Cours d'algorithmique
BTS SIO première année

Nicolas FRANCOIS
nicolas.francois@free.fr

4 septembre 2011

MCours.com

Table des matières

I Introduction **1**

- I Informatique, information . . . 2
- II Connaissances . . . 2
- III Codage . . . 2
- IV Algorithmes . . . 3
  - A D'abord, le mot ! . . . 3
  - B Oui, mais en pratique ? . . . 4
  - C Quelques exemples d'algorithmes . . . 4
  - D Et l'ordinateur dans tout cela ? . . . 5
- V Les qualités essentielles d'un bon algorithme . . . 6
  - A Entrées . . . 6
  - B Sorties . . . 6
  - C Finitude . . . 7
  - D Définition . . . 7
  - E Efficacité . . . 7
- VI En résumé . . . 8
- Annexe : quelques grands noms de l'informatique . . . 9
- TD 1 – Une introduction en douceur à l'algorithmique avec Guido . . . 10

II Les objets de bases de l'algorithmique **15**

- I Que retenir des séances de travail sur Guido ? . . . 16
- II Les commentaires, l'indentation du code . . . 16
- III Les entrées-sorties . . . 17
- IV Les variables et les types de données simples . . . 18
  - A Les variables . . . 18
  - B Les types de données simples, et les opérateurs associés . . . 19
- V Les fonctions et procédures . . . 20
  - A Procédures . . . 21
  - B Fonctions . . . 21
  - C Passage des paramètres . . . 22
- TD 2 – Affectations, entrées-sorties . . . 24

III Les structures de contrôle **27**

- I Introduction . . . 28

II Les conditionnelles 28
III Les boucles 29
TD 3 – Structures de contrôle 30

IV Les tableaux et les chaînes de caractères 35
I Les tableaux à une dimension : vecteurs 36
A Notion de tableau 36
B Exploration d'un tableau 36
II Tableaux à deux dimensions : matrices 37
III Les chaînes de caractères 38
TD 4 – Tableaux 39
TD 5 – Algorithmes de tri 42
TD 6 – Chaînes de caractères 43

V La récursivité 45
I Un premier exemple 46
II Le principe de la récursivité 46
TD 7 – Récursivité 47

Chapitre I

Introduction

Sommaire

I Informatique, information **2**
II Connaissances **2**
III Codage **2**
IV Algorithmes **3**
A D'abord, le mot ! 3
B Oui, mais en pratique ? 4
C Quelques exemples d'algorithmes 4
D Et l'ordinateur dans tout cela ? 5
V Les qualités essentielles d'un bon algorithme **6**
A Entrées 6
B Sorties 6
C Finitude 7
D Définition 7
E Efficacité 7
VI En résumé **8**
Annexe : quelques grands noms de l'informatique **9**
TD 1 – Une introduction en douceur à l'algorithmique avec Guido **10**

I. Informatique, information

Vous avez choisi un BTS informatique, il est donc bon que vous ayez une idée assez précise de ce qu'est l'informatique ! Voici la définition qu'en donne le Larousse :

> **informatique** : Science du traitement automatique et rationnel de l'information en tant que support des connaissances et des communications.

et la définition de l'information (dans son acceptation informatique) :

> **information** : Élément de connaissance susceptible d'être codé pour être conservé, traité ou communiqué.

Il ressort de ces deux définitions trois concepts importants :

- la notion de connaissance,
- la notion de codage,
- et la notion de traitement.

Nous allons détailler dans la suite de cette introduction ces trois notions.

II. Connaissances

Pour commencer, qu'est-ce que la connaissance ? De quelle manière appréhendons nous le monde qui nous entoure ? En y réfléchissant, On peut distinguer trois types de connaissances :

- La *connaissance déclarative* : elle concerne le "quoi" ; elle donne des définitions et des propriétés caractéristiques de la notion étudiée. Par exemple, on peut définir la racine carrée d'un réel positif ou nul x comme étant l'unique réel y positif ou nul dont le carré est x. Cette définition permet de *tester* si un réel est bien la racine carrée d'un autre, mais elle ne donne pas de méthode pour *déterminer* la racine carrée d'un réel.
- La *connaissance impérative* : elle concerne le "comment" ; elle donne des procédés permettant de construire les objets étudiés. Par exemple, vous avez peut-être rencontré dans vos études au lycée la *méthode de Héron*, permettant d'obtenir des valeurs approchées de plus en plus précises de la racine carrée d'un réel x :

  H1. partir d'une estimation $e > 0$ de la racine cherchée ;

  H2. si $e^2 \approx x$, s'arrêter en retournant e ;

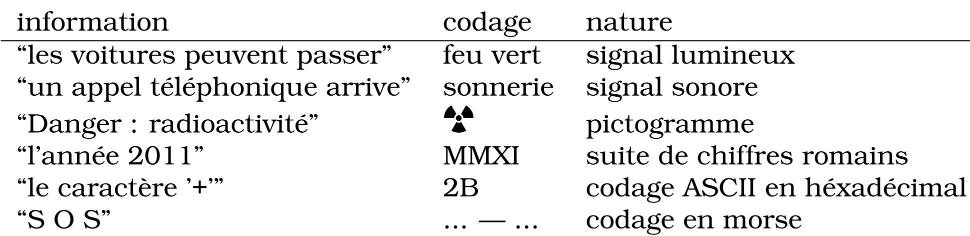
  H3. sinon, remplacer e par $\frac{1}{2}\left(e + \frac{x}{e}\right)$, et retourner à [H2].

  Les mathématiques montrent qu'un tel procédé *converge*, ce qui répond bien à nos préoccupations.
- La *connaissance conditionnelle* : elle s'occupe du "quand" ; elle donne les conditions dans lesquelles une connaissance doit être utilisée. Par exemple, on peut donner des contextes dans lesquels il peut être intéressant d'utiliser la racine carrée d'un réel, quand on connaît des propriétés de son carré.

III. Codage

Pour manipuler une connaissance, nous devons la *coder*, de manière, comme indiqué par la définition du Larousse, à pouvoir la stocker, la traiter ou la transmettre. Pour cela, nous utilisons des *symboles* : lettres, chiffres, signaux (lumineux, sonores, électromagnétiques...), pictogrammes, etc.

Voici quelques exemples :



information	codage	nature
“les voitures peuvent passer”	feu vert	signal lumineux
“un appel téléphonique arrive”	sonnerie	signal sonore
“Danger : radioactivité”	☢	pictogramme
“l’année 2011”	MMXI	suite de chiffres romains
“le caractère ’+’”	2B	codage ASCII en héxadécimal
“S O S”	... — ...	codage en morse

Le plus souvent, ce n’est pas un symbole qui est utilisé pour coder une information, mais un ensemble de symboles, appartenant à un certain *alphabet*, assemblés selon un certain nombre de “règles”. L’association d’un alphabet et de ces règles s’appelle un *langage*.

La *syntaxe* est l’ensemble des règles d’assemblage des symboles. Elle indique quelles sont les “phrases” effectivement constructibles dans le langage donné, en général en fournissant une *grammaire*. Par exemple, “$1\,2\;=\;+4$” est un phrase non valide du langage des expressions arithmétiques. Il est en général relativement simple de tester si une phrase est syntaxiquement correcte, et il existe de nombreux outils, en particulier dans le monde de l’informatique, permettant d’accomplir cette tâche de façon automatique.

La *sémantique* d’un langage exprime la signification associée aux groupes de symboles, c’est elle qui établit la correspondance entre le codage et les éléments de connaissance. Malheureusement, cette correspondance est souvent floue. Une phrase peut très bien être syntaxiquement correcte mais ne pas exprimer une connaissance valide, comme la phrase “$1 + 2 = 4$”. D’autre part, il est souvent très compliqué de vérifier si une phrase signifie effectivement quelque chose. Ce sera le travail du locuteur que de vérifier que ses phrases ont un sens.

Il faut bien comprendre qu’une même connaissance peut être codée de différentes façons, selon le langage qu’on emploie. Par exemple, l’entier 71 peut être codé :

- 71 (codage décimal),
- “soixante et onze” (codage en langue française),
- “seventy one” (codage en anglais)
- 1000111 (codage en binaire)

Réciproquement, le groupe de symboles 71 peut représenter :

- une quantité (les dossiers de 71 candidats ont été retenus par une première sélection),
- une étiquette attribuée à une classe d’objet (la ligne de bus 71),
- le code téléphonique du département de Haute-Loire,
- la notation décimale du code ASCII du caractère G,
- la notation octale du code ASCII du caractère 9,
- la notation hexadécimale du code ASCII du caractère q
- ...

Le “contexte” permet en général (mais pas toujours, ce qui conduit parfois à des catastrophes[1]) à un humain de coder ou décoder sans ambiguïté un message, mais c’est un domaine dans lequel les ordinateurs ont encore de gros progrès à faire !

IV. Algorithmes

A. D’abord, le mot !

Je ne crois pas avoir lu un seul cours d’algorithmique qui ne commence par l’origine du mot. Ne coupons pas à la tradition !

[1] Communiqué de CNN le 30 septembre 1999 : “NASA lost a 125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation, according to a review finding released Thursday.”

Le mot “algorithme” est une latinisation de la ville d'origine de Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî[2], mathématicien (entre autres) musulman perse, dont l'ouvrage le plus célèbre, *Kitab al jabr w'al muqabala*, a permis l'introduction de l'algèbre en Europe.

Voici la définition qu'en donne le Petit Robert :

> **algorithme** : Suite finie séquentielle de règles que l'on applique à un nombre fini de données, permettant de résoudre une classe de problèmes semblables.

Nous verrons dans la section consacrée aux qualités essentielles d'un bon algorithme l'importance de chacun des mots de cette définition.

B. Oui, mais en pratique ?

Comme on l'a vu dans la deuxième partie, un théorème mathématique affirmant que tout nombre positif a une racine carrée est fascinant du point de vue théorique, mais ne nous intéresse que très peu si l'on a effectivement besoin de la valeur de cette racine carrée. Dans ce cas, on a besoin d'une méthode permettant de calculer cette racine, ou bien, si cela est impossible, d'en obtenir des valeurs approchées aussi précises que possible.

L'algorithmique est la science qui étudie les problèmes du point de vue impératif, concevant des méthodes pour les résoudre, construisant leurs solutions, et étudiant les qualités et les défauts de ces méthodes. C'est elle qui construit les *traitements* des informations connues afin d'obtenir d'autres informations.

Cette notion de traitement s'articule autour de trois concepts :

description : la méthode de passage des données aux résultats est décrite par un texte (en soi, c'est aussi une information, susceptible d'être codée !),

exécution : une réalisation effective du traitement est mise en œuvre sur des données spécifiques,

agent exécutant : c'est l'entité effectuant une exécution ; cette entité est donc capable de mettre en œuvre la méthode.

Dans le contexte de l'informatique, la description sera souvent exprimée à l'aide d'un *algorithme*, l'exécutant sera un *processeur* et une exécution sera appelée un *processus*.

Mais il y a bien d'autres contextes dans lesquels on fait du traitement de l'information. Par exemple, dans une cuisine, une description sera nommée “recette”, l'exécutant sera le cuisinier, et une exécution de la recette permet de passer des ingrédients à l'objet de la recette.

Il ne faut pas confondre la description, l'exécution et l'agent : la recette, en général écrite sur une feuille de papier, n'a pas vraiment de valeur nutritionnelle, et à moins qu'on ait des mœurs spéciales, l'agent ne doit pas être consommé. Seule une exécution particulière de la recette par le cuisinier va fournir un résultat comestible.

On confond souvent, en particulier, description et exécution. Il est pourtant facile de voir que la méthode permettant de multiplier entre eux deux entiers ne doit pas être confondue avec l'exécution du produit 46×17 : la méthode ne donne pas le résultat de cette opération particulière, et l'exécution ne dit pas comment multiplier 45 et 18 !

C. Quelques exemples d'algorithmes

On a déjà rencontré quelques exemples d'algorithme :

- une recette est un algorithme permettant (si tout se passe bien !) de passer des ingrédients isolés au plat alléchant qu'on voit sur la photo ;
- lorsque votre instituteur (ou votre institutrice) vous a appris à multiplier ou diviser deux entiers, les méthodes qu'il ou elle vous a donné sont des algorithmes, que nous implémenterons lorsque nous travaillerons en TP avec les *entiers longs*.

[2] Littéralement : “Père de Ja'far, Mohammed, fils de de Moses, natif de Khowârizm”, ville maintenant nommée Khiva, qui se trouve maintenant en Ouzbékistan

L'un des premiers algorithmes mathématiques connus est le célèbre *algorithme d'Euclide*, permettant de calculer le pgcd de deux entiers (par exemple pour simplifier des fractions). Voici comment on peut l'énoncer de façon moderne[3] :

Algorithme d'Euclide : étant donnés deux entiers positifs non nuls m et n, trouver leur plus grand diviseur commun, c'est à dire le plus grand entier positif les divisant tous les deux.

E1. [Calcul du reste] : effectuer la division euclidienne de m par n, soit r le reste de cette division *(ainsi,* $0 \leqslant r < n$*).*

E2. [Est-il nul ?] : si $r = 0$, l'algorithme termine, n est le résultat.

E3. [Échange] : faire $m \leftarrow n$, $n \leftarrow r$, et retourner à l'étape E1.

Nous allons, dans la prochaine partie, expliquer les qualités exemplaires de cet algorithme, mais avant cela, un petit exercice qui va vous permettre de vous exercer à l'art subtil de la création d'un algorithme :

EXERCICES :

1) Voici un algorithme, appelé *méthode de multiplication russe*, permettant de calculer le produit de deux entiers a et b :

   - si $a = 0$, le résultat est 0 ;
   - si $a \neq 0$, alors diviser[4] a par 2, multiplier b par 2, *calculer le produit* des deux nombres résultant de ces opération, et, si a est impair, ajouter b au résultat.

   a) Que doit savoir faire l'agent exécutant pour mettre en œuvre cet algorithme ?
   b) Pourquoi est-il bien adapté à un traitement informatique ?
   c) Remarquez que pour calculer le produit de a et b, il faut savoir calculer le produit de deux nouveaux entiers (deuxième ligne) ; cette description est-elle valide ? Que doit-on faire pour la rendre valide si ce n'est pas le cas ?
   d) Utiliser cet algorithme pour calculer le produit de $a = 171$ et $b = 28$ (en base 10), puis le produit de $a = 10101011_2$ et $b = 11100_2$ (en base 2).

2) Vous disposez d'une pile de crêpes[5] de diamètres différents, et vous voudriez les ranger dans l'ordre de taille de manière à ce que la plus grande soit en dessous et la plus petite au dessus.

   Pour cela, vous ne disposez que d'une “manipulation” : vous pouvez insérer votre spatule entre deux crêpes, et retourner en bloc toute la pile de crêpes au dessus de la spatule.

   Le but de cet exercice est de trouver un algorithme efficace pour atteindre l'objectif.

   Pour ceux qui voudrait un challenge un peu plus costaud : on s'est aperçu que chaque crêpe avait un coté plus brûlé que l'autre. Comment s'assurer que la face moins brûlée soit systématiquement sur le dessus ?

D. Et l'ordinateur dans tout cela ?

Un algorithme étant connu, on peut le mettre en œuvre de manière automatique en concevant un mécanisme adapté. C'est ce que fit Pascal lorsqu'il inventa sa “Pascaline” pour effectuer les quatre opérations de base (à l'aide de roues dentées). On peut ainsi concevoir une machine pour chaque tâche qu'on souhaite automatiser, réalisant ainsi le rêve des mathématiciens depuis des siècles. Ces machines existent, et s'appellent des machines à programme fixe. Votre calculette en est un bon (et sophistiqué) exemple.

Mais allons encore plus loin. Imaginons une machine qui prendrait en entrée un algorithme, et se modifierait de manière à appliquer cet algorithme. Une telle machine pourrait donc réaliser n'importe laquelle des tâches pour lesquelles on possède un algorithme. Une telle machine existe :

---

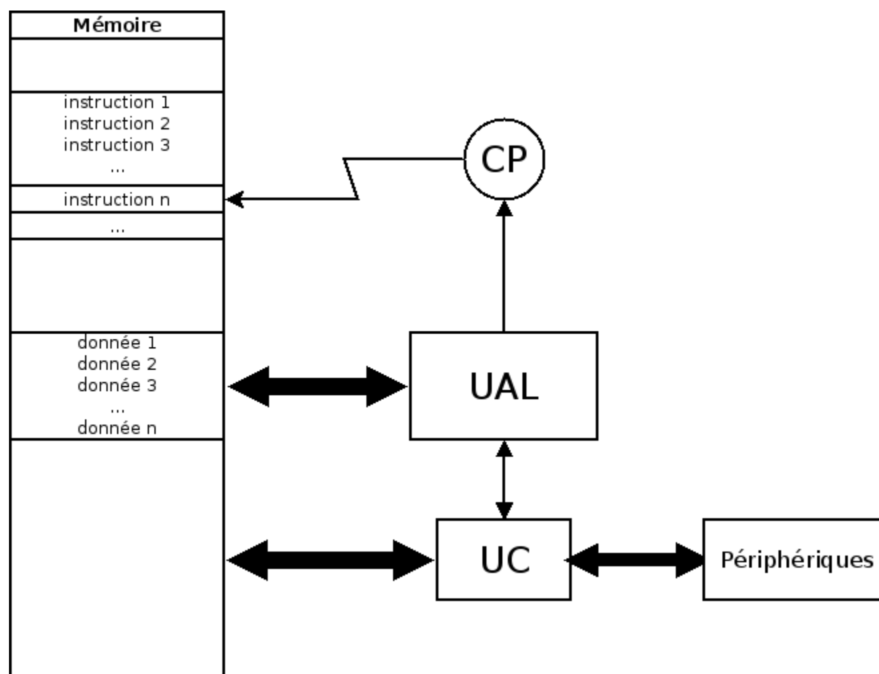
[3] tel qu'il est décrit dans le volume 1 de l'extraordinaire ouvrage du non moins extraordinaire Donald E. Knuth : “The Art Of Computer Programming”.

[4] c'est une division entière !

[5] Cet exercice est tiré de l'excellent article “genèse d'un algorithme” du site Interstices (`http://interstices.info/`), site à consulter absolument !

c'est l'ordinateur. Ainsi, tout ce que l'on a à faire pour faire faire un nouveau calcul à un ordinateur est de concevoir l'algorithme le réalisant, l'implémenter à l'aide d'un langage de programmation, et à fournir les entrées.

Voici le schéma d'un ordinateur moderne :



L'UAL, ou *unité arithmétique et logique* (ALU en anglais) est le centre de calcul. Le compteur programme CP point sur une adresse de la mémoire, contenant une instruction à faire exécuter par l'unité de contrôle UC. Celle-ci peut charger des données de la mémoire dans l'UAL, faire exécuter à celle-ci des opérations, transférer les résultats en mémoire, ou bien les transférer vers les périphériques (écran, carte réseau, clavier, souris, imprimante...).

On voit, et c'est un élément fondamental apporté par John Von Neumann, que la mémoire contient à la fois les données manipulées *et* le code du programme à exécuter. Ainsi, il est possible d'envisager des instructions qui modifie le code lui-même !

V. Les qualités essentielles d'un bon algorithme

Voici de quelle façon D.E. Knuth décrit la notion d'algorithme : en plus d'être une suite finie de règles donnant une séquence d'opérations permettant de résoudre un type spécifique de problème, un algorithme a les spécificités suivantes :

A. Entrées

Un algorithme peut avoir des entrées, qui sont des données sur lesquelles il va travailler. Par exemple, les deux entrées de l'algorithme d'Euclide sont les deux entiers m et n. Dans l'exercice proposé, l'entrée est la pile de crêpes.

Ces entrées sont des objets d'un certain ensemble aux propriétés spécifiées, et il ne faut pas s'étonner si on donne à un algorithme des données qui ne respectent pas ces spécificités.

B. Sorties

Un algorithme renvoie une ou plusieurs sorties, qui sont en relation avec les entrées. Par exemple, la sortie de l'algorithme d'Euclide est l'entier n de l'étape E2, qui est le pgcd des entrées m et n. La sortie de l'"algorithme des crêpes" est la version triée de la pile de crêpes initiale. La ou les règles permettant de relier les sorties aux entrées sont les *spécifications* de l'algorithme.

C. Finitude

On a rencontré ce mot dans les définitions données, parfois plusieurs fois. C'est un aspect essentiel de la notion d'algorithme : un algorithme doit toujours se terminer[6], après avoir exécuté un nombre *fini* d'opérations (qui peut quand même être parfois monstrueusement grand !).

Dans l'exemple de l'algorithme d'Euclide, il n'est pas évident de prouver que le procédé s'arrête. Il faut constater qu'à chaque étape, on remplace n par le reste de la division euclidienne de m par n, qui est par définition strictement inférieur à n. Ainsi, la suite des valeurs stockées dans la variable n est une suite d'entiers strictement décroissante, qui doit nécessairement prendre la valeur 0 après au plus n passages dans l'étape E1 de l'algorithme.

À titre d'exercice, vous pouvez essayer de prouver que l'algorithme de retournement des crêpes que vous avez inventé a bien un caractère fini. Ce n'est pas très compliqué.

Il faut bien faire attention au fait qu'il ne suffit pas qu'une méthode s'exprime à l'aide un nombre fini de mots pour mériter le titre d'algorithme : la phrase "avance d'un pas, tourne à droite, et recommence" est un exemple simple d'une description finie (elle comporte seulement 9 mots !) d'un processus infini, qu'on appelle souvent "boucle de la mort" en informatique : le processeur répète indéfiniment la même opération sans rien pouvoir faire d'autre.

Un procédé qui a toutes les qualités d'un algorithme, sauf celle-ci, est appelé *méthode calculatoire*. Un exemple est le calcul d'un réel défini comme limite d'une suite. Ces méthodes ont bien entendu de l'intérêt, mais le travail permettant d'en tirer des algorithmes nécessite entre autres de régler le problème de l'arrêt du calcul.

D. Définition

Chaque étape de l'algorithme doit être précisément et de façon non ambiguë définie. Ce concept nécessite de prendre en compte l'entité qui effectuera les opérations : le *processeur*.

Ainsi, on ne pourra pas énoncer l'algorithme d'Euclide de la façon dont on l'a énoncé à une personne qui ne sait pas ce qu'est une division euclidienne. De la même façon, on ne décrira pas une recette de cuisine de la même façon à un chef ayant 35 ans d'expérience et à un parfait débutant qui n'a jamais monté des blancs en neige ! Qu'est-ce qu'une "pincée" de sel pour quelqu'un qui n'a jamais fait la cuisine ? Ou une "béarnaise"[7] ?

De la même façon, vous ne pourrez pas concevoir des programmes corects sans prendre en compte les "connaissances" du langage que vous utiliserez. Dans le cadre de ce cours, nous prendrons un sous-ensemble minimal suffisant du langage Pascal, dont un résumé vous sera donné en temps voulu. Ce langage connaît la division euclidienne, mais pas le pgcd !

E. Efficacité

Les algorithmes que nous voulons concevoir ne devront pas seulement effectuer un nombre fini d'opérations pour accomplir leur tâche. Nous aimerions que ce nombre d'opérations soit *raisonnablement fini* ! Rien ne sert d'avoir un algorithme permettant de trouver la "réponse à la grande question de l'univers[8]" si cet algorithme demande un temps de calcul supérieur à l'âge de l'univers (de l'ordre de 30 milliards d'années d'après les estimations actuelles).

On ne peut malheureusement en général pas connaître le nombre exact d'opérations effectuées par un calcul. Ou plutôt, ce nombre dépend des entrées, et plus précisément de leur taille. Pour des entiers, ce sera leur grandeur, pour un tableau, le nombre de ses cellules... Nous passerons donc pas mal de temps pendant l'année à examiner le fonctionnement des algorithmes que nous concevrons, pour déterminer le plus précisément possible (ou au moins *majorer* au plus près) le nombre d'opérations nécessaires pour réaliser un calcul. Comme chaque étape de ce calcul nécessite en général plusieurs opérations, nous aurons à évaluer les opérations les plus significatives, ou bien les plus coûteuses pour le matériel.

[6] ce qui n'a en général rien d'évident, demandez à Alan Turing !

[7] Remarquons à ce sujet qu'un livre de cuisine exhaustif contient aussi la recette de la béarnaise, ainsi que la définition d'une pincée de sel, et les équivalence température-thermostat permettant de reproduire exactement les conditions dans lesquelles la recette doit être exécutées. Ces définitions et recettes peuvent être considérées comme des sous-programmes, qu'on utilise souvent, et qu'on ne veut pas réécrire à chaque fois.

[8] Consulter "Le guide du routard galactique" du regretté Douglas Adams pour plus de détails.

Par exemple, on a vu que dans l'algorithme d'Euclide, à chaque étape du calcul, on effectue une division euclidienne, suivie obligatoirement d'une comparaison du reste à 0, et d'un échange de variables (sauf à la dernière division). Comme la division est l'opération la plus compliquée effectuée à chaque étape, on utilisera le nombre de divisions effectuées comme indicateur du temps nécessaire pour le calcul.

Ainsi, si les entrées sont les entiers m et n, on a vu qu'on effectue au plus n divisions euclidiennes dans chaque étape E1. On peut ainsi *majorer* le nombre d'opérations nécessaires au calcul du pgcd de deux entiers. Mais si n est de l'ordre du milliard, cela donne un nombre très important d'opérations !

En fait, ici, l'estimation est franchement pessimiste, puisqu'on peut constater qu'il faut beaucoup moins de n divisions quelques soient les valeurs de m et n choisies. Déterminer le nombre exact de divisions est ici impossible, mais on peut démontrer qu'il est majoré par $\ln n$, $\ln$ étant le logarithme népérien (voir le cours de mathématiques).

Un autre exemple : pour trier un tableau comportant n entrées, les algorithmes peu efficaces ont besoin d'effectuer de l'ordre de n^2 opérations. Le tri d'un tableau de 10 entrées se fait donc de façon quasi instantanée. Par contre, que se passerait-il si l'on souhaitait trier un tableau comportant une soixantaine de millions d'entrées (par exemple, un tableau recensant la population d'un pays comme la France) ? Il faudrait alors de l'ordre de 4 millions de milliards d'opérations. En supposant que l'ordinateur utilisé effectue un milliard de comparaisons par seconde (ce qui est *très* optimiste !), il faudrait environ 4 millions de secondes pour trier ce tableau, soit 45 jours !

Certains algorithmes que nous rencontrerons nécessitent encore plus d'opérations, et ne sont donc en pratique utilisables que pour de petites tailles des données.

D'autre part, il n'y a pas que le temps qui soit déterminant dans le fonctionnement d'un algorithme. De la même façon, la mémoire (qu'elle soit vive ou de masse) dont nous disposons est limitée. Il ne faudrait pas que pour effectuer un calcul, il faille plus de bits de mémoire que le nombre d'atomes de l'univers (de l'ordre de 10^{100} à quelques dizaines de zéros près) !

Dans le cas de l'algorithme d'Euclide, il est assez facile de se convaincre que trois cases mémoire suffisent pour l'ensemble du calcul. En effet, une fois r déterminé dans l'étape E1, on peut se passer de la valeur de m, et donc y ranger l'ancienne valeur de n, puis ranger la valeur de r dans la case n ainsi libérée.

VI. En résumé

En pratique, les questions essentielles auquel il faut répondre lorsqu'on conçoit un algorithme sont donc :

1) l'algorithme se termine-t-il (et ce pour n'importe quelle entrée, et non pas seulement les quelques-unes que l'on a testé) ?

2) l'algorithme est-il correct, c'est-à-dire renvoie-t-il le bon résultat dans tous les cas ?

3) l'algorithme est-il performant (ce qui revient à trouver un lien entre le temps et/ou l'espace mémoire nécessaire et la "taille" des entrées) ?

Ces questions sont absolument fondamentales de nos jours. En effet, la plupart des systèmes numériques sont maintenant "embarqués" dans des outils dont notre tranquillité, notre sécurité, voire notre vie, dépend. Peut-on par exemple imaginer un système de gestion du freinage d'une voiture qui ne fonctionne pas lorsqu'un capteur est encrassé, ou bien qui réagit dans certaines conditions trois ou quatre secondes trop tard ?

On peut souvent associer les deux premiers points dans la recherche d'une *preuve*, et souvent, l'analyse de cette preuve permet d'obtenir une estimation du nombre de fois où chaque instruction de l'algorithme est effectuée, permettant de répondre au troisième point.

MCours.com

ANNEXE : QUELQUES GRANDS NOMS DE L'INFORMATIQUE

- **Blaise Pascal** (France, 1623-1662) a construit en 1642 l'un des premiers calculateurs mécaniques, la célèbre *Pascaline*, pour aider son père à calculer les taxes de la région de Haute-Normandie.

- C'est **Charles Babbage** (Grande Bretagne, 1791-1871) qui a le premier conçu les plans d'une machine à calculer mécanique "programmable" (à l'aide de carte perforées utilisées à l'époque pour les machines à tisser), la *machine analytique*. On y trouve tous les "ingrédients" des ordinateurs modernes. Malheureusement, cette machine n'a pas été concrétisée à l'époque (mais des scientifiques modernes se sont amusés à en construire une pour prouver que le bestiau était réalisable).

- **Ada Lovelace** (Grande Bretagne, 1815-1852) a travaillé sur le modèle théorique de Babbage et peut être considérée comme la première "programmeuse". On a d'ailleurs donné son prénom à un langage de programmation.

- Les travaux en logique de **George Boole** (Grande Bretagne, 1815-1864) et **Gottlob Frege** (Allemagne, 1848-1925) ont largement contribué à la science des ordinateurs.

- Les travaux théoriques de **Kurt Gödel** (Allemagne, 1906-1978), et des américains **Alonzo Church** (1903-1995), **Stephen Cole Kleene** (1909-1994), **Emil L. Post** (1897-1954) et **Claude Shannon** (1916-2001), ont largement contribué à faire avancer la science des ordinateurs.

- **Alan Turing** (Grande Bretagne, 1912-1954) a travaillé sur le concept de calcul, et l'a concrétisé en une machine théorique qui sert encore aujourd'hui de référence. Cette *machine de Turing*, extrêmement rudimentaire, permet de définir précisément ce qu'est un calcul réalisable par une machine.

- Les américains **J. Presper Eckert** (1919-1995) et **John Mauchly** (1907-1980) ont construit le premier ordinateur digital électronique, l'ENIAC, en 1945.

- **John von Neumann** (Hongrie, 1903-1957) a entre autres décrit l'architecture des ordinateurs, qui est toujours utilisée de nos jours.

- **Grace Hopper** (États-Unis, 1906-1992) conceptualisa la notion de langage de programmation indépendant de la machine. On peut en quelque sorte la considérer comme la mère de tous les langages informatiques modernes !

- Quelques langages célèbres, avec leur(s) inventeur(s) : COBOL[9] (collectif), FORTRAN (**John Backus**), LISP (**John McCarthy**), C (**Dennis Ritchie** et **Ken Thompson**, aussi auteurs du système d'exploitation UNIX), PASCAL (**Niklaus Wirth**), JAVA (**James Gosling** et **Patrick Naughton**, PYTHON (**Guido van Rossum**)... que les nombreux autres qui ne sont pas cités ici me pardonnent !

- Une mention spéciale à **Donald E. Knuth** (États-Unis, 1938-) et **Edsger W. Dijkstra** (Pays-Bas, 1930-2002), qui ont joué un grand rôle dans la science qui nous intéresse tout particulièrement, l'algorithmique. En particulier, "The Art Of Computer Programming", l'une des œuvres majeures de D. Knuth, est une somme sans équivalent sur les algorithmes et les structures de données. À noter que D. Knuth est aussi l'auteur de TEX, dont est dérivé LATEX, langage de composition de documents utilisé par l'ensemble de la communauté scientifique. Il est remarquable de constater qu'après 34 ans de bons et loyaux services, on a toujours pas trouvé mieux pour écrire des textes scientifiques ! Une telle longévité est rarissime dans le monde de l'informatique.

  À titre d'exercice, vous pouvez chercher comment prononcer les noms de ces deux scientifiques, et la signification des initiales E, pour Knuth, et W, pour Dijkstra !

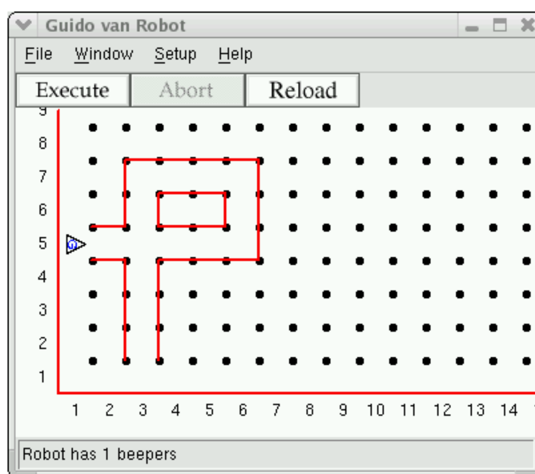
[9] pour faire plaisir à D. Gallot !

TD 1 – Une introduction en douceur à l'algorithmique avec Guido

1) **Présentation de Guido**

"Guido Van Robot" est un environnement de programmation d'un petit robot utilisant un micro-langage, en fait un sous-ensemble minimum de Python. Bien que ce langage soit extrêmement rudimentaire, il est très pédagogique.

Guido est un robot évoluant dans un monde en deux dimensions, constitué de rues et d'avenues. Voici un exemple de situation de départ :




```
define untour:
    do 3:
        move
    turnleft
    do 2:
        move
    turnleft
    do 3:
        move
    turnleft
    do 2:
        move

do 2:
    move
putbeeper
untour
pickbeeper
do 3:
    move
turnoff
```

Guido est matérialisé par une triangle dont l'orientation indique dans quelle direction il va avancer. Il se déplace de case en case, les coins des cases étant matérialisés par les points noirs. Les traits rouges sont des murs, que le Robot ne peut traverser. La barre de statut indique que le robot transporte un "beeper", une sonnette en français. Il peut à loisir prendre des sonnettes présentes sur le terrain ou en déposer s'il en a dans son *sac à sonnettes*.

Pour diriger Guido, on écrit un *code*, en utilisant les commandes connues par Guido, ou celles que l'on définit soi-même. Guido peut ainsi avancer, tourner sur lui-même, tester son environnement (présence ou absence de murs, de sonnettes...). On peut lire ci-dessus un exemple d'un tel code. Le plus simple est que vous constatiez de vous-même ce que ce programme fait effectuer au robot.

Les 12 premières lignes ne sont pas exécutées, elles constituent une *définition de fonction* permettant de découper la tâche à effectuer en sous-tâches plus simples à appréhender. La première commande exécutée par le robot est une *boucle* `do 2`, qui lui demande de répéter deux fois les instructions suivant la ligne `do` qui sont *indentées*. Ensuite, il dépose sa sonnette (`putbeeper`), exécute l'ensemble des instructions définissant la fonction `untour`, ramasse sa sonnette (`pickbeeper`), avance encore de trois pas, et s'éteint.

Pour bien comprendre ce qui se passe, le plus simple est que vous essayiez de modifier une ligne ou deux, pour voir ce qui se passe. Et s'il se passe quelque chose qui n'est pas prévu, tentez de comprendre en suivant pas-à-pas le déroulement des instructions.

2) **Les commandes de Guido**

Les commandes permettant de contrôler Guido sont de 3 types :

- les commandes *primitives* :
  - `move` : avancer d'une case vers l'avant,
  - `turnleft` : tourner de 90° vers la gauche,
  - `pickbeeper` : ramasser une sonnette,
  - `putbeeper` : déposer une sonnette,
  - `turnoff` : s'éteindre ;
- les *tests* :
  - `front_is_clear` : vrai si il n'y a pas de mur devant,
  - `front_is_blocked` : vrai si il y a un mur devant,
  - `left_is_clear` : vrai si il n'y a pas de mur à gauche,
  - `left_is_blocked` : vrai si il y a un mur à gauche,
  - `right_is_clear` : vrai si il n'y a pas de mur à droite,
  - `right_is_blocked` : vrai si il y a un mur à droite,
  - `next_to_a_beeper` : vrai si la case comporte une sonnette,
  - `not_next_to_a_beeper` : vrai si la case ne comporte pas de sonnette,
  - `any_beepers_in_beeper_bag` : vrai si Guido transporte au moins une sonnette,
  - `no_beepers_in_beeper_bag` : vrai si Guido ne transporte aucune sonnette,
  - `facing_north` : vrai si Guido regarde vers le nord,
  - `not_facing_north` : vrai si Guido ne regarde pas vers le nord,
  - `facing_south` : vrai si Guido regarde vers le sud,
  - `not_facing_south` : vrai si Guido ne regarde pas vers le sud,
  - `facing_east` : vrai si Guido regarde vers l'est,
  - `not_facing_east` : vrai si Guido ne regarde pas vers l'est,
  - `facing_west` : vrai si Guido regarde vers l'ouest,
  - `not_facing_west` : vrai si Guido ne regarde pas vers l'ouest ;
- les *structure* :
  - la *conditionnelle* : "`if <condition>:  <actions>`" qui exécute les actions si la condition est vraie, et rien sinon,
  - la *boucle itérative* : "`do <nombre>:  <actions>`" qui exécute les actions le nombre de fois indiqué,
  - la *boucle conditionnelle* : "`while <condition>:  <actions>`" qui exécute les actions *tant que* la condition reste vraie,
  - la *définition de fonction* : "`define <fonction>:  <instructions>`" qui définit `fonction` comme un raccourci pour la séquence des `instructions`.

Un tutoriel est disponible dans le répertoire de la classe, il vous est demandé d'en suivre au moins le début, et de faire les exercices, afin de vous familiariser avec l'ensemble des commandes.

3) **Des exercices**

Exercice n°1 : Guido à Rolland-Garros

31) Guido veut travailleur à Rolland-Garros comme ramasseur de balles. Le responsable lui fait passer un test : partant de la situation de la figure 1, Guido doit ramasser toutes les balles (les sonnettes) disséminées sur le terrain et les remettre dans la corbeille dans le coin inférieur gauche, pour arriver à la situation de la figure 2.

    Écrivez un code permettant à Guido de réaliser cette opération. Il devra comporter une boucle `do` répétant 6 fois le ramassage d'une rangée de balles, cette opération étant elle-même constituée d'une boucle *do*.

32) Ça y est, Guido est engagé ! Lors de son premier travail, il doit ramasser les balles laissées sur le terrain après l'entraînement d'un champion. Mais il se rend compte que la situation (celle de la figure 3) est un peu différente de celle de son test d'embauche : il y a parfois plusieurs balles au même endroit, et d'autres endroits ne comportent pas de balle.

    Modifiez le code de votre premier programme de manière à ce que Guido puisse accomplir cette nouvelle tâche.

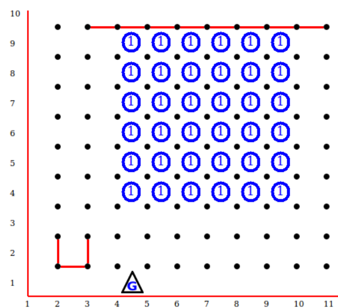


Figure 1

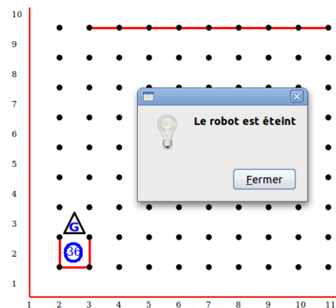


Figure 2

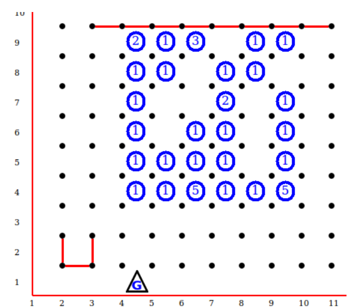


Figure 3

Exercice n°2 : Guido fait la course

31) Guido a décidé de faire du sport : il veut se mettre à la course de haie. Une course se passe de la manière suivante : Guido part du coin inférieur gauche, comme indiqué à la figure 4. Il doit courir vers la droite et contourner les haies représentées par des murs, et s'arrêter à la 17ème intersection (figure 5).

    i. Écrivez une fonction `saute_haie` indiquant à Guido comment "sauter une haie" : partant de la gauche de la haie, regardant vers l'est, il doit se retrouver derrière la haie, regardant encore vers l'est.

    ii. Utilisez cette fonction pour écrire un programme réalisant l'intégralité de la course.

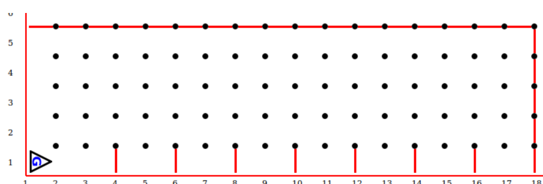


Figure 4

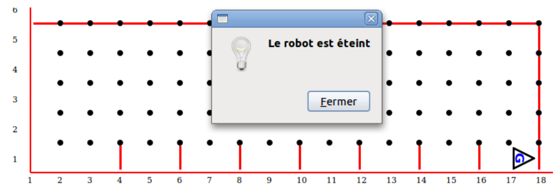


Figure 5

32) Guido trouve la course de haies trop facile, il veut se mettre à la "course de murs" : les obstacles sont disposés aux mêmes endroits sur la piste, mais il sont de hauteur variable (voir figure 6).

    Modifiez votre code de manière à permettre à Guido d'exceller dans ce nouvel exercice.

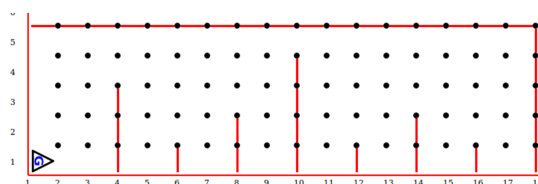


Figure 6

4) Travail personnel

Voici trois exercices plus complexes, qui ne sont pas simplement des exercices de programmation, mais pour lesquels il faudra *concevoir* un véritable algorithme. Il vous faudra réfléchir à une méthode plus ou moins astucieuse pour parvenir aux objectifs à atteindre, en tenant compte des contraintes.

De l'autre coté du miroir

Guido se trouve d'un coté d'un mur horizontal, et voudrait bien passer de l'autre coté. Le malheur, c'est qu'il ne sait pas quelle est la largeur du mur. La figure 7 montre la situation initiale, et la figure 8 la situation finale.

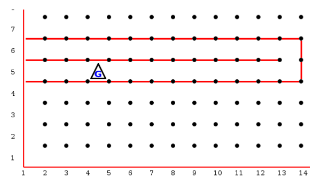


Figure 7

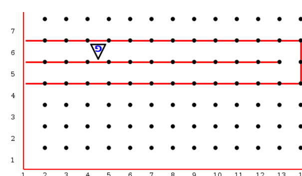


Figure 8

41) Dans un premier temps, on va supposer que :
   - Guido n'a pas de sonnettes dans son sac à sonnettes,
   - le parcours jusqu'au bord du mur est semé de sonnettes,
   - par contre, il n'y a pas de mur vertical pour indiquer le bord du mur.

   La figure 9 montre la situation de départ :

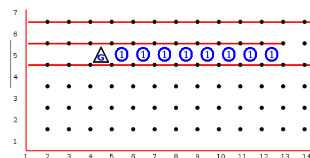


Figure 9

   Concevez un algorithme permettant à Guido de se retrouver de l'autre coté du miroir. Vous avez le droit de ramasser les sonnettes, de les déposer autre part, seule la position finale de Guido sera prise en compte pour évaluer votre travail. Bien entendu, votre algorithme doit s'adapter à la taille du mur. Vous devrez le tester avec un mur de longueur différente pour vérifier ce fait.

42) Dans cette deuxième partie, il n'y a pas de sonnettes sur le terrain. La situation initiale est donc celle de la figure 7. Par contre, Guido a un sac rempli de sonnettes, dont il peut disposer à sa guise. Vous pouvez compter sur la présence du mur à droite, mais vous n'êtes pas obligé de l'utiliser !

   Concevez un algorithme permettant à Guido de passer de l'autre coté du miroir. Encore une fois, seule la position finale du robot sera évaluée.

L'échiquier

Le père de Guido lui a demandé de peindre un échiquier. Il part d'un terrain rectangulaire délimité par des murs, comme dans la figure 10 ci-dessous, et doit peindre une case sur deux en y déposant une sonnette. Le résultat à atteindre est montré à la figure 11.

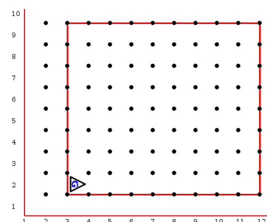


Figure 9

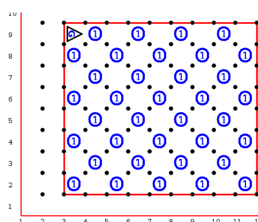


Figure 10

Bien entendu, Guido ne connaît pas initialement la taille du terrain, votre programme doit donc être adaptable. En particulier, ne pas oublier le cas des terrains filiformes, donc une (ou les deux !) dimension est égale à 1.

Pour les plus forts : trouver le milieu

Guido se trouve dans une pièce rectangulaire dont il ne connaît pas la largeur. Son père lui a demandé de positionner une sonnette au milieu de manière à percer un trou dans le mur pour accrocher la télévision.

Tout ce dont Guido dispose, c'est d'un sac rempli de sonnettes. Il sait qu'il se trouve à gauche de la pièce. Si la pièce est de largeur impaire, la sonnette devra être au milieu. Par contre, si la pièce est de largeur paire, l'une ou l'autre des deux positions centrales sera satisfaisante. Ainsi les deux figures ci-dessous sont acceptées comme positions finales :

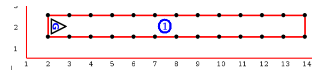


Figure 11

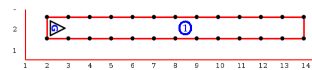


Figure 12

Concevez un algorithme permettant à Guido de positionner sa sonnette.

Chapitre II

Les objets de bases de l'algorithmique

Sommaire

I Que retenir des séances de travail sur Guido ? . . . 16
II Les commentaires, l'indentation du code . . . 16
III Les entrées-sorties . . . 17
IV Les variables et les types de données simples . . . 18
A Les variables . . . 18
B Les types de données simples, et les opérateurs associés . . . 19
V Les fonctions et procédures . . . 20
A Procédures . . . 21
B Fonctions . . . 21
C Passage des paramètres . . . 22
TD 2 – Affectations, entrées-sorties . . . 24

I. Que retenir des séances de travail sur Guido ?

Nous avons vu dans les premières séances utilisant l'outil d'initiation *Guido van Robot* la plupart des éléments de bases des algorithmes[1] : les commandes élémentaires (avancer, tourner...), les *structures* (conditionnelles, boucles itératives ou conditionnelles) et les *fonctions*.

Nous allons voir dans ce premier chapitre les objets de base manipulés par les algorithmes : les *variables*. Nous verrons aussi de quelle manière l'ordinateur communique avec le monde extérieur.

II. Les commentaires, l'indentation du code

Du point de vue de leur action sur l'ordinateur, les commentaires ne servent à rien ! Le compilateur (ou l'interpréteur), lorsqu'il rencontre un commentaire, l'ignore superbement et recherche l'instruction suivante.

Pourquoi alors commencer par quelque chose qui ne sert à rien ? Tout simplement parce que pour nous pauvres humains, ces commentaires sont au contraire essentiels ! Voici un certain nombre de raisons :

- écrire un commentaire permet de s'assurer de la validité du code qu'on écrit ; si l'on ne sait pas pourquoi on écrit une ligne de code, c'est peut-être qu'il faut revoir l'algorithme dans son ensemble !
- si le code qu'on a écrit ne fonctionne pas, ou ne donne pas les résultats attendus, il va falloir le relire ; dans ce cas, il est toujours agréable d'avoir un balisage, surtout si le source à relire dépasse quelques dizaines de lignes ;
- si une autre personne utilise le code, et veut le corriger, l'étendre, ou simplement le comprendre, il est encore plus fondamental qu'il soit très précisément commenté ; il est très difficile de comprendre un code qu'on n'a pas écrit soi-même[2] ; donc dans un environnement de travail où plusieurs personnes doivent cohabiter, des commentaires précis doivent être ajoutés (il est d'ailleurs utile, dans ce cadre, de définit des règles de présentation et d'organisation du code communes) ;
- lorsque vous écrirez des algorithmes lors de vos évaluations, il vous sera demandé de commenter certains passages de votre code, pour montrer que vous avez compris ce que vous faites, ou pour justifier vos choix ; plus vous faciliterez la tâche du correcteur, meilleure sera votre note, toutes autres choses étant égales par ailleurs !

Algorithmique	Traduction Pascal
(∗ Un commentaire ∗)	*(∗ Un commentaire ∗)* *{ Un autre commentaire* *sur plusieurs lignes }* *{ (∗ un commentaire imbriqué* *dans un autre ∗) }*

Toujours pour des raisons de lisibilité, on s'attachera à *indenter* correctement le code. Nous avons vu avec les séances Guido que c'était une nécessité syntaxique en Python. En Pascal, cela n'a aucune importance syntaxique, cela en a par contre une grande pour la lisibilité. Comparez les deux présentations suivantes :

[1]à l'exception peut-être de la notion de *variable*, qui existe, mais est un peu cachée !

[2]C'est même un indice du fait que la programmation est plus un art qu'une science : à chaque programmeur son style, et sa façon d'interpréter la partition des spécifications qui lui sont imposé.

```
Program ToursDeHanoi;
Var disks:integer;
Procedure Hanoi(source, temp, destination: char;  n: integer);
begin if n>0 then begin
Hanoi(source,destination,temp,n-1);
writeln('Deplacer le disque ',n:1,' du pied ',source,' au pied ',destination);
Hanoi(temp,source,destination,n-1);
end;end;
begin
write('Entrer le nombre de disques : ');
readln(disks);
writeln('Solution : ');
Hanoi('A','B','C',disks);
end.
```

Un code illisible

```
Program ToursDeHanoi;

Var disks : integer;

Procedure Hanoi(source, temp, destination: char;  n: integer);

begin
   if n > 0 then
   begin
      Hanoi(source, destination, temp, n - 1);
      writeln('Deplacer le disque ',n:1,' du pied ',source,' au pied ',destination);
      Hanoi(temp, source, destination, n - 1);
   end;
end;

begin
   write('Entrer le nombre de disques : ');
   readln(disks);
   writeln('Solution : ');
   Hanoi('A','B','C',disks);
end.
```

Un code plus clair

III. Les entrées-sorties

Pour interagir avec son (ou ses) utilisateur, un programme doit pouvoir lire des données provenant du “monde extérieur”. Nous utiliserons essentiellement le clavier comme instrument d’entrée, même si on parlera des lectures de fichiers. On peut aussi envisager des lectures de données présentes sur la carte réseau, ou les actions d’une souris. Vous verrez tout ceci en détail avec vos professeurs d’informatique.

Nous utiliserons l’instruction `Lire` (ou `Lireln`) lorsque nous écrirons un algorithme. Sa traduction en Pascal est `read` ou `readln`[3].

Pour écrire le résultat d’un calcul, un programme doit de même pouvoir transmettre des données au monde extérieur. De la même façon, nous écrirons principalement nos résultats sur une *console* (un écran en mode texte). Nous parlerons plus tard d’écriture de fichiers, et vous verrez en cours d’informatique d’autres moyens de communication plus sophistiqués, comme l’écran graphique, le réseau...

[3] La différence entre les deux versions de la fonction est simplement un passage à la ligne après la dernière saisie pour la version `readln`.

La version algorithmique de l'opération d'écriture est `Ecrire` ou `Ecrireln`, et sa version Pascal est `write` ou `writeln`[4].

EXEMPLES :

1) Le programme le plus simple qu'on peut concevoir est le classique "Bonjour Monde" ("Hello World" pour les anglais) :

Algorithmique	Traduction Pascal

Ecrire ('Bonjour Monde !')	**program** HelloWorld;
	
	begin
	    writeln ('Bonjour␣Monde␣!');
	    readln
	end.

2) Voici un programme qui demande le prénom de l'utilisateur, et lui souhaite la bienvenue :

Algorithmique	Traduction Pascal

Ecrire ('Quel est votre prenom ? ')
Lire (prenom);
Ecrire ('Bonjour, ',prenom,' !');
	program HelloWorld;
	
	var prenom : string;
	
	begin
	    write ('Quel␣est␣votre␣prenom␣?␣');
	    readln (prenom);
	    writeln ('Bonjour,␣',prenom,'␣!');
	    readln
	end.

IV. LES VARIABLES ET LES TYPES DE DONNÉES SIMPLES

A. Les variables

Un ordinateur n'a aucune "mémoire". Ou plutôt, il ne met pas automatiquement comme nous en mémoire les résultats de ses calculs. Si l'on veut qu'il se souvienne d'un résultat, il faut le lui indiquer explicitement, sous forme d'une *instruction d'affectation*.

En Pascal comme dans beaucoup d'autres langages, les variables doivent être *déclarées* avant d'être utilisées, comme indiqué ci-dessous :

Algorithmique	Traduction Pascal
x <– 2+3	**var** x : **integer**; ... x := 2+3;

Les instructions ci-dessus font les choses suivantes :

- la déclaration `var` prépare de l'espace en mémoire pour une variable nommée `x`, qui sera de type `integer` (entier),
- le calcul à droite du symbole d'affectation := est effectué,

[4]Même remarque, plus une autre : en informatique, si on veut écrire des accents dans les chaînes de caractères, on doit se préparer à quelques nuits blanches à lire des documentations, évidemment sans accents car en anglais ! Pour éviter des tracas (qui en plus ne sont pas transposables d'un environnement à l'autre), on évitera l'usage des accents dans le cours d'algorithmique. Mme Gallot vous dira certainement que c'est une bien piètre solution, mais je fais ce que je veux !

- le résultat de ce calcul est rangé dans la case mémoire dont l'adresse est référencée par le nom "x".

Ainsi, si l'on a besoin de cette valeur plus tard, il suffira d'utiliser le nom de la variable :

Algorithmique	Traduction Pascal
y <– x–1	**var** y : **integer**; ... y := x–1;

Ici, l'ordinateur commence par calculer "x-1", remplaçant x par la valeur 5 stockée dans la mémoire de nom x, puis il range le résultat (soit 4 dans la case mémoire de nom[5] y.

D'un point de vue pratique, il est important que les noms des variables soient correctement choisis : positionx est bien plus clair à relire que u ! Un indice de boucle sera en général noté i, j, k... Ne pas hésiter à commenter le code pour éclairer le lecteur sur la signification et l'utilité d'une variable.

EXERCICE : Comment implémenteriez vous la notion de variable dans l'environnement Guido[6] ? Cela vous semble-t-il pratique ?

B. Les types de données simples, et les opérateurs associés

Les types les plus simples[7] sont :

boolean : le type des booléens, les valeurs de vérités, notées par les constantes false (pour faux) et true (pour vrai). Les opérateurs utilisables avec ces variables sont :

Opérateur	Algorithmique	Traduction Pascal
négation	non	not
conjonction	et	and
disjonction	ou	or
ou exclusif	oubien	xor

integer : le type des entiers signés, appartenant au sous-ensemble $[-\texttt{maxint}, \texttt{maxint}]$, maxint étant une valeur prédéfinie dépendant de l'ordinateur utilisé. Pour un ordinateur 32 bits, $\texttt{maxint} = 2^{32} - 1 = 2\,147\,483\,647$. Les opérateurs utilisables avec ces variables sont :

Opérateur	Traduction Algorithmique/Pascal
addition	+
soustraction	-
multiplication	*
division entière	div
reste d'une division	mod

On remarquera qu'il n'y a pas d'opérateur d'exponentiation, et on se souviendra que l'opérateur div est l'opérateur de division *entière* : 5 div 2 donne pour résultat 2, et non pas 2.5 (qui n'est d'ailleurs pas un entier !).

real : le type des réels en virgule flottante. En dehors de 0, sont codés les nombres *décimaux* compris entre 10^{-37} et 10^{37}, et ayant 7 ou 8 chiffres après la virgule dans leur représentation en notation ingénieur (on parle de *chiffres significatifs*. On utilise la lettre E pour l'exposant

[5]Notons qu'en général, le programmeur comme l'utilisateur du programme ne connaissent pas l'endroit de la mémoire pointé par les noms x et y. Ces "adresses" sont fournies par le système d'exploitation à la demande, et peuvent très bien changer d'une exécution du programme à l'autre. Ces problèmes de gestion de mémoire sont très intéressants, mais sortent largement du cadre de ce cours.

[6]Sachez qu'il existe une version étendue de GvRng qui inclue le concept de variable, et qui en fait est une véritable initiation à la programmation Python : c'est RUR-PLE (http://rur-ple.sourceforge.net/). Les plus "accros" à Guido y trouveront d'autres exercices facilement transposable à GvRng, et montrant qu'on peut demander à Guido d'apprendre à additionner des nombres !

[7]qui sont d'ailleurs ceux que peut renvoyer une fonction en Pascal

: -123450 est noté `-1.2345E5`, ce qui signifie $-1{,}2345 \times 10^5$, 0.0001548 se note `1.548E-4` qui signifie $1{,}548 \times 10^{-4}$.

Les opérateurs utilisables avec ces variables sont :

Opérateur	Algorithmique	Traduction Pascal
addition	+	+
soustraction	-	-
multiplication	*	*
division	/	/
partie entière	`ent`	`trunc`
carré	$()^2$	`sqr`
racine carrée	$\sqrt{}$	`sqrt`
logarithme népérien	ln	`ln`
exponentielle	exp	`exp`
nombre aléatoire	`rand`	`random`

On remarquera encore une fois qu'il n'y a pas d'opérateur d'exponentiation. On verra dans le cours de mathématique comment en fabriquer un. D'autres fonctions existent, mais sont moins utiles, la documentation permet de les retrouver au cas par cas.

D'autre part, le générateur de nombre aléatoire doit être initialisé par un appel à la fonction `randomize`. `rand` renvoie un nombre pseudo-aléatoire x tel que $0 \leqslant x < 1$, et si `limite` est un entier positif, `rand(limite)` renvoie un entier y tel que $0 \leqslant y <$ `limite`.

char : c'est le type des *caractères*. Chaque caractère est codé sous la forme d'un octet représentant son *code ASCII*. On écrit les caractères usuels entre guillemets : `'T'`. On peut aussi écrire leur code ASCII précédé du symbole `#` : `#65` est équivalent à `'A'`. Enfin, on peut utiliser les caractères de contrôle en écrivant une lettre précédée d'un accent circonflexe : `^G` est équivalent à `#7`, et permet d'émettre une sonnerie.

V. Les fonctions et procédures

Il est assez rare qu'une tâche puisse être réalisée d'un bloc. En général, le problème à résoudre sera découpé en sous-tâches plus simples, soit parce que cela simplifiera leur mise au point, soit parce que certaines sous-tâches seront utilisées à plusieurs endroit du programme.

Par exemple, un programme réalisant un crible d'Erathostene est très intéressant en lui-même, mais il est plus que probable que le programmeur ne veut pas juste contempler le tableau de nombres premiers qu'il produit, mais utiliser ces nombres premiers pour réaliser une autre tâche. On pourrait bien-sûr insérer le code de notre programme dans un programme plus vaste, mais ça n'est pas la bonne méthode.

Dans un autre programme, on peut avoir besoin de réaliser une tâche complexe (c'est-à-dire ne se réduisant pas à une ou deux instructions) à plusieurs endroits différents du code. On peut bien entendu écrire une fois le code, puis faire des copier-coller partout où on en a besoin. Mais là encore, ce n'est pas la bonne méthode. En effet, que se passera-t-il si l'on s'aperçoit qu'une erreur a été commise ? Ou bien si on veut modifier notre façon de procéder ? Il faudra alors chercher dans tout le programme le code à corriger, ce qui s'avèrera bien compliqué si le projet dépasse les quelques centaines de lignes.

Une bien meilleure méthode consiste à regrouper l'ensemble des instructions réalisant la sous-tâche dans une boîte, et se donner un moyen d'utiliser cette boîte là où on en a besoin. On appelle une telle boîte une *procédure*, ou une *fonction*, selon le mode d'utilisation.

Cette méthode de découpage des programmes en procédures et fonctions réalisant une partie de la tâche principale est très efficace pour analyser correctement un programme, le tester efficacement, et aussi pour diviser le travail entre plusieurs programmeurs. Il est alors nécessaire de bien préciser les spécifications de chacun des éléments, de manière à ce qu'ils s'emboîtent bien les uns dans les autres !

Par la suite, nous nous contenterons très souvent d'écrire des procédures et des fonctions, et ce sera votre travail que de les inclure dans des programmes pour les tester et les utiliser.

A. Procédures

Une procédure réalise une tâche à partir des arguments qui lui sont donné. Une procédure est aussi appelée un *sous-programme*. En pascal, elle est définie de la façon suivante :

```
procedure <nom>(<liste des paramètres>);

var <liste des variables internes>

begin
  <corps de la procedure>
end;
```

Ceci définit une procédure de nom `nom`. Les variables internes sont les variables qui seront utilisées pour réaliser le calcul. Ces variables sont créées au moment où la procédure est utilisée, et détruites la fin de son exécution. Elle ne sont pas *visibles* à l'extérieur de la procédure.

Lorsque la procédure est appelée, les instructions formant son corps sont exécutées avec les valeurs des paramètres. Voici un exemple d'une fonction `range` qui prend en argument deux entiers `x` et `y`, et qui place dans `x` la plus grande valeur et dans `y` la plus petite :

```
procedure range(var x,y : integer);

var u : integer;

begin
   if x<y then begin
(* si x<y,                  *)
      u := y;
      y := x; (* on echange les valeurs de x et y *)
      x := u
   end       (* sinon, il n'y a rien a faire !
*)
end; {range}
```

B. Fonctions

La seule différence notable entre une procédure et une fonction est que cette dernière renvoie une valeur.

```
function <nom>(<liste des paramètres>) : <type>;

var <liste des variables internes>

begin
  <corps de la fonction>
end;
```

`type` est le type de la valeur retournée, qui ne peut être qu'un type simple (une fonction ne peut pas renvoyer un tableau ou une chaîne de caractères). Dans le corps de la fonction, `nom` est utilisé comme une variable de type `type`, et sa valeur finale est retournée comme résultat du calcul.

Voici par exemple une fonction `sdiv` prenant en argument un entier `n`, et renvoyant la somme de ses diviseurs, ainsi qu'un programme l'utilisant pour trouver tous les *nombres parfaits*, c'est-à-dire les nombres dont le double est égal à la somme de leurs diviseurs :

```
program parfaits;

const N = 10000;
var k : integer;

function sdiv(n : integer) : integer;
var i : integer;
begin
   sdiv := 0;
   i := 1;
   while i <= n do begin
      if n mod i = 0 then sdiv := sdiv + i;
      i := i + 1
   end
end; {range}

begin
   for k := 2 to N do
      if sdiv(k)=2*k then writeln(k);
   readln
end.
```

C. Passage des paramètres

Comme on l'a vu plus haut, on peut passer les arguments d'une procédure de deux façons différentes[8], selon qu'on fait précéder le nom de la variable dans la déclaration du mot-clé `var` ou pas. Cela modifie la façon dont la fonction va influer sur ce paramètre.

Paramètres transmis par valeur : lorsque le nom de la variable n'est pas précédé de `var`, l'argument est transmis *par valeur*. Cela signifie que la procédure peut utiliser la valeur de l'argument (et éventuellement la modifier dans le cours du calcul), mais qu'à la fin de la tâche, la variable conserve la valeur qu'elle avait avant. En pratique, au moment de l'appel, le compilateur crée une nouvelle variable, dans laquelle il copie la valeur de l'argument, et c'est cette nouvelle variable qui est utilisée à l'intérieur de la procédure.

Paramètres transmis par adresse : lorsque le nom de la variable est précédé de `var`, l'argument est transmis *par adresse*. Cela signifie que la procédure travaille avec la variable originale, et non pas une copie. Toute modification de la valeur de cette variable dans le corps de la procédure perdurera après la fin de son exécution.

Voici deux exemple permettant de comprendre ce qui se passe :

[8] Une procédure ou une fonction peut agir d'une autre façon sur les paramètres du programme : c'est en modifiant les variables *globales*, qui ne font pas partie de ses paramètres d'appel. Même si ce procédé est parfaitement valide, il entraîne de nombreuses complications dans la correction et la mise au point des programmes, et doit donc être évité autant que faire se peut. Pour bien faire, les seuls paramètres globaux qu'une fonction devrait avoir le droit d'utiliser sont les *constantes* globales.
La *programmation fonctionnelle* est un paradigme de programmation dans lequel on se refuse à toute modification de variables globales. Sans aller jusqu'à cet extrémisme, on s'efforcera de minimiser l'usage des variables globales, et surtout leur modification par des procédures en dehors des transmission par adresse.

Passage par valeur	Passage par adresse

Passage par valeur

```
program pass_valeur;

var x : integer;

procedure toto(u : integer);

begin
   u := 3;
   writeln('Pendant l''appel, x=',u);
end; { toto }

begin
   x := 2;
   writeln('Avant l''appel, x=',x);
   toto(x);
   writeln('Apres l''appel, x=',x);
   readln
end.
```

Passage par adresse

```
program pass_valeur;

var x : integer;

procedure toto(var u : integer);

begin
   u := 3;
   writeln('Pendant␣l''appel,␣x=',u);
end; { toto }

begin
   x := 2;
   writeln('Avant␣l''appel,␣x=',x);
   toto(x);
   writeln('Apres␣l''appel,␣x=',x);
   readln
end.
```

On voit que la seule différence entre ces deux programmes est la ligne d'en-tête de définition de la procédure `toto`. La différence se fait à l'exécution lors du troisième affichage : le programme de gauche montre que `x` vaut 2, celui de droite montre que `x` a changé de valeur et vaut maintenant 3.

TD 2 – Affectations, entrées-sorties

1) Ne pas perdre la balle des yeux

a) À l'issue de l'algorithme suivant :

```
A←1
B←A+1
A←B+2
B←A+2
A←B+3
B←A+3
```

Quelles sont les valeurs contenues dans les variables `A` et `B` ?

b) À l'issue de l'algorithme suivant :

```
N←1
S←N
N←N+1
S←S+N
N←N+1
S←S+N
N←N+1
S←S+N
N←N+1
S←S+N
```

Quelles sont les valeurs contenues dans les variables `N` et `S` ?

2) Périmètre et aire d'un rectangle

Écrire un programme demandant à l'utilisateur la longueur et la largeur d'un rectangle, et affichant son périmètre et son aire.

3) Conversion temporelle

Écrire un programme demandant à l'utilisateur un entier s et renvoyant le temps correspondant à s secondes, affiché sous la forme `hh:mm:ss`, `hh` étant le nombre d'heures, `mm` le nombre de minutes et `ss` le nombre de secondes.

4) Rendre la monnaie

On désire écrire un programme aidant à rendre la monnaie. On suppose qu'on dispose d'un nombre illimité de pièces de 1, 2, 5, 10, 20 et 50 centimes. Une somme S (en centimes) étant donnée, on veut la décomposer en le plus petit nombre possible de pièces de monnaie. Par exemple, si l'on doit rendre 93 centimes, le programme devra afficher :

```
Pour rendre 93 centimes, il faut donner :
- 1 pièce de 50 centimes,
- 2 pièces de 20 centimes,
- 0 pièces de 10 centimes,
- 0 pièces de 5 centimes,
- 1 pièces de 2 centimes,
- 1 pièces de 1 centime
```

Vous noterez que l'affichage n'est pas très satisfaisant : il y a des fautes d'orthographe, et des lignes inutiles. Si vous connaissez les conditionnelles, vous pouvez améliorer l'affichage en corrigeant ces défauts.

5) Échange de variables

a) Écrire un algorithme échangeant les contenus de deux variables `a` et `b`.

b) Écrire de même un algorithme échangeant les contenus de trois variables `a`, `b` et `c`, selon le schéma `a -> b -> c -> a`.

c) Le langage Python autorise les affectations “simultanées”, de la forme : `a,b = b,a`.
Que fait, selon vous, l’interprète Python lorsqu’il rencontre une telle ligne ? Cette syntaxe vous paraît-elle utile ? efficace ?

d) Que fait l’algorithme suivant :

```
a := a+b;
b := a–b;
a := a–b;
```

Quel est l’avantage de cet algorithme par rapport à celui de la première question ? Quel en est le principal inconvénient ?

e) Concevoir un algorithme similaire répondant à la deuxième question.

f) Travail personnel : démontrer qu’on ne peut pas échanger les contenus de deux variables en moins de trois instructions.

g) Par quelle opération pourrait-on remplacer l’addition si les variables à permuter étant de type booléen, ou caractère ?

h) Travail personnel : concevoir un algorithme effectuant une permutation circulaire sur n variables `a[1],...,a[n]`, envoyant `a[1]` sur `a[k+1]`, `a[2]` sur `a[k+2]`, etc.

Chapitre III

Les structures de contrôle

Sommaire

I Introduction 28
II Les conditionnelles 28
III Les boucles 29
TD 3 – Structures de contrôle 30

I. Introduction

Si l'on ne pouvait qu'indiquer à l'ordinateur une suite inamovible d'instructions à répéter pour accomplir sa tâche, on écrirait des algorithmes bien peu adaptables.

Par exemple, si l'on souhaite écrire un programme sachant résoudre une équation du premier degré $ax + b = 0$, on doit prévoir le cas où $a = 0$. Or nous ne savons pas encore comment réaliser cela.

De la même façon, si l'on veut calculer la somme des n premiers entiers, n étant donné, il va falloir effectuer un nombre variable d'addition, ce que nous ne savons pas non plus faire.

Les *structures de contrôle* sont là pour permettre à l'ordinateur d'accomplir des actions plus complexes. Ce chapitre va les étudier en détail.

II. Les conditionnelles

La structure la plus simple est la *conditionnelle*. On veut exécuter une action ou une autre selon qu'un test est vrai ou faux.

Par exemple, voici un programme résolvant une équation du premier degré $ax+b = 0$, en envisageant les différents cas possibles :

```
program premier_degre;

var a,b : real;
begin
   writeln('Resolution de l''equation ax+b=0');
   write('Valeur de a : ');
   readln(a);
   write('Valeur de b : ');
   readln(b);
   if a = 0.0
   then          (* equation 0x+b=0 *)
      if b = 0.0
      then       (* equation 0=0 *)
         writeln('Tous les reels sont solutions.')
      else       (* equation b=0 *)
         writeln('Aucun reel n''est solution.')
   else          (* cas general *)
      writeln('Une solution : ',-b/a)
end.
```

On voit dans le code de ce programme qu'on a envisagé l'ensemble des cas possibles : le cas où $a = 0$, qui se subdivise en deux sous-cas : $b = 0$ ou $b \neq 0$, et le cas général $a \neq 0$. Une seule des trois instructions `writeln` est exécutée à chaque lancement du programme.

La structure conditionnelle s'écrit sous deux formes :

Algorithmique	Traduction Pascal
Si <condition> Alors <action>	**if** <condition> **then** <action>;

Dans cette première forme, le bloc d'instruction `<action>` n'est exécuté que si `<condition>` est vrai. Dans le cas contraire, aucune instruction n'est exécutée, on passe directement à l'instruction suivante.

MCours.com

Algorithmique	Traduction Pascal
Si <condition> Alors <actionV> Sinon <actionF>	**if** <condition> **then** <actionV> **else** <actionF>;

Dans cette deuxième forme, l'un des deux blocs `actionV` ou `actionF` est exécuté, selon la valeur de vérité du booléen `<condition>`.

Notons que si l'un des blocs `action` contient plusieurs instructions, il doit être encadré par une paire `begin ... end`, de manière à être vu comme une seule instruction.

Un cas plus général de conditionnelle permet de traiter le cas d'une variable pouvant prendre plusieurs valeurs différentes (par exemple un caractère ou un chiffre lu au clavier) : l'instruction `case`. Je vous invite à en lire la documentation[1] si vous en avez un jour besoin.

III. Les boucles

1. Boucle itérative : for

La boucle `for` est la boucle la plus simple et la moins flexible. Elle permet de répéter une séquence d'instructions un nombre fixé de fois.

Algorithmique	Traduction Pascal
Pour <indice> de <depart> à <fin> faire <action>	**for** <indice> := <depart> **to** <fin> **do** <action>;

Ici, l'instruction (ou le bloc d'instruction) `<action>` est répétée pour toutes les valeurs de l'`indice` comprise (inclusivement) entre `depart` et `fin`. `indice` doit être d'un type ordonné, comme `integer`, `char` ou `boolean`[2], et augmente d'une unité à chaque passage dans la boucle.

Voici par exemple un programme demandant un entier n, et affichant la somme des entiers compris entre 1 et n :

```
program somme_entiers;

var i,n,s : integer;

begin
  write('Entrer␣l''entier␣n␣:␣');
  readln(n);
  s := 0;
  for i := 1 to n do      (* a l'etape i, s contient 1+2+...+(i-1) *)
    s := s + i;           (* a la sortie, s contient 1+2+...+i
*)
  writeln('La␣somme␣des␣entiers␣compris␣entre␣1␣et␣',n,'␣est␣:␣',s);
end.
```

Comme on peut le voir dans cet exemple, on peut utiliser la valeur de `indice` dans le bloc d'instruction `action` (ici réduit à une seule instruction.

2. Boucles conditionnelles : repeat et while

[1] http://wiki.freepascal.org/Case

[2] Hé oui, il est possible d'écrire quelque chose de la forme `for i from false to true do ...` !

TD 3 – Structures de contrôle

Conditionnelles

1) Valeur absolue

Écrire une fonction prenant en argument un réel et renvoyant sa valeur absolue.

2) Recherche d'extremum

a) Écrire une fonction `max2` renvoyant le maximum de deux entiers passés en paramètres.

b) Écrire une fonction `max3` renvoyant le maximum de trois entiers passés en paramètres.
Expliquer le déroulement de l'appel `max3(-5,6,8)` puis de l'appel `max3(7,-3,-9)`.

c) À l'aide d'un arbre, expliquer comment on peut concevoir une fonction `max4` renvoyant le maximum de quatre nombres. Combien de comparaisons sont-elles nécessaires ?

d) Écrire une fonction renvoyant le maximum *et* le minimum de trois entiers.
Il serait bon de faire un petit dessin expliquant ce qu'on sait après chaque comparaison.
Recommencer avec quatre entiers. Le petit dessin est encore plus nécessaire !

3) Valeurs distinctes

a) Écrire une fonction `compare` prenant en argument deux entiers `x` et `y`, et renvoyant `1` si `x>y`, `0` si `x=y` et `-1` si `x<y`.

b) Écrire une fonction `distincts` prenant en argument trois entiers, et renvoyant le nombre d'arguments distincts. Par exemple, `distincts(1,2,3)` doit renvoyer `3`, et `distincts(1,2,1)` doit renvoyer `2`.

c) Reprendre la question précédente avec quatre arguments au lieu de trois.

4) Tri d'un petit nombre de valeurs

5) Écrire une fonction `tri3` prenant en argument trois entiers, et écrivant à l'écran ces trois entiers dans l'ordre croissant.

6) Reprendre la question précédente avec quatre entiers au lieu de trois.

7) Équation du second degré

On cherche à écrire un programme résolvant les équations du second degré, de la forme $ax^2 + bx + c = 0$.

On rappelle que cette équation se résout en calculant son *discriminant* $\beta = b^2 - 4ac$.

- Si $\Delta > 0$, alors l'équation a deux solutions, $x_1 = \frac{-b - \sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b + \sqrt{\Delta}}{2a}$,
- si $\Delta = 0$, l'équation n'a qu'une seule solution, $x_0 = -\frac{b}{2a}$,
- enfin, si $\Delta < 0$, l'équation n'a pas de solution.

a) Écrire un premier programme ne tenant compte que du cas particulier où $a \neq 0$.

b) Corriger votre programme de manière à ce qu'il traite aussi le cas particulier $a = 0$.

Boucles

1) Compte à rebours

Écrire un programme demandant un entier positif n, et écrivant à l'écran un compte à rebours de n à 0. Par exemple, si l'on tape "3", le programme doit écrire :

Plus que 3 secondes...

Plus que 2 secondes...

Plus qu'1 seconde...

Boooooooommmmm !!!!!

- à l'aide d'une boucle indexée,
- à l'aide d'une boucle indexée *dont l'indice croit de* 0 *à* n,
- à l'aide d'une boucle `tant que`,
- à l'aide d'une boucle `répéter`.

2) Factorielle

Écrire trois programmes demandant à l'utilisateur un entier n, et calculant la factorielle $n!$ de n, définie par :

$$0! = 1, \quad \text{et} \quad n! = 1 \times 2 \times \cdots \times n \text{ pour } n \geqslant 1$$

- Le premier programme devra utiliser une boucle indexée,
- le deuxième une boucle `while`,
- le troisième une boucle `repeat`.

3) Somme des entiers impairs

a) Écrire un programme demandant à l'utilisateur un entier n, et calculant la somme des n premiers entiers impairs.

b) Lorsqu'on exécute ce programme pour différentes valeurs de n, on constate que pour certaines valeurs, le résultat est négatif. Quelle est la raison de cette bizarrerie ? Comment modifier votre programme pour calculer la somme des 1000 premiers entiers impairs ?

c) Modifier votre programme pour calculer la somme des n premiers carrés : $1^2 + 2^2 + \cdots + n^2$.

4) Choix d'une boucle

Dans les exercices précédent, une boucle semble-t-elle plus adaptée que les autres ?

5) Un milliard, c'est long ?

Implémentez une boucle de manière à faire compter votre ordinateur jusqu'à 10^6, puis 10^9 (n'affichez surtout pas chaque nombre, cela augmenterait singulièrement le temps d'exécution !). Attention à la représentation des entiers en machine.

Le temps nécessaire pour ces calculs vous semble-t-il conforme à ce que vous attendiez ?

6) Étude de suites

a) Écrire une fonction calculant la somme $\displaystyle\sum_{i=1}^{n} \frac{1}{1+i^2}$ en fonction de n.

b) On définit u_n par la formule : $u_n = \sqrt{1 + \sqrt{1 + \cdots + \sqrt{1 + \sqrt{1}}}}$, la formule contenant n radicaux.

Trouver une relation entre u_{n-1} et u_n, et en déduire une fonction calculant u_n en fonction de n.

7) Suites divergentes

a) Écrire une fonction prenant en argument un réel A, et calculant le premier entier N à partir duquel $\displaystyle\sum_{i=1}^{N} \frac{1}{i}$ devient strictement supérieur à A.

b) Écrire une fonction prenant en argument un entier k, et calculant le premier entier N à partir duquel $n!$ devient strictement supérieur à n^k.

c) Écrire une fonction prenant en argument un taux t, et calculant le temps au bout duquel un placement à intérêt composé de taux t aura doublé.

8) Racine carrée entière

Soit n un entier naturel. On cherche la *racine carrée entière* de n, c'est à dire le plus grand entier p tel que $p^2 \leqslant n$. On notera $\mathrm{r}(n)$ cet entier.

Par exemple, la racine carrée entière de 16 est 4, puisque $4^2 \leqslant 16$, et $5^2 > 16$. On retrouve ainsi dans le cas des carrés parfaits la racine carrée classique.

De même, $\mathrm{r}(233) = 15$, car $15^2 = 225 \leqslant 233 < 256 = 16^2$.

81) À l'aide de la seule définition, écrire une fonction `r` associant à un entier naturel n sa racine carrée entière.

   Combien de multiplications et de comparaisons sont-elles nécessaires pour calculer la racine carrée d'un entier n donné ?

82) Proposer un algorithme basé sur une recherche dichotomique. Cet algorithme est-il plus ou moins efficace que l'algorithme "naïf" ?

83) Héron, mathématicien grec du premier siècle après JC, a proposé l'algorithme suivant : partant de $x_0 = n$, on calcule les termes successifs d'une suite (x_n) par la relation de récurrence :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{n}{x_n}\right)$$

   Bien entendu, tous les calculs se font dans l'ensemble des entiers.

   Tel quel, cette relation ne permet pas de définir un algorithme, puisqu'il manque un critère d'arrêt. On arrêtera les calculs dès que deux termes successifs de la suite seront identiques.

   i. Expliquer pourquoi il est inutile de poursuivre les calculs après avoir obtenu deux valeurs identiques.
   ii. Implémenter cet algorithme sous la forme d'une fonction.
   iii. Tester sur quelques valeurs de n cet algorithme, et comparer son efficacité à celle de l'algorithme "naïf" et de l'algorithme dichotomique.

9) Logarithme entier

On souhaite programmer une fonction calculant $L_2(n)$, logarithme entier de base 2 de l'entier n. C'est le plus grand entier k tel que $2^k \leqslant n$. Ainsi, par définition :

$$2^{L_2(n)} \leqslant n < 2^{L_2(n)+1}$$

91) Écrire une version de cette fonction utilisant une boucle "tant que" implémentant mot pour mot la définition.

92) Écrire une version de cette fonction utilisant cette définition récursive de $L_2(n)$:

$$L_2(1) = 0, \quad \text{et} \quad L_2(n) = L_2(n/2) + 1 \text{ si } n > 1$$

   où bien entendu $N/2$ est le quotient *entier* de la division de n par 2.

93) On rappelle qu'en mathématiques, on définit $\log_2(n)$ sous la forme :

$$\log_2(n) = \frac{\ln n}{\ln 2}$$

   Utiliser cette définition, et le fait que $L_2(n)$ est la partie entière (*floor*, en anglais) de $\log_2(n)$, pour écrire une troisième version de cette fonction. Comparer les mérites respectifs de ces trois implémentations.

10) Écriture de tables de Pythagore

Le but de cet exercice et de faire un programme imprimant une table d'addition ou de multiplication

a) Écrire une fonction traçant à l'écran une table d'addition. On prendra bien garde à aligner correctement les résultats, et à les séparer avec des caractères "|" et "-".

   Indication : pour l'écartement des colonnes, utiliser les options de formatage de write et writeln, ou bien utiliser une fonction insérant un nombre correct d'espace avant d'afficher le résultat (il est bien entendu conseillé d'explorer les deux solutions).

b) Recommencer avec une table de multiplication.

11) Conjecture de Syracuse

On considère la suite (u_n) définie par la donnée de u_0 entier strictement positif, et de la relation de récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Par exemple, pour $u_0 = 7$, les termes successifs de la suite sont :

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
u_n	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	1	...

et à partir de cet instant, la suite ne prend plus que les trois valeurs 4, 2 et 1. On dit qu'elle est *ultimement périodique.*

La conjecture de Syracuse (ou conjecture de Collatz) affirme que quel que soit l'entier u_0, il existe un rang n tel que $u_n = 1$ (autrement dit, toute suite construite sur ce modèle est ultimement périodique sur le cycle $4 - 2 - 1$).

a) Écrire une fonction prenant en argument un entier a et affichant les termes successifs de la suite démarrant à $u_0 = a$ jusqu'à atteindre 1.

   L'utiliser pour vérifier la conjecture de Syracuse pour les valeurs de u_0 comprises entre 0 et N, N étant une borne donnant un temps de vérification raisonnable.

b) Le *temps de vol* de la suite est le plus petit entier n tel que $u_n = 1$. Pour notre exemple initial, il est de 16. Écrire une fonction prenant en argument le terme initial de la suite et renvoyant le temps de vol.

   Dresser une table de valeurs de ce temps de vol pour les valeurs de u_0 comprises entre 1 et N.

c) On s'intéresse aux *records de temps de vol.* Un entier a établit un record si la suite de terme initial a a un temps de vol plus long que toute suite de départ $b < a$.

   Écrire un programme recherchant les records de temps de vol jusqu'à une borne fixée à l'avance.

d) Le *temps de vol en altitude* est le plus petit entier n tel que $u_{n+1} < u_0$. Ainsi, le temps de vol en altitude de notre exemple est 10 puisque u_{11} est la première valeur de u_n inférieure à u_0.

   Écrire une fonction prenant en argument la valeur initiale de la suite, et renvoyant son temps de vol en altitude.

   Y a-t-il une corrélation entre le temps de vol et le temps de vol en altitude ?

e) *L'altitude maximale* est la plus grande valeur que prend la suite. Ainsi, l'altitude maximale de notre exemple est 52. Écrire une fonction calculant l'altitude maximale d'une suite connaissant sa valeur initiale.

   Écrire ensuite un programme calculant l'altitude maximale de toutes les suites telles que u_0 est compris entre 1 et N. Essayer d'optimiser ce programme de façon à ne pas refaire des calculs déjà effectués. Par exemple, si une suite de valeur initiale supérieure à 7 passe par la valeur 7, on peut facilement calculer son altitude maximale sans calculer d'autres termes de cette suite.

Chapitre IV

Les tableaux et les chaînes de caractères

Sommaire

I Les tableaux à une dimension : vecteurs 36
A Notion de tableau 36
B Exploration d'un tableau 36
II Tableaux à deux dimensions : matrices 37
III Les chaînes de caractères 38
TD 4 – Tableaux 39
TD 5 – Algorithmes de tri 42
TD 6 – Chaînes de caractères 43

I. Les tableaux à une dimension : vecteurs

A. Notion de tableau

Jusqu'à présent, nous avons manipulé quelques données, dont les valeurs successives ont été rangées dans des cases mémoires auxquelles nous avons donné des noms, comme `a`, `x`, `indice`...

Si l'on doit manipuler un grand nombre de données, comme un fichier de clients, cette méthode devient impraticable. L'idée consiste alors, lorsque les données sont de nature similaire, à les ranger dans une structure de données linéaire appelée *tableau*, ou *array* en anglais. On donne un nom au groupe d'objets, et on attribue à chaque objet un numéro d'ordre.

Dans la plupart des langages, il est nécessaire de déclarer un tableau avant de l'utiliser, et en tout cas de l'initialiser. Cela permet au système d'exploitation de réserver de la place pour les éléments.

Opération	Traduction Pascal
déclaration	**var** t : **array**[<i>..<j>] **of** <**type**>;
affectation	t[<k>] := <valeur>
valeur	t[<k>]

La première instruction déclare un tableau nommé `t` dont les indices des éléments sont compris entre `i` et `j` (ce tableau comporte donc $j - i + 1$ éléments), de type `type`. `i` et `j` peuvent être deux entiers, ou deux éléments d'un type énuméré.

La deuxième instruction permet d'affecter la valeur `valeur` à la case numéro `k` du tableau `t`.

Enfin, la dernière ligne montre comment utiliser la valeur stockée dans la case d'indice `k` du tableau `t`.

B. Exploration d'un tableau

Un tableau est une *structure énumérative*[1], qu'on peut voir comme une fonction de l'ensemble *ordonné* des indices dans l'ensemble du type des données stockées. Un certain nombre de méthodes sont utilisées de façon récurrente pour manipuler une telle structure de donnée.

1. Exploration de l'ensemble des éléments du tableau

Si l'on doit réaliser un traitement sur l'ensemble des éléments du tableau, le plus simple est d'utiliser une boucle indexée. Voici par exemple une fonction calculant la somme des éléments d'un tableau d'entier, de type `tableau = array[a..b] of integer` :

```
function recherche_tableau(t : tableau) : boolean;

var i : integer;

begin
   somme_tableau := 0;
   for i := a to b do
      somme_tableau := somme_tableau + t[i]
end;
```

La "variable" `somme_tableau` joue le rôle classique d'*accumulateur*, elle est initialisée à zéro , et on accumule à chaque itération de la boucle l'élément rencontré dans le tableau.

2. Recherche d'éléments dans un tableau

Si l'on cherche à vérifier la présence d'un élément particulier dans un tableau, le traitement peut s'arrêter dès que l'élément a été rencontré, ou bien à défaut quand on arrive à la fin du tableau. On

[1]par opposition à une *structure itérative* ; on accède de manière immédiate à n'importe quel élément de la première, alors qu'on ne peut accéder aux éléments de la seconde qu'après avoir lu tous leurs prédécesseurs. Les deux exemples de base de structures itératives sont les *listes* et les *fichiers*.

ne peut alors plus utiliser une boucle itérative, qui ne peut être interrompue avant la fin. Dans ce cas, on utilise une boucle conditionnelle. Voici un exemple d'une fonction recherchant un élément `x` dans un tableau de type similaire à celui que nous venons d'utiliser :

```
function recherche_tableau(t : tableau; x : integer) : boolean;

var i : integer;

begin
   recherche_tableau := false;
   i := a;
   while (i <= b) and (not recherche_tableau) do
      if t[i] = x
      then recherche_tableau := true
      else i := i+1
end;
```

On voit qu'ici dans le corps de la boucle on compare l'élément indexé à la valeur recherchée. Si il y a coïncidence, on change la valeur de `recherche_tableau`, de sorte que la condition d'entrée dans la boucle devient `false`, sinon on incrémente l'indice de boucle.

Nous rencontrerons d'autres méthodes en TD. En particulier, si le tableau est *trié* (ce qui est une hypothèse assez courante dans la pratique), une méthode de recherche très efficace peut être employée : la *recherche dichotomique*. Nous étudierons cette méthode très importante en TD.

II. Tableaux à deux dimensions : matrices

Il est parfois utile de repérer des éléments d'une collection d'objets à l'aide de deux nombres. Par exemple, des coordonnées dans le plan. Les matrices vues en cours de mathématiques sont un exemple abstrait de tableaux à deux dimensions.

On peut déclarer une matrice de deux façons différentes :

- soit comme un tableau à une dimension dont les éléments sont des tableaux à une dimension :

  ```
  var t : array[a..b] of array[c..d] of <type>;
  ```

  auquel cas `t[i][j]` est un élément de type `type`, et `t[i]` un tableau de type `array[c,d] of <type>`,

- soit comme un tableau à deux dimensions, de type

  ```
  var t : array[a..b,c..d] of <type>;
  ```

  auquel cas on accède à un élément de `t` sous la forme `t[i,j]`, `t[i]` étant encore un tableau.

La manière la plus simple d'effectuer un traitement sur l'ensemble des éléments d'une matrice consiste à inclure ce traitement dans une paire de boucles imbriquées. Voici par exemple une fonction calculant la somme des éléments d'un tableau de type `tableau = array[1..N,1..M] of integer` :

```
function somme_matrice(t : tableau) : integer;

var i,j : integer;

begin
   somme_matrice := 0;
   for i := 1 to N do
      for j := 1 to M do
         somme_matrice := somme_matrice + t[i,j]
end;
```

On peut imaginer d'autres façons d'explorer les cases d'une matrice, et nous en verrons des exemples en TD.

III. Les chaînes de caractères

Une chaîne de caractère peut être considérée comme un tableau de caractères. Et d'ailleurs, le Pascal standard ne disposait pas d'un type spécifique. Le Turbo-Pascal et ses successeurs, ainsi que l'immense majorité des langages de programmation, ont ajouté un type spécifique permettant de manipuler ces chaînes d'usage très courant, et certains langages se sont spécialisés dans ce type de traitement, comme le langage Perl.

Une déclaration de variable chaîne se présente sous la forme suivante :

```
var <nom> : string[<longueur>];
```

La variable `nom` est déclarée comme étant de type `string`, `longueur` indique la taille *maximale* de cette chaîne. La représentation en mémoire ajoute un octet pour stocker la longueur réelle de la chaîne. C'est ce nombre de caractère qui sera affiché par une instruction `write` ou `writeln`. Si cette longueur n'est pas précisée, une longueur par défaut de 255 caractères est appliquée.

Il y a peut d'opération directement réalisables sur les chaînes de caractère. Si `s` et `s'` sont des chaînes de type `string[N]`, voici ce qu'on peut faire :

Opérateur	Traduction Algorithmique/Pascal
accès à un caractère	`s[i]`
longueur effective	`length(s)`
concaténation	`s+s'`

Le caractère `s[i]` est le i-ème caractère de la chaîne (le premier a pour numéro 1). Le résultat est de type `char`, mais il peut être affecté à une chaîne.

Concaténer deux chaînes consiste à ajouter la deuxième au bout de la première. Si la chaîne résultante est trop longue, elle est simplement *tronquée*.

Par exemple, on aurait pu écrire le programme "Hello World" de la façon suivante :

```
program hello_chaine;

var s1,s2,s3 : string;

begin
   s1 := 'Hello';
   s2 := ', ';
   s3 := 'World';
   writeln(s1+s2+s3)
end.
```

TD 4 – Tableaux

Vecteurs

1) Initialisation d'un tableau

a) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et le remplissant de zéros.

b) Écrire une fonction prenant en argument un tableau `t` de type `array[1..N] of integer`, et le remplissant avec les valeurs d'une fonction `f` déclarée ailleurs dans le code. La case `t[i]` doit contenir la valeur `f(i)`.

c) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et le remplissant de valeurs aléatoires.

Indication : on rappelle que `random(m)` renvoie un entier pseudo-aléatoire compris entre 0 et m.

2) Impression d'un tableau

a) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et l'affichant sur la console. Le tableau devra apparaître sous la forme

```
[ 1; 2; 5; 3; 7; 18; 1]
```

b) Modifier cette fonction de manière à ce qu'elle n'imprime que les 20 premiers éléments du tableau, suivis de points de suspension si `N` est plus grand que 20.

3) Moyennes

a) Écrire une fonction `moyenne` prenant en argument un tableau de réels, et renvoyant la moyenne de ses éléments.

b) Écrire une fonction `moyenne_elaguee` prenant en argument un tableau de réels, et renvoyant de même la moyenne de ses éléments, *élaguée du plus petit et du plus grand élément du tableau.*

Par exemple, la moyenne élaguée du tableau `[12.0; 14.5; 8.1; 15.3]` est `13.25=(12+14.5)/2`, car `8.1` et `15.3` sont enlevés de la série.

4) Appliquer une fonction aux éléments d'un tableau

a) Écrire une fonction prenant en argument un tableau d'entier `t` de type `array[1..N] of integer`, et augmentant tous ses éléments de 1.

b) Modifier votre fonction de manière à ce qu'elle remplace chaque élément de `t` par son image par la fonction `f` définie autre part dans le code.

5) Ce tableau est-il trié ?

a) Écrire une fonction `croissant` prenant en argument un tableau de type `array[1..N] of integer`, et renvoyant vrai si et seulement si le tableau est trié en ordre croissant.

b) Modifier cette fonction de manière à ce qu'elle renvoie 0 si le tableau n'est pas trié, 1 si le tableau est trié en ordre croissant, et -1 s'il est trié en ordre décroissant.

Attention au fait que les éléments du tableau ne sont pas nécessairement tous distincts.

6) Miroir d'un tableau

a) Écrire une fonction `miroir` prenant en argument un tableau de type `array[1..N] of integer`, et en renvoyant l'*image miroir*, en échangeant `t[1]` et `t[N]`, `t[2]` avec `t[N-1]`, etc.

b) Modifier votre fonction de manière à ce qu'elle prenne deux arguments supplémentaires `u` et `v`, et qu'elle retourne le tableau `t` après avoir retourné la partie constituée des case d'indices compris entre `u` et `v` (inclus). Cette fonction sera utilisée plus bas.

7) Avant de trier, on mélange, et on se fait un petit poker !

On souhaite réaliser une procédure qui mélange aléatoirement les éléments d'un tableau donné en paramètre (par exemple pour simuler le mélange d'un jeu de carte).

a) Écrire une procédure prenant en argument un tableau d'entiers de type `array[1..N] of integer` et deux entiers `i` et `j`, et échangeant les éléments d'indices `i` et `j` du tableau.
En déduire une première procédure de mélange.

b) On souhaite simuler la méthode de mélange qu'on voit souvent dans les films : on coupe le jeu en deux, et on insère les cartes d'un des deux paquets entre les cartes de l'autre.
Pour cela, on va composer un nouveau tableau, et le remplir en prenant aléatoirement des éléments de la première ou de la seconde partie du tableau initial. Écrire la procédure mettant en oeuvre cette idée.

c) Utiliser ces deux méthodes de mélange pour simuler un grand nombre de distribution de cinq cartes d'un jeu de 52 cartes, et calculant la fréquence des figures usuelles du poker : une paire, deux paires, une couleur...

8) Recherche dans un tableau

a) Écrire une fonction prenant en argument un tableau d'entiers `t`, de type `array[1..N] of integer`, et un entier `x`, et renvoyant le nombre d'occurrences de `x` dans `t`.

b) Écrire une fonction prenant les mêmes arguments, et renvoyant 0 si `x` ne se trouve pas dans `t`, et l'indice de la première position (on dit *occurrence*) de `x` dans `t` dans le cas contraire.

9) Crible d'Erathostene

Matrices

1) Initialisation d'un tableau à deux dimensions

a) Écrire une fonction prenant en argument une matrice de type `array[1..N,1..M] of integer`, et initialisant l'ensemble de ses cases à 0.

b) Modifier cette fonction pour qu'elle remplisse le tableau d'entiers positifs aléatoires.

2) Moyenne des éléments d'une matrice

Écrire une fonction calculant la moyenne des termes d'une matrice de type `array[1..N,1..M] of real`.

3)

a)

Des exercices plus compliqués

1) Rotation d'un tableau

On veut écrire une fonction `rotation` prenant en argument un tableau de type `array[1..N] of integer`, et un entier k, et renvoyant la *rotation* d'ordre k de ce tableau : `t[1]` va en `t[k+1]`, `t[2]` et `t[k+2]`,... `t[N-k-1]` en `t[N]`, `t[N-k]` en `t[1]`,... et `t[N]` en `t[k]`.

a) Une première idée consiste à utiliser un tableau auxiliaire pour recopier le premier tableau, avant de faire une boucle pour remplir le tableau à modifier. Implémenter cette idée.

b) Une deuxième idée, plus difficile à mettre en oeuvre, consiste à sauvegarder `t[1]`, placer `t[k+1]` dans `t[1]`, puis `t[2k+1]` dans `t[k+1]`, et ainsi de suite.
Si tout se passe bien (quand est-ce le cas ?), l'algorithme se termine en une seule passe. Dans le cas contraire, il faut poursuivre par la première case qui n'a pas encore été modifiée, et ainsi de suite.
Implémenter cette idée.

c) Une idée beaucoup plus simple, mais plus astucieuse, consiste à constater qu'on peut se ramener à des images miroirs. En effet, si la structure du tableau est | A | B |, on peut obtenir le résultat final | B | A | en prenant le miroir de la partie A, puis le miroir de la partie B, et enfin le miroir du tableau obtenu. Faire un petit dessin pour vérifier ceci, et implémenter cette idée à l'aide de la fonction `miroir` construite plus haut.

2) **Recherche dans un tableau, version ultra-rapide**

Lorsque le tableau dans lequel on effectue une recherche est trié (en ordre croissant), on peut accélérer grandement la recherche en utilisant la méthode de *dichotomie*. Elle s'appuie sur le pseudo-algorithme suivant :

- au départ, l'élément `x` cherché doit se trouver dans le tableau `t[1..N]` ; on initialise `u` à `1` et `v` à `N` ;
- (B) si `u>v`, alors `x` ne se trouve pas dans le tableau ; on retourne `0` ou `false`, selon ce qui nous intéresse ;

  sinon, soit `m=(u+v)/2` ; on compare `x` à `t[m]` :
  - si `x=t[m]`, on retourne `true`, ou la valeur de `m`
  - si `x<t[m]`, `x` ne peut se trouver que dans la partie `t[u,m-1]` du tableau ; on pose `v=m-1`, et on retourne en (B) ;
  - sinon (i.e. si `x>t[m]`), `x` ne peut se trouver que dans la partie `t[m+1,v]` du tableau ; on pose `u=m+1`, et on retourne en (B).

a)

TD 5 – Algorithmes de tri

Algorithmes simples

1) **Tri par insertion**

   a)

2) **Tri à bulle**

   a)

3)

   a)

Algorithmes plus efficaces (mais plus complexes !)

1) **Tri rapide**

   a)

2) **Tri rapide "stochastique"**

   a)

3)

   a)

TD 6 – Chaînes de caractères

1)

a)

Chapitre V

La récursivité

Sommaire

I Un premier exemple 46
II Le principe de la récursivité 46
TD 7 – Récursivité 47

I. Un premier exemple

II. Le principe de la récursivité

TD 7 – Récursivité

Des exercices d'introduction

1)

a)

Des exercices plus complexes

1)

a)

MCours.com