’enrichie d’un système d’indexation, afin d’accélérer la recherche, dans ce cas, c’est la taille de la structure qui augmente.

ATTENTION ! Ne pas confondre Optimisation du code et Réduction de la taille de l'algorithme !

Simplicité

Le choix de la simplicité est certainement un des plus grands gages de qualité.

Exemple :
Parmi les différentes solutions admissibles pour résoudre un problème quelconque, celle qui reste la plus simple permet souvent d’atteindre le but (la validité) assez rapidement.

Qualité totale

Attention, la qualité fait partie de ces domaines difficilement mesurables ! Aussi, il est rappelé ici que :
« La fiabilité à 100% n’existe pas »,

« Le risque 0 n’existe pas », mais aussi …

« On ne peut pas atteindre le niveau 100% de la sécurité »,

… le niveau 100% de qualité n’existe pas non plus, en revanche, nous devons tout mettre en œuvre pour tenter de l’atteindre (principe d’obligation de moyens).

Convertisseur de chiffres romains

Un premier programme doit permettre, la saisie d'un chiffre en écriture arabe (0, …, 9), pour la traduire en chiffres romains.

Un second doit effectuer l’opération inverse.

Rappel :

Correspondance entre chiffres romains et chiffres en écriture arabe :

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Quelques exemples :

```
IV = 4,
IX = 9,
XIX = 19,
XL = 40,
XC = 90,
CD = 400,
CM = 900,
MMCDLXIX = 2469,
XVII = 17.
```

Calendrier perpétuel

Le tableau, en fin d'énoncé, permet de déterminer le jour de la semaine qui correspond à une date donnée de l'aire chrétienne, et aussi de résoudre d'autres problèmes faisant intervenir les mêmes éléments.

Mode d'emploi :

La première colonne, intitulée indice, donne les indices relatifs à tous les éléments de la ligne à laquelle ils appartiennent.

Pour obtenir le jour correspondant à une date donnée, on relève les indices relatifs :

- au chiffre des centaines, à celui des dizaines et à celui des unités de l'année considérée,

- au mois et au quantième de la date,

et l'on fait la somme de ces cinq indices. Le reste de la division par sept de cette somme est l'indice du jour cherché.

Pour les années bissextiles, on utilise les mois de janvier et de février suivi de la lettre B.

Suivant la parité du chiffre des dizaines de l'année, on utilise l'une ou l'autre des deux colonnes relatives aux chiffres des unités.

Pour faciliter l'opération, les chiffres des dizaines et des unités sont écrits en italique

lorsque les dizaines sont impaires.

Exemples :

Premier type de problème :

2 janvier 1975

Centaines 19, indice : 1
Dizaines 7 indices : 0
Unités 5, indice : 1
Mois janvier, indice : 0
Quantième 2, indice : 2
Total 4
Reste de la division par 7 : 4

Jour cherché : jeudi

12 février 1944

Centaines 19, indice : 1
Dizaines 4 indices : 5
Unités 4, indice : 0
Mois janvier B, indice : 2
Quantième 12, indice : 5
Total 13
Reste de la division par 7 : 6

Jour cherché : samedi

Second type de problème :

Trouver les vendredis 13 de l'année 1948. Soit x l'indice du mois inconnu et k un entier positif ou nul, on a :
1 + 5 + 5 + x + 6 = 5 + (7 * k) ou x = 2, les mois concernés sont août et février.

Troisième type de problème :

Quel était le troisième jeudi d'octobre 1951 ?

Réponse : le 18.

Quatrième type de problème :

En quelques années du XIXe siècle, le 29 février est-il tombé un mardi ?

Réponse : 1820, 1848, 1876.

Tableau des correspondances :

Indice	Année						Mois	Quantièmes	Jours
	Centaine		Dizaine		Unité				
	Calendrier Julien <4 oct.1582	Calendrier Grégorien >15 oct.1582	paire	impaire	dizaine paire	dizaine impaire			
1	4,11	15,19,23,27	2		5	5	Mai	1,8,15,22,29	Lundi
2	3,10		6	1	0,6	0	Août,Février B	2,9,16,23,30	Mardi
3	2,9	18,22,26,30		5	1,7	1,6	Février,Mars,Novembre	3,10,17,24,31	Mercredi
4	1,8,15		0	9	2	7	Juin	4,11,18,25	Jeudi
5	0,7,14	17,21,25,29	4		3,8	2,8	Septembre,Décembre	5,12,19,26	Vendredi
6	6,13		8	3	9	3,9	Avril,Juillet,Janvier B	6,13,20,27	Samedi
0	5,12	16,20,24,28		7	4	4	Janvier,Octobre	7,14,21,28	Dimanche

Essayer de résoudre le premier problème en essayant d'atteindre une complexité de l'ordre de la constante.

Ensuite, essayer d'envisager une solution algorithmique pour les trois autres problèmes.