STAF2X : INTRODUCTION A L'ALGORITHMIQUE

1. INTRODUCTION

1.1. Buts du cours

- Prendre conscience de l'importance de l'algorithmique en programmation,
- Acquérir de bons réflexes face à un problème à résoudre.
- La démarche générale est très systématique :
  - spécifier un problème *précisément* et *a priori*;
  - décrire un algorithme, éventuellement par raffinements successifs;
  - analyser l'algorithme puis le comparer à d'autres qui résolvent le même problème.

Pourquoi apprendre l'algorithmique pour apprendre à programmer ?
=> Parce que l'algorithmique exprime les instructions résolvant un problème donné **indépendamment des particularités de tel ou tel langage**.

1.2. Notion d'algorithme

Dans la vie courante, un algorithme peut pex prendre la forme :

- d'une recette de cuisine,
- d'un mode d'emploi,
- d'une notice de montage,
- d'une partition musicale,
- d'un itinéraire routier qu'on explique à un touriste perdu,
- ….

1.3. Historique

Un algorithme est la «*spécification d'un schéma de calcul, sous forme d'une suite [finie] d'opérations élémentaires obéissant à un enchaînement déterminé*» (Encyclopedia Universalis). Cependant le terme d'*algorithme* ne concerne pas que l'informatique, et la notion d'algorithme a précédé celle d'ordinateur.

On connaît depuis l'antiquité des algorithmes sur les nombres
=> pex l'algorithme d'Euclide qui permet de calculer le PGDC de 2 nombres entiers (exercice plus tard).

Pour traiter l'information, on a développé des algorithmes opérant sur des données non numériques

- les algorithmes de tri : permettent pex de ranger par ordre alphabétique une suite de noms,
- les algorithmes de recherche d'une chaîne de caractères dans un texte
- les algorithmes d'ordonnancement, qui permettent de décrire la coordination entre différentes tâches, nécessaire pour mener à bien un projet.

1.4. Définition(s) d'algorithme

- résolution en un certain nombre d'étapes d'un problème clairement défini.

- indépendant du langage de programmation utilisé.

- suite d'instructions, qui exécutée correctement, conduit à un résultat donné et voulu .

- Un algorithme décrit un traitement sur un certain nombre, fini, de données. C'est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est:

  - définie de façon rigoureuse et non ambiguë;

  - effective, ie pouvant être effectivement réalisée par une machine
    => pex la division entière est une opération effective, mais pas la division avec un nombre infini de décimales.

- Un programme est généralement la description d'un algorithme dans un langage accepté par l'ordinateur.
- Un algorithme informatique se ramène toujours à la combinaison de 4 briques de base :
    - l'affectation de variables
    - la lecture / écriture
    - les tests
    - les boucles

- quelques briques ou plusieurs centaines de milliers.

- Taille ≠ complexité : de longs algorithmes peuvent être assez simples, et de petits très compliqués.

1.5. Faut-il être matheux pour être bon en algorithmique ?

Non, l'algorithmique demande 2 qualités :

- ***Rigueur***

  - chaque fois qu'on écrit une série d'instructions, il faut se mettre à la place de la machine qui va les exécuter, pour vérifier si le résultat obtenu est bien celui escompté.

- ***Intuition***

  - aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu.
  - Alors on est plus ou moins intuitifs, mais les réflexes, cela s'acquiert, et l'expérience finit par compenser largement des intuitions.

2. ALGORITHME DE BASE

2.1. Caractéristiques d'un algorithme

Pour écrire un algorithme : pseudo-code
- série de conventions,
- ressemble à un langage authentique,
- la plupart des problèmes de syntaxe sont mis de côté.

Un bon algorithme doit être :
- déterministe :
  - toute exécution d'un algorithme sur les mêmes données donne lieu à la même suite d'opérations.
- Systématique
- bien structuré (et si possible simple à comprendre)
- non ambiguë
- juste
- éventuellement élégant

- lisible :
  - Retour à la ligne
  - Indentation
  - Commentaires
    - ne paraphrasent pas le programme
    - courts
    - se limitent à l'explication des idées essentielles ou des points délicats.
- pas trop long (décomposition en modules);
- éviter les branchements
- efficace. Quelques critères d'efficacité :
  - Temps d'exécution
  - Quantité d'espace mémoire
  - Quantité d'espace disque
  - Quantité d'information à lire/écrire
  - Quantité d'information à transférer
  - ….
- préférer parfois la récursivité (chap 11.2)

2.2.Modularité

- La résolution d'un problème (= module) doit être découpée en sous-problèmes (= sous-modules) plus petits et plus simples à résoudre :

  - c'est plus clair,
  - les sous-modules sont plus facilement réutilisables,
  - les sous-modules peuvent devoir être utilisés plusieurs fois par module

  => on appelle le sous-module à chaque fois que c'est nécessaire.

- Découper l'algorithme en modules aussi indépendants que possible :

  =>l'interface des différents modules avec l'extérieur doit être décrite précisément :
  - variables d'entrée-sorties,
  - variables globales utilisées et/ou modifiées.

  => les liaisons enter les différents modules doivent être clairs :
  - quel module utilise quel autre module,

- quels modules partagent une même variable globale.

- le rôle de chaque module doit être explicité clairement,
  => soit sous forme de formules,
  => soit sous forme d'une phrase en langage naturel: on donne la spécification de ce module.

2.3. Spécification d'un algorithme

- décrit ce que l'algorithme fait, sans détailler comment (rôle des commentaires souvent).
- concis, au risque d'être imprécis
- en italique.

Attention : la comparaison d'algorithmes n'a de sens que si les spécifications sont les mêmes.

2.4 .Langage algorithmique

- permet la description de la résolution d'un problème en utilisant des opérations et des enchaînements d'opérations qui sont ceux des ordinateurs sans toutefois être particulier à un ordinateur précis.

- Un algorithme peut d'abord être écrit en langage « parlé » décrivant la succession des opérations qui doivent être faites.
  - Puis, par *raffinage successif* se transformer en langage algorithmique (*pseudo-code*) qui se trouve à mi chemin entre le parlé et le code.

2.5. Patron d'un algorithme

```
Algorithme NomAlgorithme
Début
  ... actions
Fin
```

- *profil* (donne le nom de l'algorithme)
- *délimiteur de début*
- *les différentes actions*
- *délimiteur de fin.*

Ainsi, l'algorithme suivant est valide :

```
Algorithme Bonjour
Début
  Afficher('Salut Iris')
  ALaLigne
Fin
```

3. DU PROBLEME A SON EXECUTION

3.1. Cycle de développement

Le cycle de développement d'un "programme (ou d'une application) informatique " peut se résumer ainsi :

Problème --> Analyse --> Algorithme --> Programme --> Compilation --> Exécution

Problème --> Analyse --> Algorithme --> Programme --> Compilation --> Exécution

- Problème :

  Ex : donner le plus court chemin dans le métro entre 2 stations

- Analyse :
  - phase de réflexion
    - permet l'identification des caractéristiques du Problème puis
    - permet de découper le problème en une succession de tâches simples et distinctes.

  Ex :
  - on identifie des données importantes (le temps de parcours entre 2 stations et le temps de changement de ligne),
  - on élabore des stratégies comme : ne pas passer 2 fois par la même station, ...

Problème --> Analyse --> Algorithme --> Programme --> Compilation --> Exécution

- Algorithme :
  - description des opérations à mettre en oeuvre (en langage algorithmique) expliquant comment obtenir un résultat à partir de données.
  - description compréhensible par un être humain de la suite des opérations à effectuer pour résoudre le Problème.
  - Néanmoins, ce langage algorithmique est suffisamment proche des langages de programmation pour pouvoir être traduit aisément vers ces derniers.

Ex :

- démarrer par la station de départ,
- rechercher toutes les stations voisines accessibles de cette station
- parmi cette liste, si la station d'arrivée figure dans la liste alors .....
- sinon rechercher toutes les stations voisines des voisines .....
- .....

Problème --> Analyse --> Algorithme --> Programme --> Compilation --> Exécution

- Dans la réflexion, on peut précéder l'étape de l'algorithme par celle de la représentation graphique du squelette de l'application : c'est ***l'algorithme fonctionnel***.
  - permet de comprendre d'un seul coup d'œil ce qui se passe.
  - se situe à un niveau plus général, plus abstrait, que l'algorithme normal,
  - Dans la construction et la compréhension d'une application, les 2 documents peuvent être complémentaires, et constituer 2 étapes successives de l'élaboration du projet.

- Programme :
  - fichier résultant de la traduction de l'algorithme dans un langage de programmation.
  - fichier texte des instructions et de leurs enchaînements
  - un ou plusieurs algorithmes utilisés.

3.2. Des problèmes sans solution

Tous les problèmes ne se résolvent pas grâce à un algorithme. 2 cas :

1/ Complexité algorithmique exponentielle :

- algorithmes qui ne permettent pas de traitement du problème dans un temps raisonnable

- ressources nécessaires à leur exécution en temps et en mémoire trop importante

- Ex : le jeu d'échec :
  - Possible de faire un programme calculant toutes les conséquences de tous les coups possibles => meilleur que tout joueur humain.
  - MAIS Il faudrait considérer de l'ordre de 10^{19} coups possibles pour décider de chaque déplacement. (10^{19} ms est de l'ordre de 300 millions d'années).
  - Complexité trop importante => pas envisageable de mettre un tel algorithme en pratique.

2/ Indécidabilité :

- parfois il n'existe aucun algorithme pour certains problèmes.

- Ex : le paradoxe du barbier :

  - dans une ville où les gens soit se rasent eux-mêmes, soit se font raser par le barbier, qui rase le barbier ?
  - Soit il se rase lui-même et, dans ce cas, il n'est pas rasé par le barbier,
  - Soit il ne se rase pas lui même et il est donc rasé par le barbier.
  - Donc : soit il se rase lui-même et donc il ne se rase pas lui-même, soit il ne se rase pas lui-même et donc il se rase lui-même.

=> On doit donc conclure qu'une telle ville avec de tels habitants ne peut exister et, de la même manière, l'algorithme n'existe pas non plus.

4. LES BOUCLES

Une des 4 structures de base des algorithmes : les boucles (ou structures répétitives, ou structures itératives).

Ex :

- on pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non).
- Mais l'utilisateur risque de taper autre chose.
- Dès lors, le programme peut planter ou produire des résultats fantaisistes.

=> Alors, on peut mettre en place un **contrôle de saisie** (pour vérifier que les données entrées correspondent bien à celles attendues par l'algorithme).

On peut faire cela avec une boucle SI :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
Si Rep <> "O" ET Rep <> "N" Alors
Ecrire "Saisie erronnée. Recommencez"
Lire Rep
FinSi
```

- Si l'utilisateur ne se trompe qu'une seule fois => OK
- Mais pour prévoir le cas de deuxième erreur, il faut rajouter un SI.
- Et ainsi de suite….

=> impasse…

4.1. Boucle « Tant que »

La seule issue est donc une structure de boucle, qui se présente ainsi :

```
Algorithme JusquAuMur
Début
  Tant que Non(DevantMur) faire
    Avancer
  Fin tant que
Fin
```

Le principe:

1. le programme arrive sur la ligne du TantQue.
2. Il examine alors la valeur du booléen (variable booléenne ou une condition).
3. Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue.
4. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite.
5. Ça ne s'arrête que lorsque le booléen prend la valeur FAUX.

Illustration avec notre problème de contrôle de saisie :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" ET Rep <> "N"
  Lire Rep
  Si Rep <> "O" ET Rep <> "N" Alors
    Ecrire "Saisie erronnée. Recommencez"
  FinSi
FinTantQue
```

Il existe d'autres boucles….

4.2. Boucle « Si… Alors…Sinon… »

```
Algorithme QueFaireCeSoir
Début
  Si Pluie
  Alors
    MangerPlateauTélé
    SeCoucherTot
  Sinon
    MangerAuRestaurant
    AllerAuCinema
  Fin si
Fin
```

4.3. Boucle « Répéter »

```
Algorithme JusquAuMurVersionRépéter
Début
  Répéter
    Avancer
  jusqu'à DevantMur
Fin
```

- contrairement à la boucle *tant que*, on passe toujours au moins une fois dans une boucle *répéter*.
- Ainsi, dans l'ex ci-dessus, on avancera forcément d'un case... il conviendrait donc de tester si l'on n'est pas devant un mur avant d'utiliser cet algorithme.

4.4. Variables et types

(en détail au chap 6)

- Une variable est constituée d'un nom et d'un contenu (d'un certain type).

- Les types différents : booléen, caractère, chaîne de caractères, nombre entier, nombre réel, etc.

- Une déclaration de variables et de types présente plusieurs intérêts :
  - contrôle de type
  - précision sur le profil d'un algorithme

4.5. Boucle « Pour »

- Avec variables, on peut écrire la boucle *pour*.
- Lorsque l'on sait exactement combien de fois on doit itérer un traitement, c'est cette boucle qui doit être privilégiée.

```
Algorithme CompteJusqueCent
Début
  Pour i allant de 1 à 100 faire
    Afficher(i)
    ALaLigne
  Fin Pour
Fin
```

4.6. Des boucles dans des boucles

Une boucle peut contenir d'autres boucles.

```
Variables Truc, Trac en Entier
Pour Truc ⇦ 1 à 15
  Ecrire "Il est passé par ici"
    Pour Trac ⇦ 1 à 6
      Ecrire "Il repassera par là"
    Trac Suivant
Truc Suivant
```

Dans cet ex, le programme écrira 1x "il est passé par ici" puis 6x de suite "il repassera par là", et ceci 15x en tout. A la fin, il y aura donc eu 15 x 6 = 90 passages dans la 2^{e} boucle (celle du milieu).

Des boucles peuvent être imbriquées ou successives, mais elles ne peuvent jamais être croisée => aucun sens logique.

4.7. Comparaisons des boucles pour un problème simple

Prenons un même problème (Affichage des 100 premiers nombres entiers) dont on écrit plusieurs algorithmes différents, en changeant la boucle utilisée :

boucle « *pour* »

```
Algorithme CompteJusqueCentVersionPour
Variable i : entier
Début
  Pour i allant de 1 à 100 faire
    Afficher(i)
    ALaLigne
  Fin Pour
Fin
```

boucle « *tant que* »

(il faut initialiser *i* avant la boucle, et l'augmenter de 1 à chaque passage) :

```
Algorithme CompteJusqueCentVersionTQ
Variable i : entier
Début
  i=1
  Tant que (i <= 100) faire
    Afficher(i)
    ALaLigne
    i = i+1
  Fin tant que
Fin
```

boucle « *répéter* »

(noter que la condition d'arrêt est ici la négation de la condition du *tant que*):

```
Algorithme CompteJusqueCentVersionRepeter
Variable i : entier
Début
  i = 1
  Répéter
    Afficher(i)
    ALaLigne
    i=i+1
  Jusqu'à (i > 100)
Fin
```

Solution récursive également possible (cf chap 11.2)

5. EXERCICE : EXEMPLE DE L'ALGORITHME D'EUCLIDE

Ex : l'algorithme d'Euclide : *Trouver le pgdc de 2 nombres entiers (144, 96)*.

1. comprendre le problème et savoir le résoudre
2. Rédiger proprement la méthode:
3. Rédiger un algorithme
4. Ecrire un programme.

Outils à disposition : papier, crayon, cerveau

1. comprendre le problème et savoir le résoudre

Trouver le pgdc de 2 nombres entiers (144, 96)

- si a>b alors pgdc(a, b) = pgdc(a-b, b)
- pgdc(a,b)= pgdc(b,a)
- pgdc(a,a)= a
- (144,96) => (48,96) => (96,48) => (48,48) => pgdc=48

2. Rédiger proprement la méthode:

- pour calculer pgdc de a et b :

Tant que a est différent de b exécuter les 2 lignes suivantes :
Si a > b, retrancher b à a
Sinon, retrancher a à b
Affirmer que le pgdc est a

3. Rédiger un algorithme

- pgdc(a,b : entiers strictements positifs) : entier

```
Répéter tant que a ≠ b
    Si a>b
        a ← a-b
    Sinon
        b←b-a
    renvoyer a et b
```

6. LES VARIABLES

6.1. Déclaration

- Dans un programme, on a besoin de stocker provisoirement des valeurs
    - o données issues du disque dur, frappées au clavier, obtenues par le programme, intermédiaires ou définitifs).
    - o Ces données peuvent être de plusieurs types : nombres, texte,…
- Dès que l'on a besoin de stocker une information dans un programme, on utilise une **variable**.
- Image : une variable est une boîte, repérée par une étiquette.
    - o Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

- Avant de pouvoir utiliser une variable il faut la créer et lui coller une étiquette

  - c'est la **déclaration des variables**.
  - se fait tout au début de l'algorithme,
  - avant même les instructions proprement dites.

- Le **nom** de la variable obéit à des impératifs dépendant du langage, mais en générale :

  - commence impérativement par une lettre.
  - peut comporter lettres et chiffres,
  - mais pas la plupart des signes de ponctuation (en particulier les espaces).

6.2. Type des variables

Une fois la boîte créée (emplacement mémoire réservé), il faut préciser ce que l'on veut mettre dedans, car cela influence :

- la taille de la boîte (de l'emplacement mémoire)

- le **type** de codage utilisé.

  - Type numérique
  - Type alphanumérique (type caractère)
  - Type booléen

6.2.1 Types numériques classiques

- variable destinée à recevoir des nombres (cas le plus fréquent)
- Le type de codage (= le type de variable) choisi pour un nombre va déterminer :
  - Les valeurs maximales et minimales des nombres pouvant être stockés dans la variable
  - la précision de ces nombres (dans le cas de nombres décimaux).

Tous les langages offrent plusieurs types numériques, généralement :

Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	-3,40E38 à -1,40E-45 pour les valeurs négatives 1,40E-45 à 3,40E38 pour les valeurs positives
Réel double	1,79E308 à -4,94E-324 pour les valeurs négatives 4,94E-324 à 1,79E308 pour les valeurs positives

Pourquoi ne pas déclarer toutes les variables numériques en réel double?

- A cause du principe de l'économie de moyens.
- Un bon algorithme marche tout en évitant de gaspiller les ressources de la machine.
- Sur certains programmes de grande taille, l'abus de variables surdimensionnées peut entraîner des ralentissements notables à l'exécution, voire un plantage pur et simple.

Une déclaration algorithmique de variables aura ainsi cette tête :

Variable g **en Entier Long** **Variables** PrixHT, TauxTVA, PrixTTC **en Réel Simple**

6.2.2 Types non numériques

type alphanumérique *(ou* ***type caractère****)*

- stocke des caractères (lettres, signes de ponctuation, espaces, ou chiffres)

- Une série de caractères s'appelle une **chaîne** de caractères.

    - toujours notée entre guillemets, parce que 423 peut représenter le nombre 423, ou la suite de caractères 4, 2, et 3, selon le type de variable qui a été utilisé pour le stocker. Les guillemets permettent d'éviter toute ambiguïté à ce sujet.

type booléen

- stocke uniquement les valeurs logiques VRAI et FAUX

6.3. L'instruction d'affectation

6.3.1 Syntaxe et signification

- L'affectation de variable (= attribution d'une valeur) est la seule chose qu'on puisse faire avec une variable.

- En algorithmique, cette instruction se note avec le signe ⇦.

  - Ex : Toto ⇦24
    - Attribue la valeur *24* à la variable *Toto* (<= *Toto* est une variable numérique).

- On peut attribuer à une variable la valeur d'une autre variable, telle quelle ou modifiée.
  - Ex : Tutu ⇦ Toto
    - Signifie que la valeur de Tutu est maintenant celle de Toto.
    - Ne modifie pas la valeur de Toto : une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.

- Questions à Iris :

  - Tutu ⇦ Toto + 4
    - si Toto contenait 12, Tutu vaut combien ?
    - et Toto maintenant ?

  - Tutu ⇦ Tutu + 1
    - Si Tutu valait 6, il vaut combien maintenant ?
    - La valeur de Tutu est modifiée, puisque Tutu est la variable située à gauche de la flèche.

6.3.2 Ordre des instructions

L'ordre dans lequel les instructions sont écrites joue un rôle essentiel dans le résultat final.

Considérons les 2 algorithmes suivants :

```
Variable A en Entier
Début
  A ⇦ 34
  A ⇦ 12
Fin
```

```
Variable A en Entier
Début
  A ⇦ 12
  A ⇦ 34
Fin
```

Question à Iris : quelle est la valeur finale de A dans le premier cas ?
et dans le second ?

6.3.3 Expressions

Dans une instruction d'affectation, on trouve :

- à gauche de la flèche : uniquement un nom de variable.

- à droite de la flèche : une **expression** :

    - ensemble de valeurs liées par des **opérateurs**,
    - son résultat final est du même type que la variable à gauche.

6.3.4. Opérateurs

- signe qui peut relier 2 valeurs pour produire un résultat.

- Les opérateurs possibles dépendent du type des valeurs qui sont en jeu.

- *opérateurs numériques* :
  - \+ addition
  - \- soustraction
  - * multiplication
  - / division
  - ^ " puissance "

- utilisation possible des parenthèses (mêmes règles qu'en math).
  - La multiplication et la division ont " naturellement " priorité sur l'addition.
  - Les parenthèses ne servent qu'à modifier cette priorité naturelle.
    - Ex : en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent 41.
    - mais 12 * (3 + 5) vaut 12 * 8 soit 96.

- *opérateur alphanumérique* (&) :
  - permettant de **concaténer** (=agglomérer) 2 chaînes de caractères

- *opérateurs logiques* (ET, OU et NON).
  - *(cf chap 8.3)*

6.3.5. Remarques

- En mathématiques, une " variable " est généralement une inconnue. En informatique, une variable a toujours une valeur et une seule. Les variables non encore affectées sont considérées comme valant zéro. Et cette valeur ne varie que lorsqu'elle est l'objet d'une instruction d'affectation.

- En algorithmique, le signe de l'affectation est le ⇦. Mais en pratique, la quasi totalité des langages emploient le signe égal. La confusion est facile avec les maths. Alors on dit souvent non pas « égal » mais « prend pour valeur ».

7. NOUVEAU PATRON D'UN ALGORITHME

Distinguons les paramètres externes et les variables internes à l'algorithme. Ainsi, le patron d'un algorithme devient

```
Algorithme NomAlgorithme (paramètres...)
Variable ...
Début
  ... actions
Fin
```

Par exemple,

```
Algorithme DessineEtoiles (n : entier)
Variable i : entier
Début
  Pour i allant de 1 a n faire
    Afficher('*')
  Fin pour
Fin
```

8. LES TESTS

Séries d'instructions que l'ordinateur doit effectuer selon que la situation se présente d'une manière ou d'une autre.

8.1. Structure d'un test

Que 2 formes possibles pour un test ; complète ou simplifiée.

Si booléen **Alors**
Instructions 1
Sinon
Instructions 2
Finsi

Si booléen **Alors**
Instructions
Finsi

Note : Un booléen est une expression dont la valeur est VRAI ou FAUX. Cela peut donc être :

- une variable de type booléen
- une condition (comparaison, cf chap 8.2)

Explications :

- arrivé à la première ligne (Si…Alors) la machine examine la valeur du booléen.
  - Si ce booléen a pour valeur VRAI, elle exécute la série d'instructions 1.
    - A la fin de cette série d'instructions, au moment où elle arrive au mot " Sinon ", la machine sautera directement à la première instruction située après le " Finsi ".
  - au cas où le booléen avait comme valeur " Faux ", la machine saute directement à la première ligne située après le " Sinon " et exécute l'ensemble des " instructions 2 ".

Si l'une des deux " branches " du Si est vide, plutôt qu'écrire " sinon ne rien faire du tout ", il est plus simple de ne rien écrire.

Exprimé sous forme de pseudo-code, la programmation d'un touriste perdu donnerait :

Exemple
Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation **Alors**
Tournez à droite
Avancez
Prenez la deuxième à gauche
Sinon
Continuez jusqu'à la prochaine rue à droite
Prenez cette rue
Prenez la première à droite
Fin si

8.2. Une condition

- Une condition est une comparaison.

- 3 éléments :
  - une valeur
  - un **opérateur de comparaison**
  - une autre valeur

- Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…)

- Les opérateurs de comparaison sont : = < > < > >= =<

- L'ensemble constitue donc une affirmation, qui a un moment donné est VRAIE ou FAUSSE.

- Les opérateurs de comparaison s'emploient aussi avec des caractères.
    - codés par la machine dans l'ordre alphabétique,
    - les majuscules placées avant les minuscules
        - "t" < "w" VRAI
        - "Maman" > "Papa" FAUX
        - "maman" > "Papa" VRAI
- Certains raccourcis du langage peuvent mener à des non-sens informatiques.
    - Pex : la condition " Toto est compris entre 5 et 8 ".
        - On peut être tenté de la traduire par : 5 < Toto < 8.
        - a du sens en mathématiques, mais ne veut rien dire en programmation.

        => condition composée

8.3. Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus.

- Pour " Toto est inclus entre 5 et 8 ", il y a en fait 2 conditions.
    - " Toto > 5 " et "Toto < 8 ".
    - reliées par un *opérateur logique*, le mot ET.

- 3 opérateurs logiques : ET, OU, et NON.

ET (même sens en informatique que dans le langage courant). Pour que :
C1 ET C2 soit VRAI, il faut que C1 soit VRAIE et que C2 soit VRAIE.

OU (Le OU informatique ne veut pas dire " ou bien "). Pour que :
C1 OU C2 soit VRAI, Il suffit que C1 soit VRAIE ou que C2 soit VRAIE.
De plus, si C1 est VRAIE et que Condition2 est VRAIE aussi, C1 OU C2 est VRAIE.

NON : inverse une condition :

Condition VRAI ⇔ NON (Condition) FAUX

NON (X > 15) revient à écrire X =< 15

On représente ceci dans des **tables de vérité** :

C1 ET C2		C1	
		VRAI	FAUX
C2	VRAI	VRAI	FAUX
	FAUX	FAUX	FAUX

C1 OU C2		C1	
		VRAI	FAUX
C2	VRAI	VRAI	VRAI
	FAUX	VRAI	FAUX

C1 XOR C2		C1	
		VRAI	FAUX
C2	VRAI	FAUX	VRAI
	FAUX	VRAI	FAUX

8.4. Tests imbriqués

SI peut ouvrir 2 voies, ou plus... Pex, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre 3 réponses possibles :

```
Exemple
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
Ecrire "C'est de la glace"
Sinon
Si Temp < 100 Alors
Ecrire "C'est du liquide"
Sinon
Ecrire "C'est de la vapeur"
Finsi
Finsi
Fin
```

- économies au niveau de la frappe du programme : au lieu de devoir taper 3 conditions, dont une composée, on a plus que 2 conditions simples.
  => plus simple et plus lisible

- économies sur le temps d'exécution de l'ordinateur. Si la première possibilité est la bonne, le programme passe directement à la fin sans tester le reste, forcément faux.
  => plus performant

8.5. Variables Booléennes

- Les variables booléennes stockent les valeurs VRAI ou FAUX.

- on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

On peut réécrire l'exemple de l'eau ainsi :

Exemple
Variable Temp **en Entier**
Variables A, B **en Booléen**
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
A ⇦ Temp =< 0
B ⇦ Temp < 100
Si A **Alors**
Ecrire "C'est de la glace"
Sinon
B **Alors**
Ecrire "C'est du liquide"
Sinon
Ecrire "C'est de la vapeur"
Finsi
Finsi
Fin

- A priori, cela n'a guère d'intérêt : on a alourdi l'algorithme de départ, en lui faisant recourir à 2 variables supplémentaires, mais :

- Une variable booléenne n'a besoin que d'un seul bit pour être stockée

  => alourdissement moindre.

- Dans le cas de conditions composées très lourdes (avec plein de ET et de OU)

  =>facilite le travail
  De programmation.

8.6. Jeux logiques

Dans le cas de conditions composées (ET + OU), les parenthèses jouent un rôle important.

```
Exemple
Variables A, B, C, D, E en Booléen
Variable X en Entier
Début
Lire X
A ⇦ X < 2
B ⇦ X > 12
C ⇦ X < 6
D ⇦ (A ET B) OU C
E ⇦ A ET (B OU C)
Ecrire D, E
Fin
```

Questions a IRIS : Si X = 3, que vaut D ? et que vaut E ?

- S'il n'y a que des ET ou que des OU, les parenthèses ne changent rien.

- Toute structure de test avec ET peut être exprimée de manière équivalente avec OU, et réciproquement.

- La règle d'équivalence :

Si A **ET** B **Alors**

Instructions 1

Sinon
Instructions 2

Finsi

Si NON A **OU NON** B **Alors**
Instructions 2
Sinon
Instructions 1
Finsi

9. LES TABLEAUX

- *Tableau* : ensemble de valeurs portant le même nom de variable et repérées par un nombre
- *Indice* : nombre qui sert à repérer chaque valeur.

Ex : pour un programme, nous avons besoin simultanément de 12 valeurs (pex des notes pour calculer une moyenne).

9.1. Notation et utilisation algorithmique.

- Dans notre ex, on crée donc un tableau appelé Note, chaque note individuelle sera désignée Note(1), Note(2),...

- Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra : **Tableau** Note(12) **en Entier**

- On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, mais aussi tableaux de caractères, de booléens.

- L'avantage des tableaux, c'est qu'on peut les traiter en faisant des boucles. Pex :

```
Tableau Note(12) en Entier
Variables i, Som en Entier
Variable Moy en Réel
...
  Som ⇦ 0
  Pour i ⇦ 1 à 12
    Som = Som + Note(i)
    i Suivant
  Moy = Som / 12
```

La valeur d'un indice doit toujours être:

- = **au moins à 0 ou à 1** (selon le langage, le premier élément d'un tableau porte l'indice zéro ou l'indice 1)
- **un nombre entier**
- =< **au nombre d'éléments du tableau**.

- Ne pas confondre **l'indice** d'un élément d'un tableau avec son **contenu**.
  - La 3^{e} maison de la rue n'a pas forcément 3 habitants.
  - En notation algorithmique, il n'y a aucun rapport entre i et truc(i).

9.2. Les Tableaux multidimensionnels

- tableaux à 2 dimensions : valeurs repérées par 2 coordonnées :

  - **Tableau** Cases(8, 8) **en Entier**

  => espace de mémoire pour 8 x 8 entiers

- Peu importe le nombre de dimensions, le principe reste le même.

  => Si je déclare un tableau Titi(3, 5, 4, 4), il contient 3 x 5 x 4 x 4 = 240 valeurs.

  Chaque valeur est repérée par 4 coordonnées.

10. LES FONCTIONS PREDEFINIES

- Certains traitements *ne peuvent pas* être effectués par un algorithme
    - pex le calcul du sinus d'un angle

- Tous les langages ont un certain nombre de **fonctions** qui permettent de connaître immédiatement ce genre de résultat.
    - Certaines sont indispensables : elles permettent d'effectuer des traitements impossibles sans elles.
    - D'autres servent à soulager le programmeur, en lui épargnant de longs algorithmes.

10.1. Structure générale des fonctions

Pex : les langages proposent généralement une fonction SIN. Pour stocker le sinus de 35 dans la variable A, nous écrirons : A ⇦ Sin(35)

Une fonction est donc constituée de 3 parties :

- le **nom** de la fonction.
  - correspond à une fonction proposée par le langage
  - ne s'invente pas

- 2 parenthèses

- une liste de valeurs (arguments, ou paramètres)
  - indispensables à la bonne exécution de la fonction.
  - certaines fonctions exigent un seul argument, d'autres 2,…
  - le nombre d'arguments nécessaire pour une fonction donnée est fixé par le langage.
  - les arguments doivent être d'un certain **type** qu'il faut respecter

10.2. Les fonctions de texte

Une catégorie de fonctions nous permet de manipuler des chaînes de caractères. Tous les langages proposent les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- Len(chaîne) renvoie le nombre de caractères d'une chaîne

Len("Bonjour, ça va ?") vaut ??

- Mid(chaîne, n1, n2) renvoie un extrait de la chaîne commençant au caractère n1 et faisant n2 caractères de long.

Mid("Zorro is back", 4, 6) vaut ??

- Left(chaîne, n) renvoie les n caractères les plus à gauche dans chaîne.

Left("Et pourtant…", 8) vaut ??

- Right(chaîne, n) renvoie les n caractères les plus à droite dans chaîne

Right("Et pourtant…", 4) vaut ??

- Trouve(chaîne1, chaîne2) renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.

Trouve("Un pur bonheur", "pur") vaut ??

- fonction renvoyant le caractère correspondant à un code Ascii donné (fonction Asc) :

Asc("N") vaut ??

10.3. Deux fonctions classiques

- Récupérer la partie entière d'un nombre : A ⇦ Ent(3,228) A vaut ??

- Génèrer un nombre choisi au hasard.
  - sert dans les jeu
    - simuler un lancer de dés
    - simuler le déplacement d'un vaisseau
    - …
  - sert dans la modélisation
    - physique,
    - géographique,
    - économique,
    - …

11. PROCEDURES ET FONCTIONS

- Une application risque de procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement.

    - Pex : saisie d'une réponse par oui ou par non, peuvent être répétés dix fois à des moments différents de la même application.

- ne pas répéter le code correspondant autant de fois que nécessaire, mais

- séparer ce traitement du corps du programme et appeler ces instructions (qui ne figurent donc plus qu'en un seul exemplaire) à chaque fois qu'on en a besoin.

- *Procédure principale* : corps du programme

- *Sous-procédures* : groupes d'instructions auxquels on a recours.

Reprenons cet exemple de question à laquelle l'utilisateur doit répondre par oui ou par non :

Mauvaise Structure
...
Ecrire "Etes-vous marié ?"
Rep = ""
TantQue Rep <> " Oui" **et**
Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Ecrire "Avez-vous des enfants ?"
Rep = ""
TantQue Rep <> "Oui" **et**
Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue

Bonne Structure
Ecrire "Etes-vous marié ?"
RéponseOuiNon()
...
Ecrire "Avez-vous des enfants ?"
RéponseOuiNon()
...
Procédure
RéponseOuiNon()
Rep = ""
TantQue Rep <> " Oui" **et**
Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Fin Procédure

11.1. Passage de paramètres

Analysons cet exemple :

- On écrit un message,
- puis on appelle la procédure pour poser une question,
- puis plus tard, on écrit un autre message, et on relance la procédure pour poser la même question,....

- C'est acceptable, mais ça peut être améliorée :
  - puisqu'avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la procédure appelée. Ca implique 2 choses :
    - lorsqu'on appelle la procédure, on doit lui préciser quel message elle doit afficher avant de lire la réponse
    - la procédure doit être *prévenue* qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

- En langage algorithmique, le message est un **paramètre**.
  - Comme il est transmis de la procédure principale vers la sous-procédure, c'est un **paramètre d'entrée**.
  - La procédure sera déclarée comme suit :

    Procédure RéponseOuiNon(Msg **en Caractère**)

  - Donc la variable Msg (dont on précise le type), signale à la procédure qu'un paramètre peut lui être envoyé.

- A présent, le corps de la sous-procédure sera :

```
Procédure RéponseOuiNon(Msg en Caractère)
Ecrire Msg
Rep = ""
TantQue Rep <> " Oui" et Rep <> "Non"
Ecrire "Tapez Oui ou Non"
Lire Rep
FinTantQue
Fin Procédure
```

La procédure principale (l'appel de procédure) devient alors :

```
RéponseOuiNon("Etes-vous marié ?")
...
RéponseOuiNon("Avez-vous des enfants ?")
...
```

- Pour récupérer le résultat de notre sous-procédure (la réponse à la question posée), on utilise un second paramètre : un **paramètre de sortie** :
  - transmettre la valeur de la variable Rep de la sous-procédure vers la procédure principale, même principe que transmettre la valeur de la variable Msg de la procédure principale vers la sous-procédure.

La déclaration de la procédure devient :

Procédure RéponseOuiNon(Msg **en Caractère,** Rep **en Booléen**)

Et l'appel devient :

```
RéponseOuiNon("Etes-vous marié ?", toto)
si toto = "Oui" alors
...
RéponseOuiNon("Avez-vous des enfants ?", tata)
si tata = "Oui" alors
...
```

11.2. Programmation récursive

- principe : utiliser, pour décrire l'algorithme sur une donnée d, l'algorithme lui-même appliqué à un sous-ensemble de d ou à une donnée d' plus petite

- certains langages ne permettent pas des appels récursifs de sous-programmes.

- Masque parfois un problème de complexité : on utilise en effet une pile « cachée» pour stocker les résultats intermédiaires.

- Ex : le calcul d'une factorielle :

  $N ! = 1 \times 2 \times 3 \times \ldots \times n$

  - Se programme avec une boucle

- Une autre manière est :

  $N ! = n \times (n-1) !$

  =>la factorielle d'un nb, c'est ce nb multiplié par la factorielle du nb précédent.

- On peut imaginer une fonction Fact, chargée de calculer la factorielle.

  - Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent.
  - Et cette factorielle du nombre précédent va être elle-même calculée par la fonction Fact.
    => Autrement dit, on va créer une fonction qui pour fournir son résultat, **va s'appeler elle-même un certain nombre de fois**.
  - c'est la récursivité
  - prévoir une condition d'arrêt :
    - ces auto-appels de la fonction Fact s'arrêtent quand on arrive au nombre 1, pour lequel la factorielle est par définition 1 :

```
Fonction Fact (N en Numérique)
Si N = 1 alors
Renvoyer 1
Sinon
Renvoyer Fact(N-1)
Finsi
Fin Fonction
```

Remarques :

- on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle.
  => caractéristique de la programmation récursive.

- peut être **très économique** en lignes de programmation

- mais **très chère en ressources machine**.
  - la machine est obligée de créer autant de variables temporaires que de " tours " de fonction en attente.

- tout problème formulé en termes récursifs peut également être formulé en termes itératifs !

- ne pas réappliquer l'algorithme à des données plus grandes

- test de terminaison (correspond à un cas où la donnée est suffisamment élémentaire pour être traitée directement sans réapplication de l'algorithme).

Reprenons l'ex de la boucle d'affichage des 100 premiers entiers (chap 4.7):

```
Algorithme CompteJusqueCentRecursif (n : entier)
Début
  Si (n <= 100)
  Alors
    Afficher(n)
    ALaLigne
    CompteJusqueCentRecursif(n+1)
  Fin si
Fin
```

11.3. Variables publiques et privées

- Les sous-procédures et les paramètres posent le problème de la **durée de vie** des variables, de leur **portée**. Une variable peut être déclarée :

  - privée (locale) : elle disparaît (et sa valeur avec) dès que prend fin la procédure ou elle a été créée.
  - publique (ou globale) : elle est conservée intacte pour toute l'application, au-delà des ouvertures et fermetures de procédures.

- En pseudo-code algorithmique, on peut utiliser le mot-clé **Public** pour déclarer une variable publique : **Public** Toto **en Entier**

- On ne déclare comme publiques que les variables qui doivent absolument l'être

  - elles consomment beaucoup de ressources en mémoire.
  - chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres par valeur plutôt que des variables publiques.

12. TRAVAIL A RENDRE

- Sujet : optimisation de l'exercice php de staf14 (ou choix similaire à discuter…)
- Date de retour : 16 avril, minuit
- Buts pédagogiques :
  - Apprendre les bases de réflexion à l'algorithmique
- Contraintes :
  - Exercice
    - Nom de fichier : http://tecfa.unige.ch/staf/staf-i/<login>/staf2x/ex1/
    - Optimisation de votre ex staf14 php
    - Critères d'évaluation :
      - Ça doit marcher
      - Code lisible, commenté et aéré.
      - Code optimisé en fonction du cours

- Rapport
  - Nom de fichier :
  - http://tecfa.unige.ch/staf/staf-i/<login>/staf2x/ex1/comment.html
  - Objectifs de l'exercice pour vous
  - Démarche conceptuelle,
  - Analyse de la situation,
  - Algorithme (en pseudo code)
  - Év. algorithme fonctionnel
  - problèmes rencontrés…
  - Critères d'évaluation :
    - Clarté, on doit pouvoir retracer votre cheminement.
    - Précision de l'algorithme
    - références

13. SOURCES

Froidevaux, C., Gaudel, M-C. & Soria, M.(1990). *Type de données et algorithmes.* Paris : McGraw-Hill

http://www.netalya.com/fr/algo-intro.asp : cours d'algorithmique : introduction

http://www.u-picardie.fr/~ferment/initiation/sommaire.html : initiation à l'algorithmique et à la programmation

http://www.boretti.gotdns.com/Ecole/fichier/138efficace.pdf.gz

http://www.grappa.univ-lille3.fr/~torre/guide.php?id=coursalgo : Initiation à l'algorithmique