Pistes pour un enseignement d'algorithmique

Pascal CHAUVIN
pascal.chauvin@ac-nantes.fr

Lycée François Truffaut

IREM des Pays de la Loire

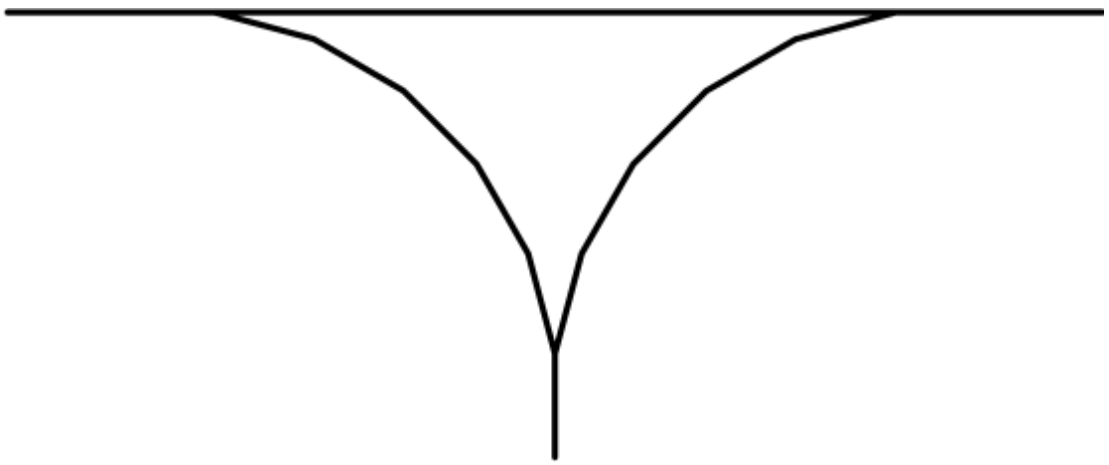
Journées nationales APMEP
23 octobre 2017 – Nantes

Le cadre

- On ne revient pas ici sur la distinction entre *algorithme* et *programme*;
- Conformément aux préconisations, l'algorithmique est abordée en enseignement secondaire sous l'angle de la programmation.

Introduction

On demande de réaliser un programme qui trace la figure suivante :



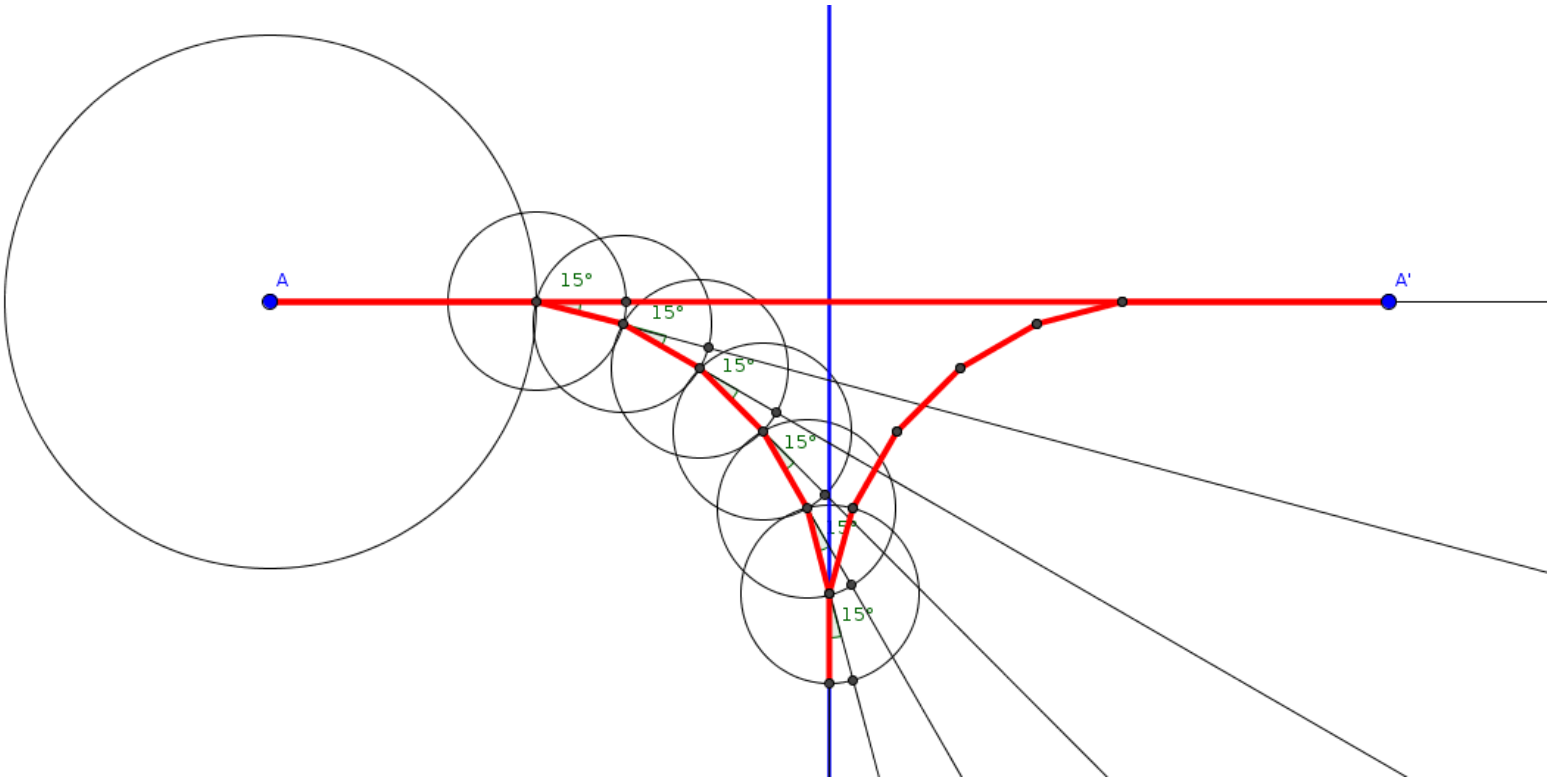
La figure peut être décrite comme suit :

- tracer un premier segment de longueur 120 pixels ;
- on répète six fois les deux instructions :
  - tourner à droite de 15 degrés ;
  - tracer un segment de longueur 50 pixels ;
- on complète le tracé par symétrie ;
- le pointeur rejoint son point de départ.

Toute la difficulté consiste à revenir au premier pixel, après de nombreux essais infructueux [1]...

1. Dès la présentation de ce qui semble être un succès pour la classe, on ne résiste pas à la tentation de modifier les longueurs initiales.

On propose alors de réaliser la figure avec un logiciel de géométrie dynamique :



Le recours aux connaissances mathématiques s'impose de lui-même :

- la trigonométrie du triangle rectangle ;
- les calculs d'angles et de longueurs ;
- la mesure des angles en radians (elle est imposée par le langage, car la fonction Python calculant le cosinus reçoit un argument exprimé en radians) ;
- la symétrie axiale...

On utilise le langage Python à la mode du langage LOGO [2] :

```
from turtle import *

forward(120)

for k in range(6):
    right(15)
    forward(50)

# suite du programme ci-dessous
```

Premier segment, puis premier « arc ».

2. Seymour PAPERT, M.I.T. Labs, 1967.

Obtention du second « arc » :

```
right(180)
forward(50)

for k in range(6):
    right(15)
    forward(50)

# suite du programme ci-dessous
```

```
forward(120 - 50)

right(180)

from math import *

forward(2*(120 + 50*(cos(pi/12) + cos(2*pi/12) +
    cos(3*pi/12) + cos(4*pi/12) + cos(5*pi/12) +
    cos(6*pi/12))))

hideturtle()
exitonclick()
```

Second segment, puis retour à l'origine du tracé.

- Le recours au logiciel de géométrie dynamique permet d'obtenir une représentation du problème qui impose d'exécuter « à la main » l'algorithme de dessin ; ce qui va permettre ensuite de concevoir le calcul correct de la solution.
- La solution proposée est adaptée *au plus près* à l'énoncé : les valeurs des données (en entrée) de l'algorithme sont inscrites « en dur » dans le texte de l'algorithme, qui perd de fait en généralité.

Les deux idées suivantes peuvent constituer un fil conducteur pour un enseignement d'algorithmique dans le cadre des programmes de Mathématiques :

- justifier un algorithme (terminaison et exactitude) ;
- convoquer les connaissances mathématiques pour produire une justification.

Un itinéraire

Premiers pas

Le programme suivant est le premier programme [3] [4] en langage Python proposé aux élèves :

```
from turtle import *

forward(50)
left(80)
forward(120)

exitonclick()
```

3. La première ligne est nécessaire pour le langage.
4. La dernière ligne est pratique pour l'utilisateur, mais ne présente, comme la première ligne, aucun intérêt mathématique.

Le programme donne à l'exécution :

Une pratique courante –recommandée pour l'apprentissage d'un langage de programmation– consiste à modifier le texte du programme afin d'identifier le rôle de chaque instruction : on effectue une modification, puis on observe l'effet obtenu lors de l'exécution.

```
forward(50)
left(80)
forward(120)
```

Par exemple :

- dans le programme initial, remplacer la valeur 120 en 20 ;
- dans le programme initial, remplacer le nombre de la seconde instruction (ligne) par 45 ;
- dans le programme initial, supprimer la dernière ligne.
- ...

Cette activité permettra de déduire le rôle des instructions *forward* et *right*.

Exercice

Écrire un programme qui trace un carré de côté 100 pixels.

```
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
```

Exercice

Écrire un programme qui trace un carré de côté 100 pixels et un carré de côté 70 pixels.

```
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)

forward(70)
left(90)
forward(70)
left(90)
forward(70)
left(90)
forward(70)
```

À ce stade, il convient d'identifier les insuffisances de cette conception :

- Comment prendre en compte la répétition pour un nombre élevé de côtés ?
- Comment prendre en compte le tracé d'un grand nombre de polygones ?
- Comment modifier aisément la longueur du côté d'un polygone régulier ?

```
longueur = 100
forward(longueur)
left(90)
forward(longueur)
left(90)
forward(longueur)
left(90)
forward(longueur)

longueur = 70
forward(longueur)
left(90)
forward(longueur)
left(90)
forward(longueur)
left(90)
forward(longueur)
```

On motive l'utilisation d'une variable ; sa valeur peut éventuellement être modifiée pendant l'exécution.

L'emploi d'une variable permet :

- de modifier aisément un programme ;
- de réduire le risque d'erreurs.

La valeur d'une variable peut évoluer au cours de l'exécution du programme.

Le caractère *mutable* d'une variable introduit de fait un nouveau type d'erreur, propre à la notion de programme : pour la plupart des langages de programmation [5], les évaluations reposent sur la succession dans le temps des modifications de l'état de la mémoire [6].

5. Modèle de John VON NEUMANN.
6. Le langage fonctionnel Haskell procède de manière conforme à l'usage mathématique.

Routines

```
def machin():
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)

machin()
forward(110)
machin()
forward(110)
machin()
```

On a introduit la notion de *routine* : un programme dans le programme.

Exercice

Écrire un programme qui trace un polygone régulier de 18 côtés de longueur 50 pixels.

```python
import turtle

def routine_A():
    turtle.forward(50)
    turtle.left(20) # pourquoi le nombre 20 ?

def routine_B():
    routine_A()
    routine_A()

# suite du programme ci-dessous
```

```
def routine_C():
    routine_B()
    routine_B()

def routine_D():
    routine_C()
    routine_C()

def routine_E():
    routine_D()
    routine_D()

routine_E()
routine_C()

turtle.exitonclick()
```

Quelques remarques :

- le programme repose sur la décomposition binaire du nombre 18 ;
- rien n'impose de décomposer 18 en base 2 : proposer un programme reposant sur la décomposition en base 6 ;
- écrire un programme qui trace un pentagone régulier de côté 20 pixels.

On peut ensuite introduire la notion de *paramètre* d'une routine :

```
def figure(longueur):
    forward(longueur)
    left(90)
    forward(longueur)
    left(90)
    forward(longueur)
    left(90)
    forward(longueur)
    left(90)

figure(100)
figure(70)
```

On observera que le programme ci-dessus ne comporte *fondamentalement* aucune variable.

- On a réalisé une abstraction : chaque programme peut bénéficier d'une nouvelle instruction nommée *figure* (qui trace un carré de côté variable), dont les détails peuvent rester inconnus de l'utilisateur.
- La notion de routine permet de réduire (et rendre plus robuste) le code du programme.

Jusqu'ici, on a défini :

- les notions d'instruction et de séquence d'instructions ;
- la notion de routine, sans ou avec paramètre(s) ;
- la notion de variable.

Dans un cursus d'enseignement, on prendra également soin d'aborder quelques compléments :

- la distinction (dans le cadre algorithmique) entre *fonction* et *procédure* ;
- la distinction des sens algorithmique et mathématique du terme de « fonction ».

Alternatives

```
x = float(input("nombre ? "))

print(x)

if x == 8:
    print("huit")

print("fini")
```

On propose aux élèves de déterminer ce que réalise ce programme, en effectuant divers choix pour la valeur du nombre.

On dispose dès lors du moyen d'écrire des programmes dont le comportement peut changer, selon le contexte.

On suppose maintenant que les élèves connaissent uniquement l'« alternative simple », c.-à-d. la construction « si ... alors ... ».

```
x = float(input("nombre ? "))

truc = 1789
a = truc

if x < 8:
    t = a
    a = t + 1
    print("bleu")

if a == truc:
    print("rouge")
```

On propose aux élèves de déterminer ce que réalise ce programme, puis on demande de proposer un programme plus simple qui donne la même réponse [7].

7. pour la même entrée, dans « tous » les cas possibles.

```
x = float(input("nombre ? "))

if x < 8:
    print("bleu")

if x >= 8:
    print("rouge")
```

On a placé les élèves en position de construire la condition contraire de « $x < 8$ ».

```
x = float(input("nombre ? "))

if x < 8:
    print("bleu")
else:
    print("rouge")
```

On présente enfin la forme complète « si ... alors ... sinon ... ».

Exercice

a et b sont deux nombres fixés, tels que $a < b$.

On demande d'écrire un programme qui demande à l'utilisateur un nombre x. Le programme indique ensuite si $x \in [\, a\,;\, b\,]$ ou non.

```
a = 7
b = 17

x = float(input("nombre ? "))

if a <= x:
    if x <= b:
        print("x dans l'intervalle")
```

Puis on demande comme nouvel exercice de compléter le programme pour délivrer un diagnostic plus précis lorsque x est hors intervalle.

On donne ensuite le « et logique » :

```
a = 7
b = 17

x = float(input("nombre ? "))

if a <= x and x <= b:
    print("x dans l'intervalle")
```

Les exemples précédents ont permis de préparer le terrain :

- pour les booléens ;
- pour la notion de « et » logique.

Itérations : premier modèle

Nous disposons à ce stade des notions de routine et d'alternative.

```
n = int(input("nombre entier naturel ? "))

def routine(k):
    if k > 0:
        print("bleu")
        routine(k - 1)

routine(10)
```

Que fait ce programme ?

```
pascal@hal9000: python3 bleu5.py
bleu
bleu
bleu
bleu
bleu
bleu
bleu
bleu
bleu
bleu
pascal@hal9000:
```

Le programme affiche dix fois le mot « bleu ».

On peut « tracer » l'exécution du programme (par une instruction d'affichage) pour y voir plus clair :

```
n = int(input("nombre entier naturel ? "))

def routine(k):
    print("la valeur de k est", k)
    if k > 0:
        print("bleu")
        routine(k - 1)

routine(10)
```

Que fait ce programme ?

On demande de modifier ce programme pour obtenir l'affichage des dix premiers multiples de 8.

```
def routine(k):
    if k > 0:
        print(k * 8)
        routine(k - 1)

routine(10)
```

```
pascal@hal9000: python3 prog1.py
80
72
64
56
48
40
32
24
16
8
pascal@hal9000:
```

Le programme affiche dix entiers consécutifs tous multiples de 8.

```
def routine(k):
    print(k * 8)
    if k > 0:
        routine(k - 1)

routine(10)
```

```
pascal@hal9000: python3 prog2.py
80
72
64
56
48
40
32
24
16
8
0
pascal@hal9000:
```

Le programme affiche cette fois onze nombres tous multiples de 8, dont le nombre 0.

Peut-on obtenir ces nombres en ordre croissant ?

```
def routine(k):
    if k > 0:
        routine(k - 1)
    print(k * 8)

routine(10)
```

```
pascal@hal9000: python3 prog3.py
0
8
16
24
32
40
48
56
64
72
80
pascal@hal9000:
```

- Les notions d'alternative et de routine suffisent à concevoir des itérations. On parle de style et de programmation *récursifs*.
- Nous avons présenté des exemples de procédures récursives ; la notion de fonction récursive [8] existe au même titre.

8. On donne quelques exemples plus loin dans l'exposé.

Itérations : second modèle

```python
n = int(input("nombre entier entre 0 et 4 (compris) ? "))

c = 0

if c < n:
    print("bleu")
    c += 1 # remplace "c = c + 1"

if c < n:
    print("bleu")
    c += 1

if c < n:
    print("bleu")
    c += 1

if c < n:
    print("bleu")
    c += 1

print("fini")
```

Que fait ce programme ? Quelles en sont les limitations ?

Au passage :

```
a = 8

""" permet d'augmenter de 1 la valeur de a au moyen d'une
variable temporaire b """
b = a + 1
a = b

""" autrement """
a = a + 1

""" un autre moyen """
a += 1
```

On ne manquera pas à ce sujet d'évoquer le malheureux signe « = » pour désigner l'affectation, dont l'emploi n'a hélas pour seule justification que l'économie de moyens dans la fabrication des premières machines à écrire...

Il est possible de remplacer tous les blocs « if » par un seul bloc « while » (il conviendrait de libeller autrement le message en début de programme) :

```
n = int(input("nombre entier entre 0 et 4 (compris) ? "))

c = 0

while c < n:
    print("bleu")
    c += 1

print("fini")
```

On observe que les itérations ne sont plus limitées en nombre.

On vient de présenter la notion de boucle « tant que ... faire ... », qui étend[9] le modèle de l'alternative complète.

9. avec la nuance essentielle de reprendre l'exécution, sous condition, en début de bloc.

Pourquoi la récursivité ?

Reconnaître un nombre entier

Avec les notions de fonction et de récursivité, on peut construire un automate qui détermine si une chaîne de caractères représente ou non un entier naturel (ici en base dix).

Voici une *grammaire* simplifiée pour un tel *automate* [10] :

```
entier = chiffre_ou_0 | chiffre_sauf_0 chiffre_s
chiffre_s = chiffre_ou_O | chiffre_ou_0 chiffre_s
chiffre_ou_0 = 0 | chiffre_sauf_0
chiffre_sauf_0 = 1 | 2 | 3 | ... | 9
```

La grammaire [11] ci-dessus est récursive.

En d'autres termes, la manière d'écrire un nombre entier est un processus récursif.

10. Cf. théorie mathématique des langages.

11. La terminologie choisie dans l'exemple est malheureuse, faute de mieux, car 0 est un chiffre depuis les derniers astronomes babyloniens...

Pour information, voici un module « entier.py » qui implémente l'automate précédent :

```
from liste import *

def est_ZERO(ch): # terminal 0
    return ch == '0'

def est_NON_ZERO(ch): # terminaux 1, 2, 3, ..., 9
    return ch in "123456789"

def chiffre_sauf_0(ch):
    return est_NON_ZERO(ch)

def chiffre_ou_0(ch):
    return est_ZERO(ch) or chiffre_sauf_0(ch)
```

(suite du code ci-après)

(suite et fin du code)

```
def chiffre_s(x):
    if longueur(x) > 1:
        return chiffre_ou_0(tete(x)) and chiffre_s(suite(x))
    else:
        return chiffre_ou_0(tete(x))

def entier(x):
    if longueur(x) > 1:
        return chiffre_sauf_0(tete(x)) and chiffre_s(suite(x))
    else:
        return chiffre_ou_0(tete(x))
```

À titre d'information, on présente le code du module « liste.py » :

```
def est_vide(l):
    return len(l) == 0

def tete(l):
    if est_vide(l):
        return None
    else:
        return l[0]

def suite(l):
    if est_vide(l):
        return []
    else:
        return [x for x in l[1:]]

def longueur(l):
    if est_vide(l):
        return 0
    else:
        return 1 + longueur(suite(l))
```

Le code du module « liste.py » complète le type primitif « list » de Python en définissant deux fonctions :

- la fonction « tete » : elle retourne le premier élément d'une liste non vide ;
- la fonction « suite » : elle retourne une nouvelle liste, dont les éléments sont ceux de la liste initiale privée de sa tête.

Les définitions précédentes suffisent (presque [12]) à mettre en œuvre la définition formelle suivante :

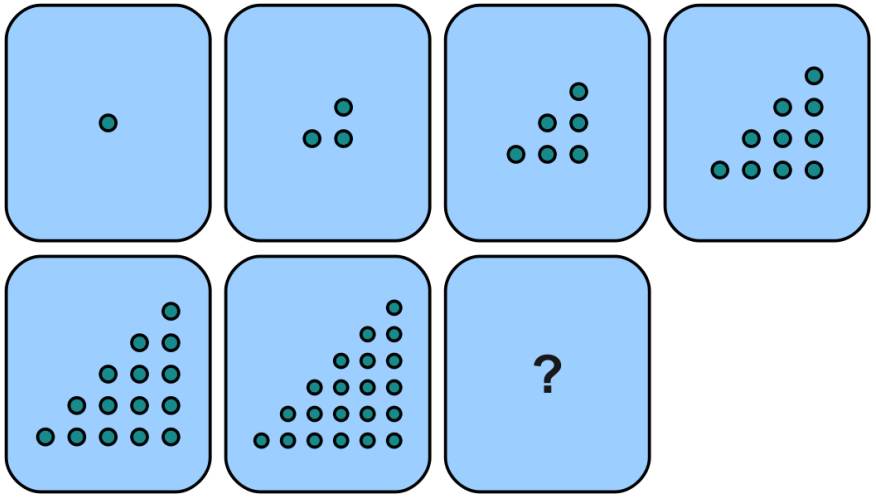
$$\text{liste} = \left| \begin{array}{c} \text{vide} \\ \underline{\text{ou}} \\ \text{élément} \quad \underline{\text{et}} \quad \text{liste} \end{array} \right.$$

La structure de liste est un type de donnée récursif.

12. Pour une implémentation, on doit pouvoir créer une liste, adjoindre un nouvel élément, supprimer un élément...

Les jetons

On dispose de cartes sur lesquelles on veut placer des jetons :



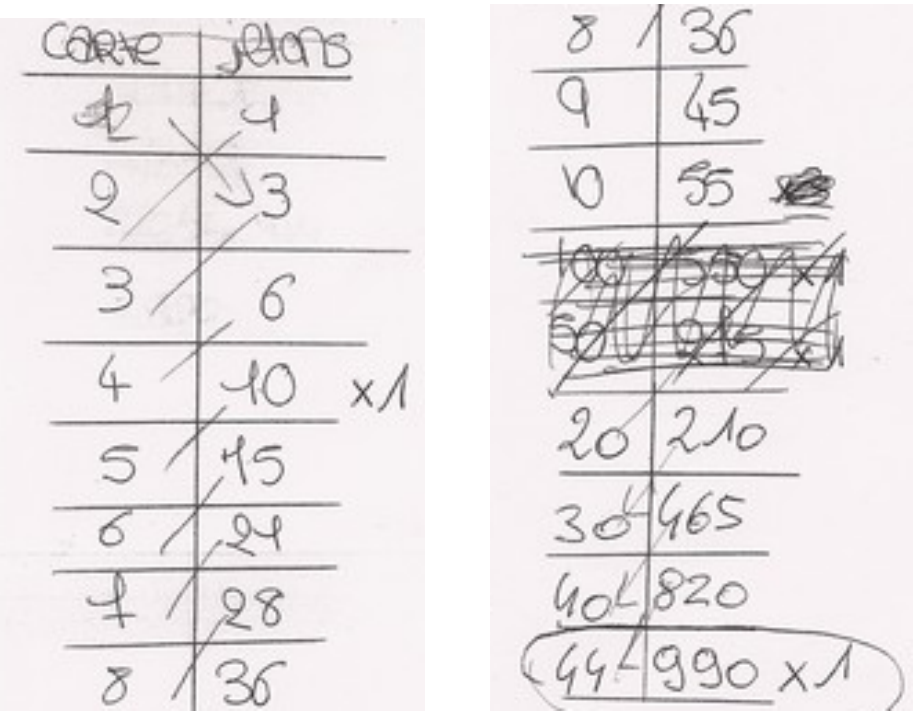
La séquence proposée aux élèves est constituée de deux parties :

1. pour le premier travail, on ne réalise qu'une seule carte (les précédentes sont détruites) ;
2. pour le second travail, on collectionne toutes les cartes que l'on peut réaliser dans la limite des jetons disponibles.

Pour réaliser de telles cartes, on dispose d'un paquet de 1 000 jetons, autant de cartes vierges et aussi grandes que nécessaire à la fabrication.

Calculer le nombre de jetons de la carte qui pourrait, en théorie, comporter le plus grand nombre de jetons possible.

Un groupe d'élèves (4e de collège) utilise un « tableur papier » : on retrouve une méthode de calcul des effectifs cumulés vue plus tôt dans l'année en statistiques.



Un autre groupe énonce un procédé récursif du calcul attendu :

On additione le nombre de jetons sur la carte précédente et le nombre de jetons sur un côté de l'angle droit du triangle de la carte suivante.

La traduction effective du calcul récursif en langage Python :

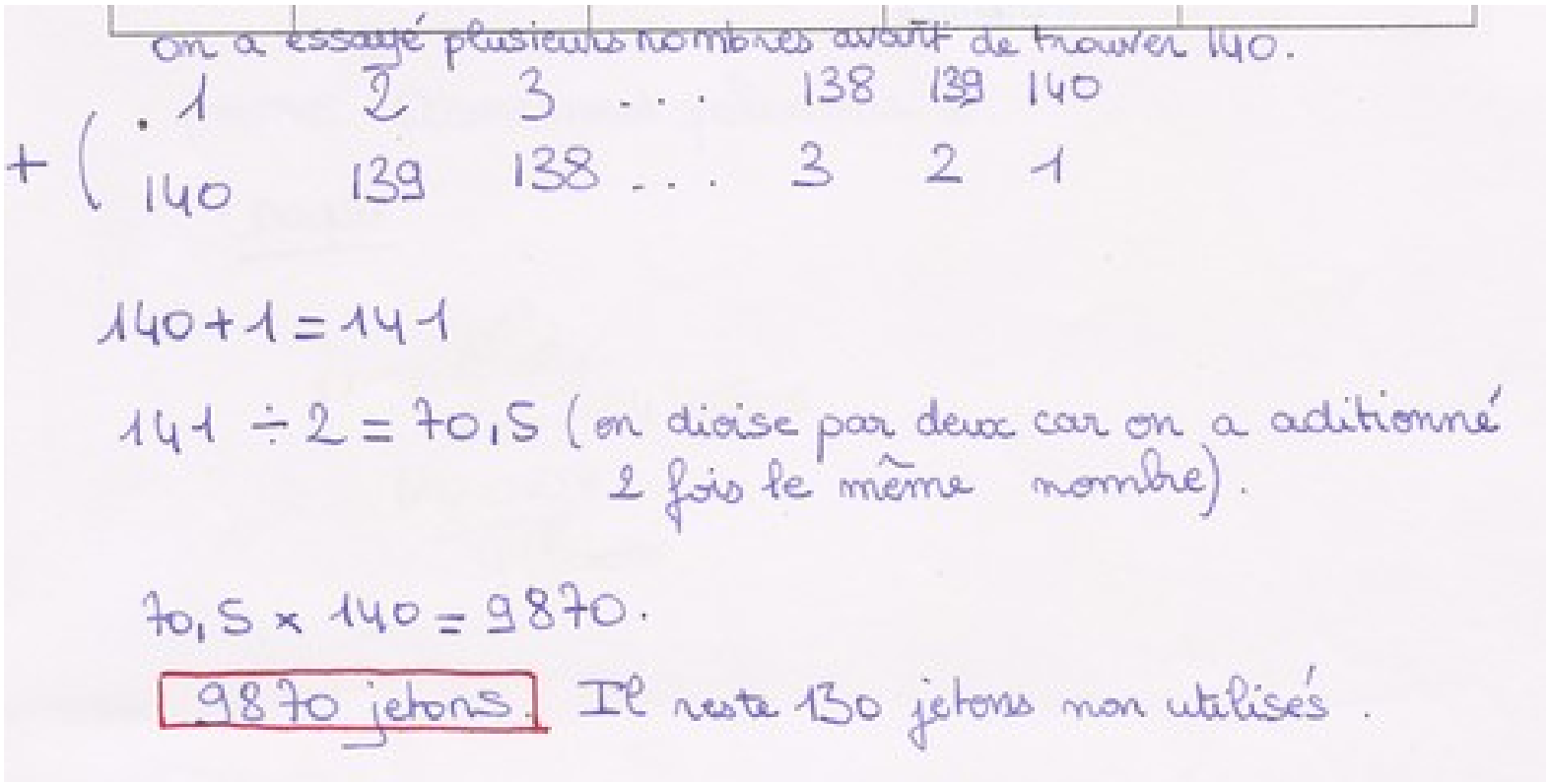
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def jetons(num_carte):
    if num_carte > 1:
        return num_carte + jetons(num_carte - 1)
    else:
        return 1

print(jetons(5))
```

Le programme peut être complété par une boucle qui teste chaque carte jusqu'à satisfaire la condition du plus grand nombre de jetons, dans la limite de 1000 jetons.

La justification du calcul par un groupe d'élèves, issue du dessin traditionnel :



on a essayé plusieurs nombres avant de trouver 140.

$$+\left(\begin{array}{cccccccc} 1 & 2 & 3 & \dots & 138 & 139 & 140 \\ 140 & 139 & 138 & \dots & 3 & 2 & 1 \end{array}\right.$$

$140 + 1 = 141$

$141 \div 2 = 70{,}5$ (on divise par deux car on a additionné 2 fois le même nombre).

$70{,}5 \times 140 = 9870$.

$\boxed{9870 \text{ jetons}}$ Il reste 130 jetons non utilisés.

L'algorithme « Shunting yard »

En 1961, pour ses travaux sur le langage ALGOL-60,
Edsger Wybe DIJKSTRA publie un article dans lequel il décrit notamment l'évaluation des expressions numériques, aujourd'hui connu sous le nom d'algorithme « Shunting yard ».

Un extrait du texte original de E. W. DIJKSTRA :

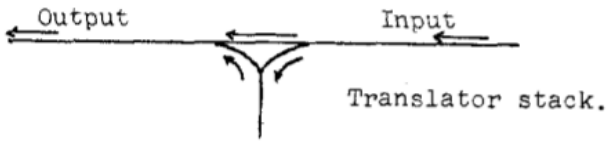
8 < ≤ = ≥ > ≠

9 + -

10 neg */ ("neg" represents the so-called unary"-"operation)

11 ↑

The translation process shows much resemblance to shunting at a three way railroad junction of the following form



At he right the symbols of the ALGOL text come in in order from left to right, at the left the successive orders of the object program are produced. The rule is that incoming identifiers are sent to the output in the form of a TAKE order ("TAKE address of" if the identifier occurs to the left of the ":=" symbol, otherwise "TAKE value of") . Incoming operators receive their priority numbers and are then sent to the translator stack, but before the latter happens, operators in the translator stack are transported from it to the output as long as their priority number is greater then or equal to the proirity number of the new operator. For instance, at a certain stage of

Où le calcul est aussi une affaire d'écriture :

notation préfixe	$+$	2	1	additionner 2 et 1
notation infixe	2	$+$	1	notation mathématique usuelle
notation postfixe	2	1	$+$	pour l'automate (ordinateur)

- En notation préfixe, l'opérateur apparaît avant ses opérandes.
- En notation infixe, l'opérateur est placé entre les opérandes.
- En notation postfixe, l'opérateur vient en dernière position, après les opérandes.

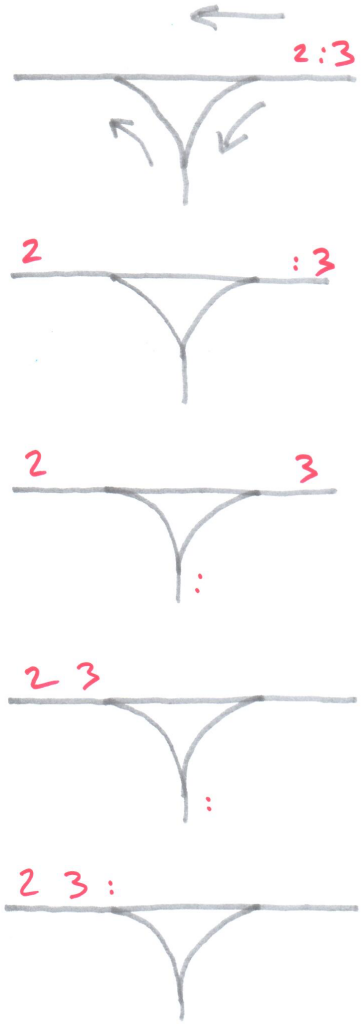
La dernière notation est particulièrement adaptée au calcul par un robot (machine, langage...) : on utilise pour cela les structures de *file* et de *pile*.

À titre d'illustration, voici quelques exemples de traduction de la notation infixe vers la notation postfixe [13].

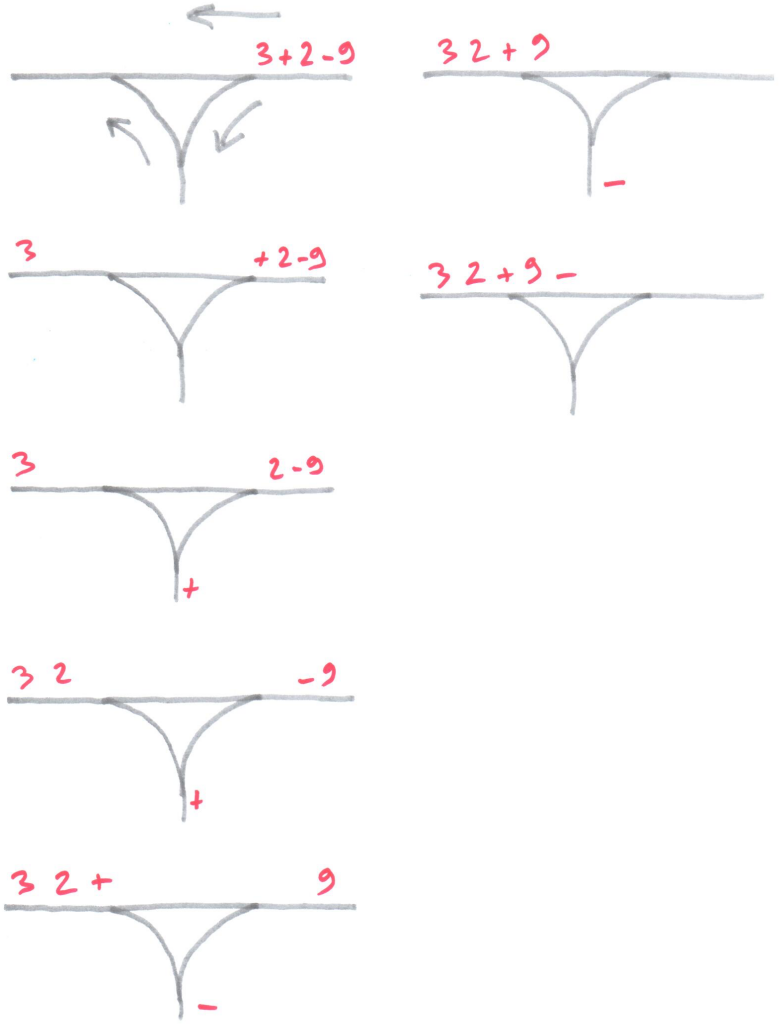
Note : dans les exemples qui suivent, pour simplifier, les nombres et les opérateurs ont été limités à un seul chiffre ou caractère du point de vue lexical, ce qui n'enlève rien à la généralité du propos.

13. Il reste tout à fait possible de passer d'une notation à une autre, au moyen d'*arbres*.

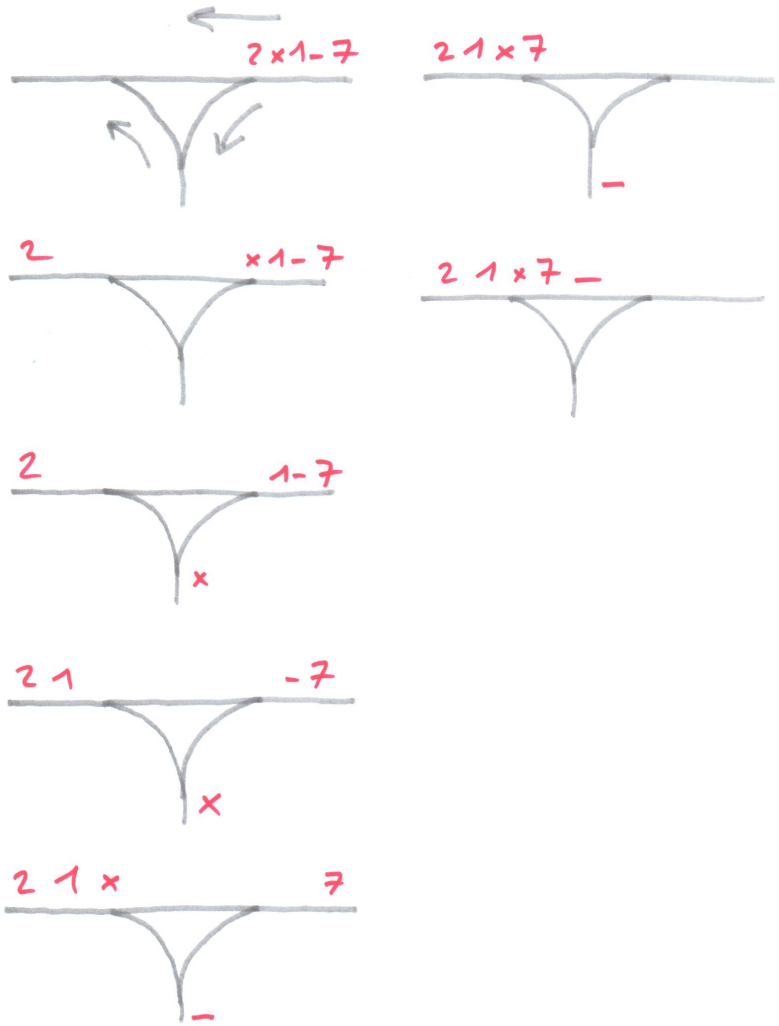
Traduction de l'expression infixe « 2 ÷ 3 » :



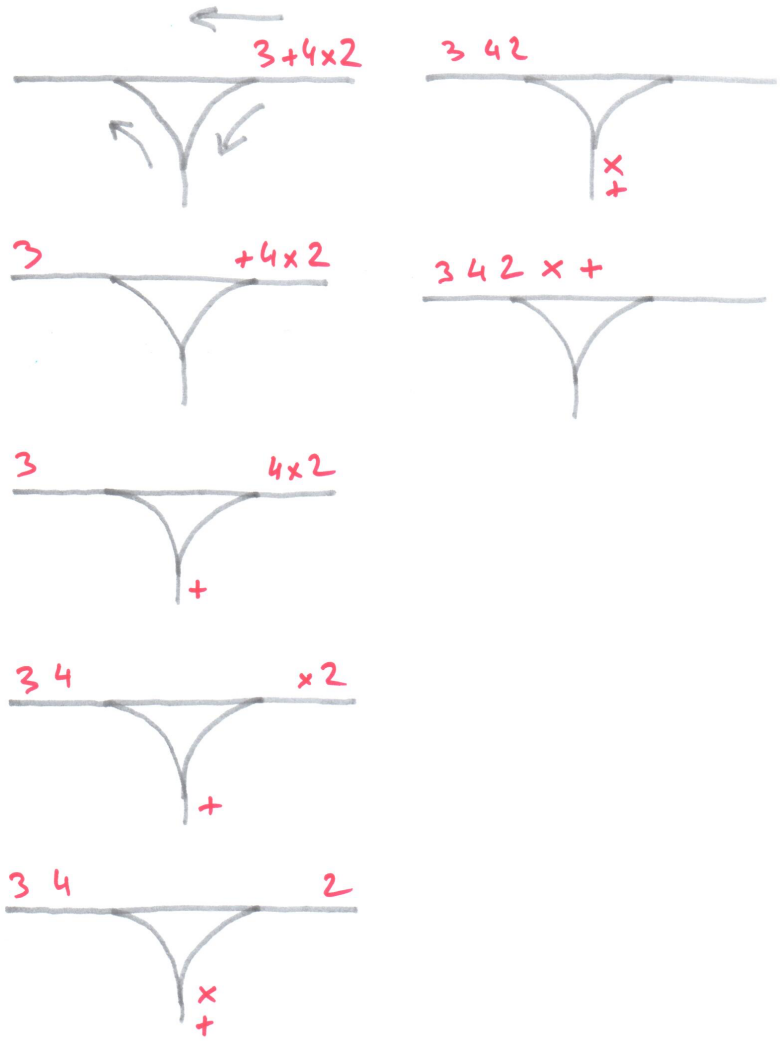
Traduction de l'expression infixe « 3 + 2 − 9 » :



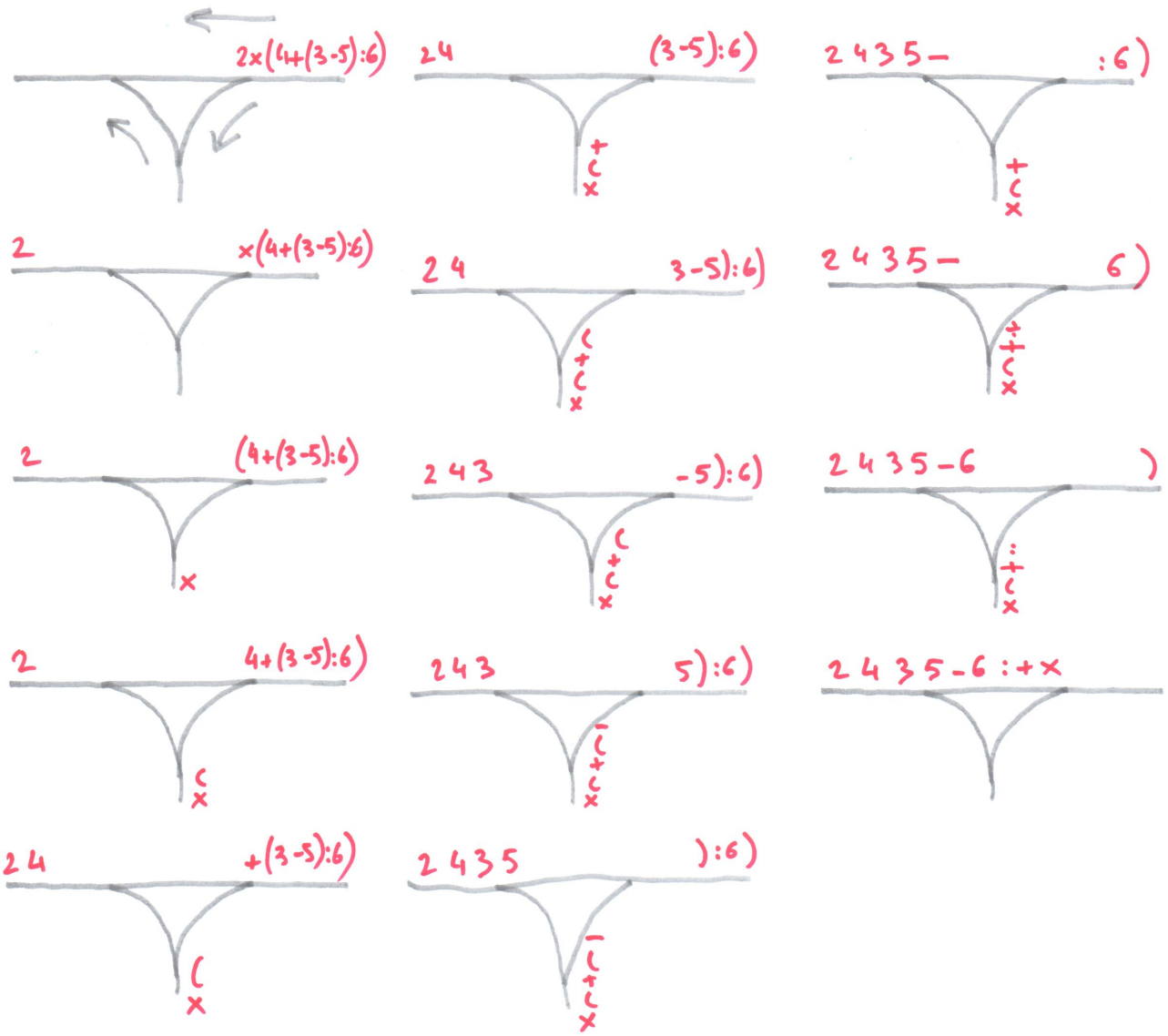
Traduction de l'expression infixe « $2 \times 1 - 7$ » :



Traduction de l'expression infixe « $3 + 4 \times 2$ » :

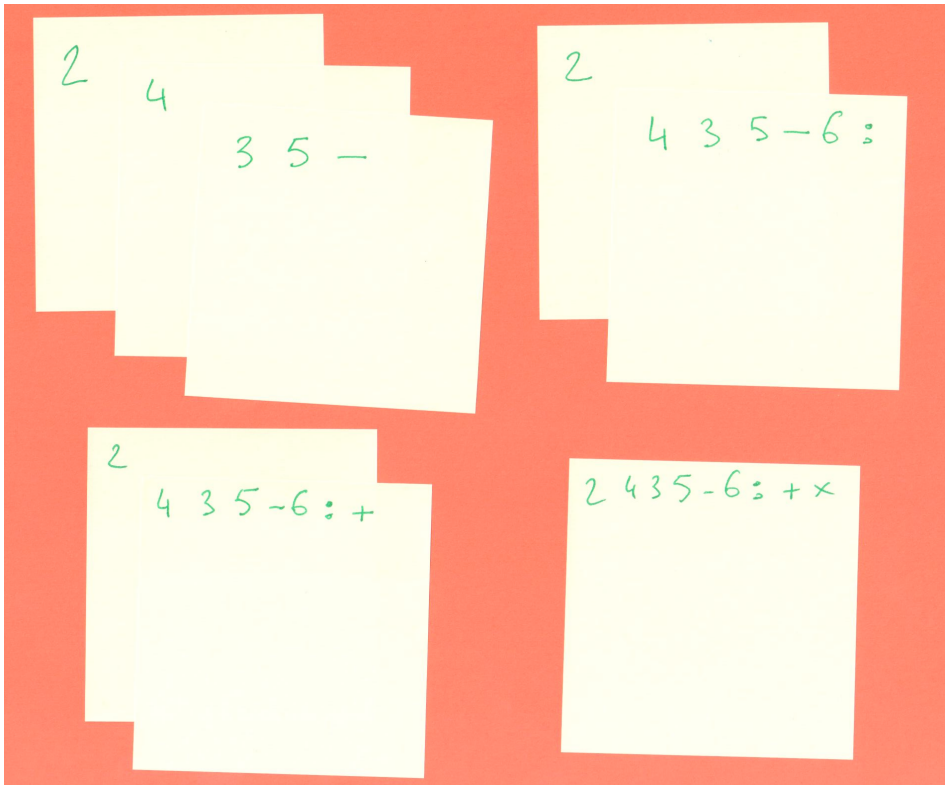


Traduction de l'expression infixe [14] « $2 \times \big(4 + (3 - 5) \div 6\big)$ » :



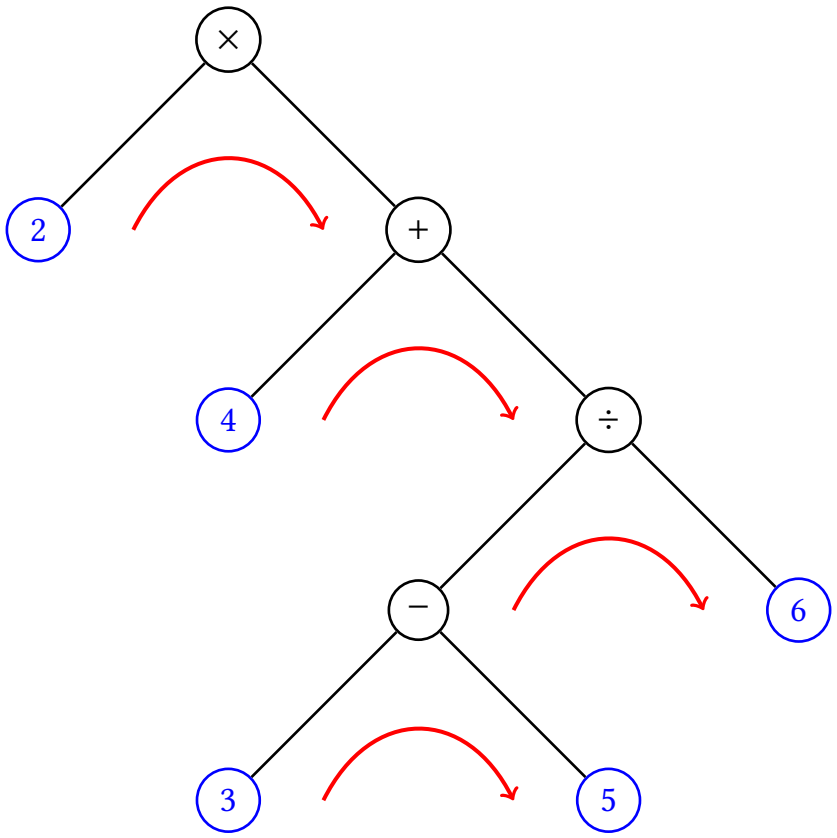
14. Une troisième paire de parenthèses est implicitement présente.

La récursivité peut être modélisée par un empilement de calques qui expriment les paires de parenthèses (imbriquées dès lors qu'on travaille avec des opérateurs binaires) :



Une expression infixe est une expression récursive !

Voici une autre représentation du calcul précédent, sous la forme d'un arbre binaire :



Un arbre binaire (au sens général) est une structure de données... récursive :

arbre	=		vide			
			<u>ou</u>			
		sous-arbre gauche	<u>et</u>	élément	<u>et</u>	sous-arbre droit

Évaluation de l'expression postfixe « $2\,;\ 4\,;\ 3\,;\ 5\,;\ -\,;\ 6\,;\ \div\,;\ +\,;\ \times$ » sur un calculateur à pile :

$$\longrightarrow$$

				5		6			
			3	3	-2	-2	$\frac{-1}{3}$		
		4	4	4	4	4	4	$\frac{11}{3}$	
	2	2	2	2	2	2	2	2	$\frac{22}{3}$

Chaque colonne représente l'évolution « au cours du temps » de la même pile de calcul.

D'une étape du calcul à la suivante :

- Si l'élément lu (dans l'expression postfixe) est un nombre, on l'empile.
- Si l'élément lu est un opérateur, on effectue l'opération en dépilant les deux opérandes [15], en prenant soin de respecter l'ordre des opérandes pour l'opérateur concerné (qui peut ne pas être commutatif). On empile ensuite le résultat.
- En fin d'évaluation, la pile ne contient qu'un nombre : le résultat.

15. on ne considère ici que des opérateurs binaires.

In memoriam

Évaluation de l'expression « 2 ; 4 ; 3 ; 5 ; − ; 6 ; ÷ ; + ; × » avec un programme de simulation d'un calculateur à pile (en Python) :

```
from libcalcul import *

effacer()

empiler(2)
empiler(4)
empiler(3)
empiler(5)
soustraire()
empiler(6)
diviser()
additionner()
multiplier()

valeur()
```

On peut proposer comme exercice aux élèves de se placer dans le rôle du calculateur.

Pour l'activité des élèves, on peut par exemple mettre à leur disposition le module « libcalcul.py » [16].

On peut proposer deux types d'exercice aux élèves :

1. Écrire le programme de calcul pour le calculateur postfixe, sur la donnée d'une expression sous sa forme usuelle infixe.
2. Évaluer l'expression en se plaçant dans le rôle du calculateur.

16. L'écriture d'un tel module est un exercice pour le professeur.

Conclusions

- L'emploi d'un véritable langage de programmation pour l'apprentissage de l'algorithmique apporte la notion centrale de routine (fonction ou procédure).
- La récursivité est une notion centrale en algorithmique, souvent occultée dans l'enseignement des mathématiques : on peut s'interroger sur les erreurs qui pourraient en résulter.
- La récursivité est indissociable de la notion mathématique de récurrence.

- L'emploi d'un véritable langage de programmation pour l'apprentissage de l'algorithmique apporte la notion centrale de routine (fonction ou procédure).
- La récursivité est une notion centrale en algorithmique, souvent occultée dans l'enseignement des mathématiques : on peut s'interroger sur les erreurs qui pourraient en résulter.
- La récursivité est indissociable de la notion mathématique de récurrence.
- « Toute virtuosité dans les calculs est exclue. » Vraiment ?

Lectures

- ALGOL-60 Translation
  Edsger W. Dijkstra
- Computer Science Unplugged
  Tim Bell, Ian Witten, Michael Fellows
- Mathématiques pour l'informatique
  Jacques Vélu, Geneviève Avérous, Isabelle Gil, Françoise Santi – Dunod
- Mini-manuel d'algorithmique et de programmation
  Vincent Granet – Dunod
- Mini-manuel de programmation fonctionnelle
  Éric Violard – Dunod
- Programmation de droite à gauche et vice-versa
  Pascal Manoury – Paracamplus

Merci de votre attention