Introduction aux fondements de l'informatique
CPI103 Semestre 1
Université Bordeaux 1

16 décembre 2013

Table des matières

1 Fonctionnement d'un ordinateur

1.1 Architecture d'un ordinateur

Un *ordinateur* est un système constitué d'une unité de calcul et d'une mémoire. La *mémoire* contient un ensemble de données codées par une suite de 1 et de 0. L'*unité de calcul* est la partie de l'ordinateur qui permet de lire, écrire et modifier la mémoire.

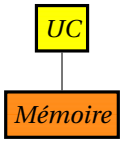


FIGURE 1 – Architecture d'un ordinateur

Il existe différentes façons de réaliser un ordinateur. Par exemple, la machine de Turing, représentée à la figure 2, est un modèle théorique d'un ordinateur.

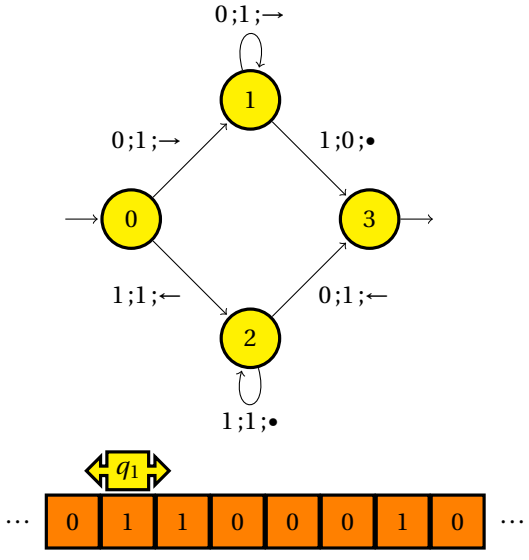


FIGURE 2 – Exemple de machine de Turing

Un modèle théorique d'un ordinateur est une représentation abstraite d'un ordinateur décrite à l'aide du langage mathématique. Dans la pratique, il n'est pas utiliser pour réaliser des programmes. Il est utilisé pour démontrer qu'un problème peut être résolu par un ordinateur.

Dans les ordinateurs modernes, l'unité centrale et la mémoire sont situées dans plusieurs endroits différents. L'ordinateur est aussi accompagné de nombreux *périphériques* : claviers, écrans, enceintes, afin de permettre aux utilisateurs d'interagir avec l'ordinateur. La figure 3 montre l'architecture d'un ordinateur actuel :

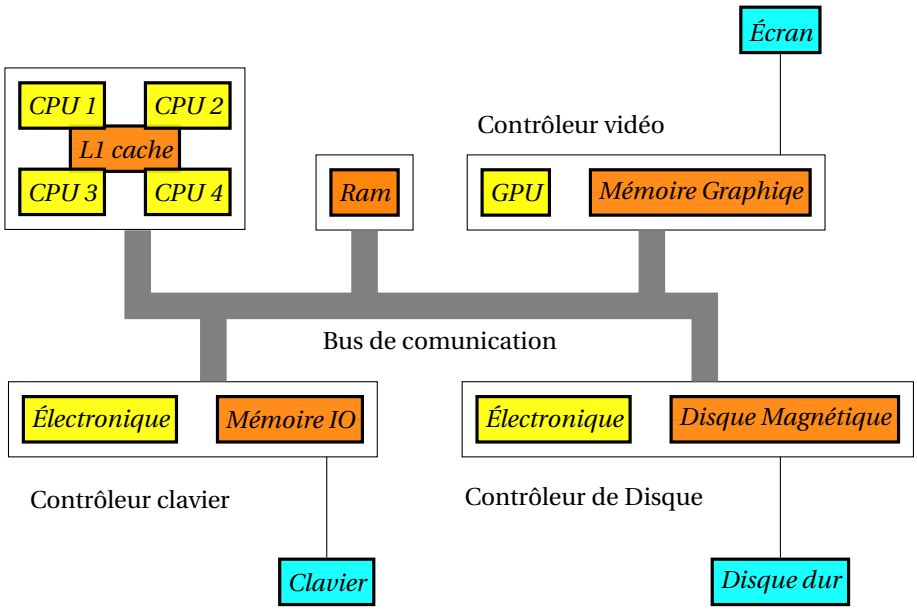


FIGURE 3 – Architecture d'un ordinateur

Dans la figure 3, les boîtes oranges représentent la mémoire, les boîtes jaunes les unités de calculs et les boîtes bleues les périphériques. Dans un ordinateur, le système d'exploitation gère les unité de calcul et les mémoires de sorte que, pour le programmeur, tout se passe comme s'il y avait une unique mémoire et une unique unité de calcul, comme cela est présentée dans la figure 1.

1.2 La mémoire

La *mémoire* d'un ordinateur est une suite de 0 et de 1.
Voici à quoi ressemble un fragment de mémoire :

...1100010100100101001101010110 01...

Chaque entier de cette suite est appelé un bit (en anglais : bit). Il désigne la quantité élémentaire d'information en informatique.

Généralement, en informatique, on regroupe les bits par groupe de 8 que l'on appelle *octet* (en anglais : Bit (avec un grand B). On décrit ainsi les tailles des différentes mémoires à l'aide de cette unité.

Ainsi,

- un disque dur contient généralement entre 100 Go et 400 Go ;
- une clé usb contient entre 1 et 32 Go ;
- un mémoire vive (RAM : Random Access Memory) contient 1 à 4 Go ;
- la mémoire cache du processeur contient (1 à 4 Mo) ;
- la taille d'un film en définition normale est d'environ 700 Mo ;
- la taille d'une musique est d'environ 3 Mo ;
- les registres mémoires d'un processeur contiennent 4 ou 8 octets (32 bits ou 64 bits) selon l'architecture du processeur.

Dans ce cours, on appellera *petit espace mémoire* un espace mémoire de la taille d'un registre de processeur. C'est à dire, 32 ou 64 bits selon l'architecture. Un *registre de processeur* est un emplacement mémoire interne au processeur. Il sert à réaliser des calculs et des transferts de données.

1.3 Programme

On appelle *programme* l'ensemble des instructions que l'unité centrale doit exécuter.

Voici un exemple de programme :

```
print( "Bonjour" )
print( "L'ordinateur va compter jusqu'à 4 :" )
for i in range(4):
    print( "numero : " + str(i+1) )
```

On appelle *langage* les règles d'écriture de ces instructions.

Il existe beaucoup de langages :

- le langage machine ;
- l'assembleur ;
- le C ;
- le python ;
- le java ;
- etc ...

Le *langage machine* est un langage lisible directement par la machine. Il est très peu compréhensible pour un humain normal.

Par exemple, voici un exemple de langage machine :

```
457f 464c 0102 0001 0000 0000 0000 0000
0002 003e 0001 0000 04c0 0040 0000 0000
0040 0000 0000 0000 1168 0000 0000 0000
0000 0000 0040 0038 0009 0040 001f 001c
0006 0000 0005 0000 0040 0000 0000 0000
0040 0040 0000 0000 0040 0040 0000 0000
01f8 0000 0000 0000 01f8 0000 0000 0000
0008 0000 0000 0000 0003 0000 0004 0000
0238 0000 0000 0000 0238 0040 0000 0000
0238 0040 0000 0000 001c 0000 0000 0000
...
```

L'*Assembleur* et le *C* sont deux langages qui sont compréhensibles par l'utilisateur, mais qui ne le sont pas par la machine.

Voici un exemple de programme écrit en assembleur :

```
    .file   "essai.c"
    .section    .rodata
.LC0:
    .string "Bonjour"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
```

```
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident    "GCC: (GNU) 4.6.0"
    .section    .note.GNU-stack,"",@progbits
```

Voici un exemple de programme écrit en C :

```
#include <stdio.h>
int main(){
    printf("Bonjour\n");
    return 0;
}
```

Pour que l'ordinateur puisse réaliser les instructions de ces deux programmes, il faut convertir ces programmes en langage machine à l'aide d'une autre programme que l'on appelle *compilateur*.

Le *python* est un langage compréhensible par l'utilisateur, mais pas par la machine.
Voici un exemple de programme écrit en python :

```
print( "Bonjour" )
```

Pour exécuter les instructions d'un langage écrit en python, il faut utiliser un interpréteur. Un interpréteur est un programme qui lit les instructions écrites en python, les convertit en instructions processeur et fait exécuter ces instructions au fur et à mesure de la lecture du code source.

1.4 L'algorithmique

Un *algorithme* est une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème.

Le mot algorithme vient du nom latinisé du mathématicien arabo-musulman Al-Khawarizmi, surnommé "le père de l'algèbre". Le domaine qui étudie les algorithmes est appelé l'algorithmique.

L'*algorithmique* est l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithmes, c'est-à-dire de processus systématiques de résolution d'un problème permettant de décrire les étapes vers le résultat.

2 Le langage de programmation Python

Nous allons expliquer dans cette partie, comment fonctionne le langage Python. Pour cela, nous allons proposer un modèle qui explique comment Python utilise et modifie la mémoire de l'ordinateur. Le modèle utilisé est cohérent et décrit assez bien ce qui se passe dans l'ordinateur. Cependant, il ne colle pas exactement aux spécifications du langage. La description de l'utilisation de la mémoire ne correspond pas exactement à celle de python. En fait, elle correspond plus au fonctionnement des langages C, Java, C++, etc ... Ce choix a été fait pour plusieurs raisons :

- ce modèle est plus simple à expliquer,
- le comportement du programme observé par l'utilisateur est quasiment identique à celui obtenu par le modèle,
- dans le monde professionnel, c'est le java, le C et les C++ qui sont les langages les plus utilisés. Le python est utilisé généralement pour l'administration système des ordinateurs.

2.1 Écrire et exécuter un programme écrit en Python

Un programme est un texte contenant des instructions écrite de haut en bas. Chaque ligne du texte correspond à une instruction. Sauf cas particulier, les instructions DOIVENT être alignées verticalement à gauche.

Voici un exemple de code :

```
a = 5                   # <-- Première instruction
a = a / 2               # <-- Deuxième instruction
print( a )              # <-- Troisième instruction
```

Voici un exemple de code qui n'est pas valide car les instructions ne sont pas alignées verticalement à gauche :

```
a = 5                   # <-- Première instruction
   a = a / 2            # <-- Deuxième instruction
print( a )              # <-- Troisième instruction
```

Sauf cas particulier, l'ordinateur exécute les instructions d'un programme de haut en bas et de gauche à droite.

Même si les instructions ressembles à des formules mathématiques, elles n'ont pas la même signification !

Lorsque une instruction est exécutée, l'ordinateur n'affiche généralement rien à l'écran. L'absence d'affichage n'est pas synonyme d'inactivité pour l'ordinateur.

Ainsi, si vous exécutez les instructions suivantes :

```
3 + 2
5 / 7
2 - 6
```

l'ordinateur réalise toutes les instructions écrites (ici ce sont des calculs arithmétiques) mais n'affiche rien à l'écran.

Si vous voulez afficher le résultat d'une instruction, il faut écrire

```
print( INSTRUCTION )
```

Par exemple, le programme

```
3 + 2
print( 5 - 7 )
2 / 6
```

affichera -2.

2.2 Les variables et les affectations

2.2.1 Les variables

Nous avons vu que le rôle de l'ordinateur était de manipuler la mémoire. En python, la manipulation de la mémoire se fait à l'aide des variables.

Une *variable* est le nom que l'on donne à un petit espace mémoire. Nous rappelons que dans ce document un *petit espace mémoire* désigne un espace mémoire de 32 ou 64 bits seulement.

On peut créer autant de variables que l'on souhaite et leurs donner le nom que l'on veut à condition qu'elles ne comportent que des caractères alphanumériques.

Pour créer une variable de nom `NOM_DE_LA_VARIABLE`, il faut écrire l'instruction suivante :

```
NOM_DE_LA_VARIABLE = INSTRUCTION
```

où `INSTRUCTION` est une autre instruction python.

Lorsque l'on écrit ce code, l'ordinateur alloue un petit espace mémoire (32 ou 64 bits), le réserve pour le programme et lui donne comme nom `NOM_DE_LA_VARIABLE`. Ensuite, il exécute l'instruction `INSTRUCTION` et le résultat de cette instruction est enregistré dans l'espace mémoire associé à `NOM_DE_LA_VARIABLE`.

Par exemple, le programme

```
tmp = 5
```

alloue un petit espace mémoire et lui donne le nom `tmp`. Ensuite, il évalue 5, le résultat est la valeur 5 et il l'enregistre dans la mémoire associée à `tmp`.

Attention, l'affectation n'a pas le même sens que le signe = en mathématique. Ainsi,

```
5 = a
```

n'est pas une instruction valide et n'a aucune signification !

Une fois la variable `NOM_DE_LA_VARIABLE` crée, l'instruction

```
NOM_DE_LA_VARIABLE = INSTRUCTION
```

modifie le contenu de `NOM_DE_LA_VARIABLE` en le remplaçant par le résultat de l'instruction `INSTRUCTION`.

Ainsi, le programme

```
a = 3
a = 5
```

Alloue un espace mémoire pour a et enregistre dans cet espace la valeur 3, puis remplace cette valeur par 5.

2.2.2 Afficher le contenu d'une variable

Il est possible d'afficher le contenu d'une variable sur le terminal. Il suffit d'utiliser l'instruction :

```
print( NOM_DE_LA_VARIABLE )
```

Par exemple, le programme :

```
a = 4
print( a )
```

Alloue un petit espace mémoire et l'appelle `a`, met ensuite 4 dans l'espace associé à a et enfin, affiche le contenu de la variable a. Il apparaît alors à l'écran le chiffre 4.

2.2.3 Copier le contenu de la mémoire

A chaque fois que l'on utilise une variable, l'ordinateur l'évalue en récupérant le contenu de l'espace mémoire qui lui est associé. Ainsi, pour copier le contenu de l'espace mémoire, il suffit d'utiliser l'affectation vu précédemment de la façon suivante :

```
NOM_DE_LA_VARIABLE_DE_DESTINATION = NOM_DE_LA_VARIABLE_A_COPIER
```

Par exemple, dans le programme suivant :

```
b = 5
a = 1
b = a
print( b )
```

Lorsque l'ordinateur exécute la troisième instruction, l'ordinateur alloue un espace mémoire pour la variable b et rempli cette espace avec le contenu de l'espace mémoire associé à `a`. Le programme finit son exécution en affichant la valeur 1.

2.3 Les entiers, les réels, les booléens et les opérations arithmétiques

Les *booléens* sont représentés pas les mots clé True et False. Les opération possible sont

```
not ( INSTRUCTION )
INSTRUCTION and INSTRUCTION
INSTRUCTION or INSTRUCTION
INSTRUCTION == INSTRUCTION
```

qui sont respectivement la négation, le et et le ou booléen mathématique.

Nous rappelons que le non, le et, le ou et l'équivalence mathématique sont définis par :

	True	False
not	False	True

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

$\Leftrightarrow$ (==)	True	False
True	True	False
False	False	True

Par exemple,

```
bool = False
print( True and bool )
```

affiche `False`

Les *entiers* sont des nombres écrits sans virgule.

Les *réels* sont des nombres écrits avec une virgule. La virgule en python est le ".".

Par exemple, dans le programme

```
a = 4
b = 4.0
```

la mémoire associée à `a` contient l'entier 4 et la mémoire associée à `b` contient le réel 4.0. Dans cet exemple, les contenus des espace mémoires de `a` et de `b` sont complètement différents.

Voici les différentes opérations qui existent sur les entiers et les réels.

- L'addition : +
- La soustraction : −
- La multiplication : ∗
- La division réel ou euclidienne : /
- l'égalité : ==
- la non égalité : ! =

Pour toutes ces opérations, le résultat est un entier si tous les termes sont des entiers. Le résultat est un réel si au moins un des deux termes est un réel.

Enfin, pour la division, si les deux termes sont réels, alors le résultat est le quotient de la division euclidienne des 2 termes. Si un des deux termes est un réel, alors le résultat est la division dans les réels des deux termes.

Par exemple, le programme

```
print( 1 + 2 )
print( 1.0 + 2 )
print( 3 / 2 )
print( 3.0 / 2 )
```

affiche respectivement 3, 3.0, 1, 1.5.

2.4 Les fonctions

Une fonction est une portion de code représentant un sous-programme qui effectue une tâche relativement indépendamment du reste du programme.

Une fonction possède une entrée (les arguments, ou les paramètres) qui constitue les données que l'on donne au départ à la fonction. La fonction exécute ensuite un travail sur ces données, puis retourne éventuellement une valeur appelée sortie du programme.

Toutes les variables définies dans une fonction sont crées dans une espace mémoire indépendant du reste du programme qui est propre à la fonction. L'espace mémoire associée à ces variables est libéré à la fin de l'exécution de la fonction. Les fonctions ne peuvent pas faire appelle à des variables situées en dehors de la définition de la fonction. Le nom des variables définies à l'intérieur de la fonction peut être identique au nom d'autres variables du programme sans affecter leurs états.

En python, on défini les fonctions de la manière suivante :

```
def NOM_FONCTION( PARAMETRE1, PARAMETRE2, ... ):
    INSTRUCTIONS_DE_LA_FONCTION
```

C'est l'indentation qui permet de déterminer les blocs `INSTRUCTIONS_DE_LA_FONCTION`. Toutes les lignes qui à la fois

- commencent avec strictement plus d'espace et de tabulations que la ligne `def NOM_FONCTION( PARAMETRE1, PARA`
- sont situées juste après `def NOM_FONCTION( PARAMETRE1, PARAMETRE2, ... )`

sont dans `INSTRUCTIONS_DE_LA_fONCTION`.

Pour exécuter le code d'une fonction, il suffit d'écrire dans le programme :

```
NOM_FONCTION( VALEUR1, VALEUR2, ... )
```

Par exemple, le programme ;

```
def afficher_somme( val1, val2 ):
    resultat = val1 + val2
    print( resultat )

afficher_somme( 3,5 )
afficher_somme( 2,1 )
```

afficher 8 puis 3.

Voici le détail de l'exécution de ce programme :

- ligne 1 - 3 : Le programme définit la fonction `afficher_somme`.
- ligne 5 : le programme exécute `afficher_somme(3,5)` :
  - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `afficher_somme`, il alloue dans cet espace mémoire deux petits espaces mémoires et les nomme `val1` et `val2`. Il rempli ensuite le contenu de l'espace mémoire associé à `val1` par le premier entier donnée en paramètre : 3. Puis, il remplit le contenu de l'espace mémoire associé à `val2` par le second entier donné en paramètre : 5.
  - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `resultat`. Il évalue ensuite `val1 + val2`, le contenu de `val1` dans la mémoire de `afficher_somme` vaut 3, celui de `val2` vaut 5, l'évaluation de `val1 + val2` vaut donc 8. Le programme rempli enfin la mémoire de `resultat` par 8.
  - ligne 3 : Le programme affiche le contenu de la mémoire associée à `resultat`, il affiche donc 8.
  - fin de la fonction : la fonction libère son espace mémoire, les données qui y sont contenues sont perdues.
- ligne 6 : le programme exécute `afficher_somme(1,2)` :
  - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `afficher_somme`, il alloue dans cet espace mémoire deux petits espaces mémoires et les nomme `val1` et `val2`. Il rempli ensuite le contenu de l'espace mémoire associé à `val1` par le premier entier donnée en paramètre : 1. Puis, il remplit le contenu de l'espace mémoire associé à `val2` par le second entier donné en paramètre : 2.
  - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `resultat`. Il évalue ensuite `val1 + val2`, le contenu de `val1` dans la mémoire de

`afficher_somme` vaut 1, celui de `val2` vaut 2, l'évaluation de `val1 + val2` vaut donc 3. Le programme rempli enfin la mémoire de `resultat` par 3.
- ligne 3 : Le programme affiche le contenu de la mémoire associée à `resultat`, il affiche donc 3.
- fin de la fonction : la fonction libère son espace mémoire, les données qui y sont contenues sont perdues.

Il est possible à tout moment, dans la fonction, d'interrompre l'exécution de fonction et de faire retourner une valeur par la fonction. Il suffit d'utiliser l'instruction :

```
return VALEUR_SORTIE
```

Cette instruction arête l'exécution de la fonction et renvoie la valeur `VALEUR_SORTIE`. Ainsi, lors de l'exécution de `NOM_FONCTION( VALERU1, VALEUR2, ...)`, l'ordinateur substitue `NOM_FONCTION( VALERU1, VALEUR2, ...)` par l'évaluation de `VALEUR_SORTIE`.

Par exemple, le programme,

```
def f(x):
    a = 4
    return a*x

a = 100
b = f(3)
print( a )
print( b )
```

affiche 100, puis 3.

Voici le détail de l'exécution de ce programme :
- ligne 1 - 3 : Le programme définit la fonction `f`.
- ligne 5 : le programme alloue une petit espace mémoire, le nomme `a` et le rempli avec 100.
- ligne 6 :
  - le programme alloue une petit espace mémoire, le nomme `b`
  - le programme exécute `f(3)` :
    - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `f`, il alloue dans cet espace mémoire un petite espace mémoire et le nomme `x`. Il rempli ensuite cet espace avec la valeur donnée en paramètre : 3.
    - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `a` et le rempli par 4.
    - ligne 3 : Le programme évalue a*x, le contenu de `a` dans l'espace mémoire de `f` vaut 4, celui de `x` vaut 3. L'évaluation vaut donc 12. L'instruction `return` interrompt l'exécution de la fonction, renvoie 12 et libère l'espace mémoire de la fonction. Les données qui y étaient stockées sont définitivement perdues.
  - le programme récupère la valeur de retour : 12
  - le programme met 12 dans l'espace mémoire associée à `b`
- ligne 7 : Le programme affiche sur le terminal le contenu de `a` dans la mémoire du programme : il affiche donc 100.
- ligne 8 : Le programme affiche sur le terminal le contenu de `b` dans la mémoire du programme : il affiche donc 12.

Exercice 1. *Prévoyez l'exécution du programme suivant :*

```
def mystere( a, b ):
    a = a + 2
    b = b + 3
    tmp = a * b
    return tmp
    print( "Calcul de la soustraction" )
    tmp = a - b
```

```
a = 3
b = 5
tmp = mystere(a,b)
print( a )
print( b )
print( tmp )
```

2.5 Les sauts conditionnels

Les *sauts conditionnels* permettent à l'ordinateur de choisir entre deux jeux d'instructions, les instructions qu'il va exécuter en fonction d'un test préalable.

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
else :
    LISTE_D_INSTRUCTIONS_2
```

Dans ce code, l'ordinateur évalue l'expression `TEST`. Si cette expression est vraie, alors il exécute les instructions de `LISTE_D_INSTRUCTIONS_1`, sinon il exécute les instructions de `LISTE_D_INSTRUCTION_2`.

C'est l'indentation qui permet de déterminer les blocs `LISTE_D_INSTRUCTION_1` et `LISTE_D_INSTRUCTION_2`.

Toutes les lignes qui à la fois

- commencent avec strictement plus d'espaces et de tabulations que la ligne `if TEST`;
- sont situées juste après `if TEST`

sont dans `LISTE_D_INSTRUCTIONS_1`. Toutes les lignes qui à la fois

- commencent avec strictement plus d'espaces et de tabulations que la ligne `else`;
- sont situées juste après `else :`

sont dans `LISTE_D_INSTRUCTIONS_1`. Enfin, les lignes `if` et `else` doivent avoir le même niveau d'indentation.

Par exemple, le programme

```
a = 2
b = 2
if a == b :
    print("a est égal à b")
    print("Comme prévu !")
else :
    print("a est différent de b")
    print( "Ce code ne devrait pas s'exécuter" )
print( "Fin du programme" )
```

affiche

```
a es égal à b
Comme prévu !
Fin du programme
```

Il est possible d'avoir plusieurs sauts conditionnels imbriquées les un dans les autres. Par exemple, le programme

```
if 1 != 2 :
    print( "1 est différent de 2" )
    if 2 == 4 :
        print( "Ne devrait pas s'afficher")
    else:
```

```
        print( "2 est différents de 4" )
    print( "Fin du bloc d'instruction" )
else:
    print( "Ne devrait pas s'afficher" )
print( "Fin du programme" )
```

affiche

```
1 est différent de 2
2 est différent de 4
Fin du bloc d'instruction
Fin du programme
```

Lorsque `LISTE_D_INSTRUCTIONS_2` ne contient pas d'instruction, on écrit l'instruction `pass` à la place.

On obtient ainsi :

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
else :
    pass
```

On peut aussi ne pas écrire la partie avec le `else`. Ainsi, le précédent code est équivalant à

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
```

2.6 Les boucles

En programmation, il est utile de pouvoir écrire des programmes qui répètent un certain nombre de fois le même code.

Dans les langages de programmation impératif, ce sont les boucles "for" et les boucles "while" qui permettent de faire exécuter plusieurs fois le même code.

La syntaxe d'une boucle "for" est la suivante :

```
for VARIABLE in range( NOMBRE_DE_REPETITION ):
    LISTE_D_INSTRUCTIONS
```

Il faut savoir qu'en python, `range( NOMBRE_DE_REPETITION )` code la liste `[0,1,2,3,... ,NOMBRE_DE_REPETITION-1]`. Dans les lignes de codes précédentes, l'ordinateur commence par allouer un petit espace mémoire ayant pour nom `VARIABLE`. Ensuite, pour chaque élément e de cette liste, dans l'ordre de la liste, le programme réalise les tâches suivantes :

- il affecte à la variable `VARIABLE` l'élément e ;
- il exécute les instructions de `LISTE_D_INSTRUCTIONS`.

A la fin, le programme aura exécuté `NOMBRE_DE_REPETITION` fois les instructions de `LISTE_D_INSTRUCTION`.

Par exemple, le programme suivant compte de 0 à 999 :

```
for compteur in range(1000):
    print( compteur )
```

La syntaxe d'une boucle while est la suivante :

```
while CONDITION :
    LISTE_D_INSTRUCTIONS
```

Lorsque l'interpréteur python rencontre ces instructions, il commence par évaluer l'expression CONDITION. Si le résultat est Vrai, alors il exécute `LISTE_D_INSTRUCTIONS` puis recommence tout au début, à l'évaluation de l'expression `CONDITION`. Si le résultat est Faux, le programme n'exécute pas `LISTE_D_INSTRUCTIONS` et l'exécution du `while` se termine.

Par exemple, le programme suivant compte de 0 à 999 :

```
compteur = 0
while compteur < 1000 :
    print( compteur )
    compteur = compteur + 1
```

2.7 Les listes et les tableaux

Une liste d'éléments est une suite ordonnée d'éléments. En python, il est possible de coder une liste contenant `N` éléments à l'aide de l'instruction :

```
[ ELEMENT1, ELEMENT2, ..., ELEMENTN ]
```

Lors de l'exécution de cette instruction, l'interpréteur python alloue `N` petit espace mémoire successif dans lequel il place les éléments, `ELEMENT1`, ..., `ELEMENT2`. Puis, l'interpréteur python, substitue `[ELEMENT1, ..., ELEMENTN]` par une référence à la liste crée. Une référence à la liste est l'adresse mémoire où se trouve la liste. Une adresse est une donnée de la taille d'un petit espace mémoire. Elle peut donc être stockée dans un petit espace mémoire associée à une variable.

Ainsi, si l'on écrit le programme

```
l = [1,3,2]
```

alors python alloue 3 petits espaces mémoires consécutifs dans la mémoire. Ensuite il substitue `[1,3,2]` par l'adresse mémoire ou se trouve la liste crée. L'interpréteur alloue alors un petit espace mémoire et l'appelle `l`. Enfin, il enregistre dans le petit espace mémoire l'adresse où se trouve la liste précédemment crée.

En python, l'espace mémoire alloué pour les listes ne dépend pas du contexte des fonctions. Ainsi, même si la création de la liste est faite à l'intérieur d'une fonction, la mémoire crée par une liste ne dépend pas de la fonction. Elle n'est donc pas détruite quand la fonction se termine.

La mémoire est détruite automatiquement si elle n'est plus utilisée, c'est à dire si il n'existe plus de référence vers la liste en question.

Par exemple,

```
def creer_tableau( a, b, c ):
    res = [ a,b,c ]
    return res
a = 10
v = creer_tableau( 1,5,6 )
```

Il est possible d'accéder au contenu d'un élément de la liste à l'aide de la commande :

```
LISTE[K]
```

où `LISTE` est le nom de la variable contenant un référence vers une liste et `K` un entier.

Ici, `LISTE[K]` est l'espace mémoire numéro K de la liste qui est située à l'adresse écrite dans le petit espace mémoire associé à `LISTE`. En python les espace mémoire sont numérotés de 0 à la taille de la liste moins 1.

Autrement dit, `LISTE[K]` veux dire :
Dans le petit espace mémoire associe à `LISTE`, vous trouverez une adresse. À cette adresse se trouve une succession de petit espace mémoire. `LISTE[K]` est l'espace mémoire numéro `K`.

Par exemple,

```
l = [-1,20,3]
print( l[0] )
print( l[2] )
```

affiche -1 puis 3 sur le terminal.

Il est possible d'affecter des valeurs à `LISTE[k]` avec l'opérateur d'affectation = en écrivant :

```
LISTE[K] = VALEUR
```

Par exemple,

```
l = [1,3,10]
print( l )
l[1] = 50
print( l )
```

affichera successivement `[1,3,10]` puis `[1,50,10]`

En python il est possible de connaître la taille d'une liste en utilisant la commande ;

```
len( LISTE )
```

où `LISTE` est une variable contenant une référence vers une liste.

Par exemple,

```
l = [1,5,2]
print( len( l ) )
```

affichera 3 à l'exécution.

Il est possible de concaténer deux listes à l'aide de l'addition.

Ainsi, la variable b du programme suivant

```
B = [1,2,3] + [4,5,6]
```

contiendra une référence vers la nouvelle liste `[1,2,3,4,5,6]`.

Il est possible d'ajouter une nouvelle valeur à la fin de la liste en écrivant :

```
LISTE.append( VALEUR )
```

où `LISTE` est une variable contenant une référence vers une liste et `VALEUR` la valeur que l'on souhaite ajouter à la fin de la liste.

Il est possible de créer une liste à n éléments à l'aide de la commande :

```
[ EXPRESSION(i) for i in range(n)]
```

où `EXPRESSION(i)` est une fonction qui qui dépend de `i`. La liste ainsi obtenue est une liste à n éléments dont l'élément numéro i est l'évaluation de l'expression `EXPRESSION(i)`.

Par exemple,

```
n = 5
l = [ 0 for i in rang(5)]
print(l)
```

affiche `[0,0,0,0,0]`

De même,

```
n = 5
l = [ 2*i for i in range(5)]
print(l)
```

affiche [0,2,4,6,8].

2.8 Les tuples

Les tuples sont identiques aux listes sauf que
– l'on utilise des parenthèses au lieu des crochets ;
– les tuples sont immuables, c'est à dire qu'ils ne sont pas modifiables.
Par exemple,

```
b = (1,4,3)
print( b )
print( b[1] )
```

affiche `(1,4,3)` puis `4`.

2.9 Le texte et les chaînes de caractères

En python, il est possible de travailler avec des chaînes de caractères. Pour utiliser une chaîne de caractères, il suffit d'écrire :

```
"LA CHAINE DE CARACTERES"
```

Si l'on veut garder un référence vers une chaîne de caractère, il suffit d'écrire :

```
l = "LA CHAINE DE CARACTERE"
```

Dans ce cas, python alloue un petit espace mémoire pour la variable `l`. Ensuite il alloue une espace mémoire plus grand pour enregistrer la chaîne de caractères `"LA CHAINE DE CARACTERES"`, enfin il enregistre dans l'espace mémoire de `l` une référence vers l'espace mémoire de la chaîne de caractères.

EXEMPLE

En python les chaînes de caractères sont immuables et uniques ! Le caractère immuable veut dire que l'on ne peut pas modifier une chaîne de caractères une fois qu'elle a été crée. Le caractère unique veux dire que si deux chaînes de caractères sont identiques, alors elles auront le même espace mémoire. Par exemple, à la fin du programme,

```
c1 = "Bonjour"
c2 = "Bonjour"
```

`c1` et `c2` contiendront la même référence vers un espace mémoire qui contient la chaîne de caractères "Bonjour".

À contrario, les listes et les tuples ne sont pas uniques. Ainsi le programme suivant

```
c1 = [1,2,3]
c2 = [1,2,3]
```

créera deux variable `c1` et `c2` contenant des références différentes vers des espaces mémoires différents ayant un contenu identique.

Pour savoir si deux variables contiennent des références à des objets identiques, il suffit d'utiliser l'opérateur `is`.

Par exemple,

```
a = "Bonjour"
b = "Bonjour"
print( a is b )
a = (1,2,3)
b = (1,2,3)
print( a is b )
```

affiche

```
True
False
```

3 Le codage des entiers, des réels et du texte par l'ordinateur

3.1 Écriture des entiers en base quelconque

Soit $\mathbb{N}$ l'ensemble des entiers naturels et $\mathbb{Z}$ l'ensemble des entiers relatifs.

Theorem 1. *Soit b un entier strictement positif. Tout entier relatif n s'écrit de manière unique sous la forme $n = b.q + r$ où est q un entier relatif et r un entier de $[0, b-1]$. On appelle quotient de la division euclidienne de n par b l'entier q. On appelle reste de la division euclidienne de n par b l'entier r.*

En python, le quotient de la division euclidienne de a par b s'écrit `a/b`. De même, le reste de la division euclidienne de a par b s'écrit `a%b`.

Theorem 2. *Soit b un entier. Tout entier s'écrit de manière unique sous la forme*

$$\sum_{k\geq 0} a_k.b^k$$

où $a_k \in [0, b-1]$ sont des entiers presque tous nuls (il existe un entier M tel que, quel que soit l'entier $k > M$, $a_k = 0$).

Soit n un entier et $(a_0, a_1, a_2, ...)$ la suite qui lui est associée par le théorème précédent. On dit que est $\overline{(a_0, a_1, a_2, ...)}^2$ est l'écriture en base b de n. Lorsque l'on écrit n en base b on écrit généralement les chiffres de poids forts (les chiffres associées à une puissance de b la plus grosse) en premier et on termine par les chiffres de poids faibles. Quand c'est le cas, les chiffres seront écrit sans les parenthèses et sans les virgules. Ainsi : $\overline{(a_0, a_1, a_2, ..., a_m)}^2 = \overline{a_m a_{m-1} \dots a_0}^b$.

Par exemple, pour $b = 2$,

$$\begin{array}{rcl} 142 & = & 0\times 2^0 + 1\times 2^1 + 1\times 2^2 + 1\times 2^3 + 0\times 2^4 + 0\times 2^4 + 0\times 2^5 + 0\times 2^6 + 1\times 2^7 \\ & = & \overline{100001110}^2 \\ & = & \overline{(011100001)}^2 \end{array}$$

3.2 Taille d'un nombre

Soit $\overline{(a_0, a_1, a_2, \dots, a_{n-1})}^b$ un entier représenté en base b avec n chiffres.

L'entier $n = \overline{(a_0, a_1, a_2, \dots, a_{n-1})}^b$ est donc compris entre 0 et $b^n - 1$. En effet,

$$0 = \sum_{k=0}^{n-1} 0.b^k \leq n \leq \sum_{k=0}^{n-1} (b-1).b^k$$

or

$$\sum_{k=0}^{n-1} (b-1).b^k \;=\; (b-1).\sum_{k=0}^{n-1} b^k \;=\; (b-1).\frac{b^n-1}{b-1} \;=\; b^n - 1.$$

De même, un entier n, codé en base b, contiendra $\left\lceil \frac{\ln(n)}{\ln(b)} \right\rceil$ chiffres. Dans la littérature, on note par $log_b(n)$, le réél $\frac{\ln(n)}{\ln(b)}$.

3.3 Changement de bases

Theorem 3. *Soit $n = \overline{(a_0, a_1, a_2, \ldots)}^b$ un nombre représenté en base b, alors*

$$a_0 = n\%b \qquad et \qquad \overline{(a_1, a_2, \ldots)}^b = \left\lfloor \frac{n}{b} \right\rfloor$$

Démonstration. Par définition, $n = \sum_{k\geq 0} a_k b^k$. On en déduit donc que

$$n \quad = \quad b.\sum_{k\geq 0} a_{k+1} b^k + a_0 \quad = \quad b.\overline{(a_1, a_2, \ldots)}^b + a_0$$

, ce qui, avec le théorème 1, termine la preuve. □

Theorem 4. *Soit n un entier et soit $\overline{(a_0, a_1, a_2, \ldots)}^b$ son écriture en base b. Alors, pour tout entier $k \geq 0$ on a :*

$$a_k = \left\lfloor \frac{n}{b^k} \right\rfloor \%b$$

où $u\%v$ est le reste de la division euclidienne de u par v.

Démonstration. On sait que $n = \sum_{k\geq 0} a_k.b^k$. On en déduit donc que $\frac{n}{b^j} = \sum_{k\geq 0} a_{k+j} b^k + \frac{\sum_{k=0}^{k=j-1} a_k b^k}{b^j}$. Or, on a vu à la section précédente que $\sum_{k=0}^{k=j-1} a_k b^k \leq b^j - 1$, on en déduit donc que $\left\lfloor \frac{n}{b^j} \right\rfloor = \sum_{k\geq 0} a_{k+j} b^k$ et que $\left\lfloor n/b^j \right\rfloor \%b = a_j$

□

Exercice 2. *Soit n un entier et b un entier strictement positif. Montrer que, pour tout entier $k \geq 0$:*

$$\left\lfloor \frac{n}{b^k} \right\rfloor = \phi^k(n)$$

où $\phi(u) = \lfloor \frac{u}{b} \rfloor$ et ϕ^k est la puissance k-ème de ϕ pour la composée des fonctions.

Pourquoi cette remarque a-t-elle un intérêt en programmation ?

3.4 Codage des entiers dans un ordinateur

En informatique, les entiers sont codés avec des 0 et des 1. En fonction du type de l'entier (signé ou non signé), différents algorithmes sont utilisés pour convertir l'entier en une suite de 0 et de 1. Nous allons maintenant voir ces algorithmes.

3.4.1 Codage des entiers non signés

Un entier non signé est un entier positif. Les entiers positifs, sont codés dans un ordinateur en base 2 à l'aide d'un nombre fini de chiffres appelés bits. Un entier codé sur n bits est donc un entier compris entre 0 et $2^n - 1$.

Par exemple, un entier codé sur 32 bits est compris entre 0 et $2^{32} - 1 = 429496729$. Un entier codé sur 64 bits est compris entre 0 et $2^{64} - 1 = 18446744073709551615$.

Cela explique pourquoi, les ordinateurs 32 bits (qui ont un processeur qui ne peut manipuler que des entiers codés sur 32 bits) ne peuvent pas avoir plus de 4 go de mémoire vive. En effet, ils ne peuvent pas adresser plus de 4 go de mémoire, c'est à dire, donner un identifiant à chaque élément atomique de la mémoire.

Ainsi, les systèmes d'exploitation 32 bits, qui n'utilisent que des jeux d'instructions processeur pour ordinateur 32 bits, ne peuvent pas accepter des mémoires de plus de 4 go, même si le système d'exploitation est installé sur un ordinateur 64 bits. Ce problème concerne notamment le système d'exploitation Windows XP.

3.4.2 Codage des entiers signés

En informatique, les entiers signés sont des entiers qui peuvent être positifs ou négatifs.

Une implémentation naïve des entiers signés seraient d'utiliser le premier bits pour coder le signe, et les autres bits pour coder la valeur absolu de l'entier. Ainsi, 11010 coderait -6 et 01010 coderait 6.

Cette première solution ne convient pas. En effet, dans un processeur, l'addition des entiers codés sur n bits est codée uniquement pour les entiers positifs et se défini mathématiquement par :

$$\text{addition}_n(a,b) := (a+b) \bmod 2^n.$$

Si on utilise le codage précédent, alors, l'addition sur 5 bits de -6 par 6 sur 5 bits donnerait en binaire :

$$\begin{array}{rr} & 10110 \\ + & 00110 \\ \hline & 11000 \end{array}$$

qui vaudrait -8.

En fait, la solution, consiste à coder sur n bits les entiers signés compris entre -2^{n-1} et $2^{n-1}-1$ par leur représentant canonique dans la classe de congruence modulo 2^n.

C'est à dire, de coder un entier a compris entre $-2^{n-1}+1$ et $2^{n-1}-1$ par :

$$\text{complet2}_n(a) := a \bmod 2^n.$$

Ainsi,

$$\begin{array}{rcl} \text{addition}_n(\text{complement2}_n(a), \text{complement2}_n(b)) & = & (a \bmod 2^n) + (b \bmod 2^n) \\ & = & (a+b) \bmod 2^n \\ & = & \text{complement2}(a+b). \end{array}$$

Le complément à 2 peut se réécrire de la façon suivante : pour tout entier a compris entre -2^n+1 et 2^n-1,

$$\text{comlement2}_n := \begin{cases} \overline{a}^2 & \text{si } a \geq 0 \\ \overline{2^n + a}^2 & \text{si } a < 0 \end{cases}$$

Par exemple, sur 5 bits, l'entier 6 est codé par 00110 et l'entier -6 est codé par l'écriture binaire de $-6+2^5 = 26$ qui est 11010.

En remarquant que $\overline{2^n + a}^2 = \overline{2^n - 1 + a + 1}^2 = \overline{11\ldots1}^2 + \overline{a}^2 + 1$, on peut réécrire cet algorithme par :

$$\text{comlement2}_n := \begin{cases} \overline{a}^2 & \text{si } a \geq 0 \\ \text{complement}(\overline{a}^2) + 1 & \text{si } a < 0 \end{cases}$$

où $compelment(a)$ est l'écriture binaire de a où les 0 ont été remplacés par des 1 et les 1 par des 0.

Si l'on reprend l'exemple précédent, on obtient,

$$\text{complement2}_n(-6) = complement(\overline{00110}^2) + 1 = \overline{11001}^2 + 1 = \overline{11010}^2$$

Maintenant, l'addition sur 5 bits de -6 avec 6 donne :

$$\begin{array}{rr} & 11010 \\ + & 00110 \\ \hline & 00000 \end{array}$$

qui vaut bien 0.

3.5 Codage des réels dans un ordinateur

Les nombres réels sont codés à l'aide de la norme IEE 754, mis au point en 1985.

Soit n un entier, $e > 0$ et $m > 0$ trois entiers tels que $e + m + 1 = n$.

Dans la norme IEE 754, le codage $b = (b_{n-1}, b_{n-2}, \ldots, b_2, b_1, b_0)$ représente un réel r. Le bit de poids fort b_{n-1} code le signe s de r.

$$s = \begin{cases} 1 & \text{si } b_{n-1} = 0 \\ -1 & \text{sinon.} \end{cases}$$

Les e bits suivants, codent des entiers positifs codés en binaire (b_{e+m-1} est le bit de poids fort). On appellera $exposant = b_{e+m-1}, \ldots, b_m$ ces e bits. Les m derniers bits servent à coder un réel compris entre 0 et 2. On appellera $mantisse = b_{m-1}, \ldots, b_0$ ces m bits. L'entier r est alors défini par

$$r = \begin{cases} s \,.\, \infty & \text{si } exposant = 1, \ldots, 1 \text{ et } \overline{mantisse}^2 = 0; \\ \text{NaNs} & \text{si } exposant = 1, \ldots, 1 \text{ et } \overline{mantisse}^2 \neq 0; \\ s \,.\, \overline{l, mantisse}^2 \,.\, 2^{\overline{exposant}^2 - decalage} & \text{sinon,} \end{cases}$$

où $decalage = 2^{e-1} - 1$, où l est définit par

$$l = \begin{cases} 0 & \text{si } \overline{exposant}^2 = 0; \\ 1 & \text{sinon} \end{cases}$$

, où $\overline{l, mantisse}^2$ signifie

$$\overline{l, mantisse}^2 = l + \sum_{i=1}^{m} b_{m-i}.2^{-i}$$

et NaNs est l'acronyme de "not a number" qui veut dire pas un nombre en anglais.

Lorsque l'exposant vaut 0, les réels codés sont appelés nombres dénormalisés. Lorsque l'exposant est différent de 0 les entiers sont appelés nombres normalisés. A l'exception du codage de 0, seul les réels sont codés à l'aide du codage normalisé. Les nombres dénormalisés ne sont généralement pas utilisés, ils sont utilisés uniquement pour coder des nombres très petits.

La norme IEEE 754, fournit ainsi 2 formats de nombres :

– les *nombres simples précisions* sur 32 bits en définissant $n = 32$, $e = 8$ et $m = 23$;

– les *nombres doubles précisions* sur 64 bits où $n = 64$, $e = 11$, $m = 52$.

3.6 Codage du texte dans un ordinateur

3.6.1 Le codage ASCII

L'American Standard Code for Information Interchange est un standard qui explique comment coder certains caractères.

L'ASCII définit 128 caractères numérotés de 0 à 127. Ils sont codés sur 7 bit de 0000000 à 1111111.

Dans les ordinateurs, les processeurs travaillent généralement avec des données dont la quantité de bits est un multiple de 8. C'est pourquoi, les caractères ASCII sont codés sur 8 bits en fixant la valeur du premier bit à 0.

Voici la table des caractères du standard ASCII (la première ligne et la première colonne donne le code en hexadécimal des caractères) :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Par exemple, le caractère A est codé en hexadécimale par 41, ce qui donne en binaire, sur 8 bits : 01000001.

Les caractères ayant un numéro compris entre 0 et 32 sont des caractères spéciaux qui ne sont généralement pas imprimables.

On peut remarquer que le codage des caractères alphabétiques minuscules (resp. majuscules) forment un intervalle et l'ordre alphabétique est préservé par leur codage.

De même, le codage des caractères numériques forment un intervalle et l'ordre numérique est préservé par leur codage.

3.7 Le codage ISO 8859-1

Le standard `ISO 8859-1`, aussi appelé `latin-1` est une extension du codage `ASCII`. Ce codage utilise le bit non utilisé du codage ASCII pour ajouter des caractères latins comme les caractères accentués.

Vois la table des caractères de l'`ISO 8859-1` :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
A	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	-	®	¯
B	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

4 Algorithmes de recherche et de tri

4.1 Recherche d'un élément dans une tableau

Dans cette section nous allons nous intéresser à la recherche d'un élément dans un tableau. Pour cela, nous allons restreindre le problème aux tableaux d'entiers, et nous allons chercher à savoir si un entier e donné se trouve dans un tableau t contenant uniquement des entiers. Le tableau t peut éventuellement contenir plusieurs fois le même entier.

4.1.1 Recherche linéaire

En toute généralité, si le tableau t n'est pas ordonné, il est nécessaire de passer en revu tous les éléments du tableau pour savoir si l'entier e se trouve dans t. On réalise cela à l'aide du programme suivant, qui renvoie l'index du premier élément e dans t si e est dans t et `None` sinon.

```
def recherche_lineaire( t, e ):
    for i in range( len(t) ):
        if t[i] == e :
            return i
    return None
```

Le temps d'exécution de ce programme peut être évalué en comptant le nombre d'instructions réalisées. Si l'on considère que modifier la valeur de i dans la boucle `for` compte aussi pour 1 instruction, et que les opérations usuelles =, ==, +, -, *, etc ... comptent pour une seule opération, alors l'algorithme précédent a réalisé, dans le pire des cas : $2.n$ opérations, où n est le nombre d'éléments du tableau t.

4.1.2 Recherche dichotomique

Il est possible d'améliorer la recherche d'un élément d'un tableau si le tableau est trié par ordre croissant. On utilise, pour cela, une recherche dichotomique.

L'idée de la recherche dichotomique, consiste à comparer l'entier e à chercher avec l'entier situé au milieu du tableau. Si e est plus petit ou égal, alors on va chercher l'entier uniquement dans la partie gauche du tableau. Si e est strictement plus grand, alors on va chercher l'entier uniquement dans la partie droite. On s'arrête lorsqu'il ne reste plus qu'un seul élément à inspecter. Si ce dernier élément n'est pas égal à e, alors c'est que le tableau t ne contient pas e.

Cela nous donne ainsi l'algorithme suivant :

```
def recherche_dichotomique( t, e ):
    if len(t) == 0 :
        return None
    i = 0
    j = len( t ) - 1
    while( i != j ):
        milieu = (i+j)/2
        if t[milieu] < e:
            i = milieu + 1
        else :
            j = milieu
    if( t[i] == e ):
        return i
    else:
        return None
```

Nous allons maintenant estimer le nombre d'instructions $C(t)$ nécessaires à cet algorithme pour chercher un élément dans un tableau t de taille n.

Si le tableau n'est pas vide, alors le programme réalise 5 opérations en dehors de la boucle. De même, le programme réalise entre 6 et 7 opérations à chaque tour de boucle. Ainsi, si le tableau n'est pas vide, le nombre d'instructions, réalisées par le programme, peut être encadré par

$$5+6.b(t) \leq C(t) \leq 5+7.b(t)$$

où $b(t)$ est le nombre de fois que l'on a exécuté la boucle `while` de l'algorithme.

Soit a le plus grand entier tel que $2^{a-1} < n \leq 2^a$, où n est la taille du tableau. Soit u_k le nombre d'éléments situés entre i et j inclus à la fin de l'exécution de la $k^{\text{ème}}$ boucle. On a alors

$$2^{a-k-1} < u_k \leq 2^{a-k}. \tag{1}$$

En effet, il suffit de faire une récurrence sur k pour le démontrer.

Le programme s'arrête au moment où $i = j$, c'est à dire lorsque $u_k = 1$. Cette condition se réalise quand $k = a$ (voir Équation 1). Comme $2^{a-1} < n \leq 2^a$ On en déduit que $b(t) = a = \left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$.

L'algorithme réalise donc entre $5+6.\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ et $5+7.\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ opérations.

Lorsque n est grand, ce nouvel algorithme est bien plus efficace que la recherche linéaire. Ceci peut être observé à l'aide de la figure 4.

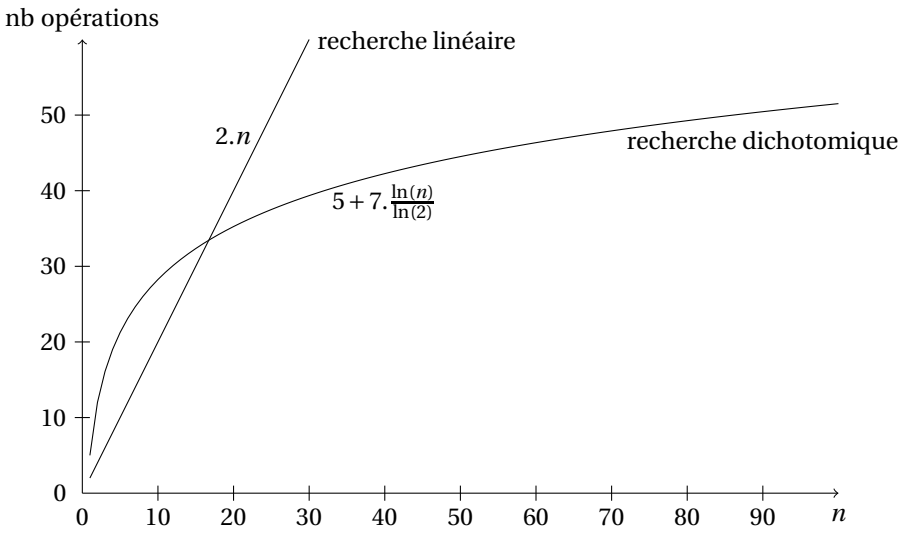


FIGURE 4 – Comparaison du nombre d'opérations réalisées par les différents algorithmes de recherche (n est la taille du tableau).

4.2 Algorithme de Tri

Soit T un tableau non ordonné d'entiers, pouvant contenir plusieurs fois le même entier.
Nous allons nous intéresser aux algorithmes qui permettent de trier ce tableau.

4.2.1 Tri à bulles

L'algorithme de tri à bulles consiste à répéter, jusqu'à ce que le tableau soit trié, une étape de tri qui consiste à parcourir les éléments du tableaux et à échanger deux éléments successifs s'ils ne sont pas dans l'ordre croissant.

Le programme Python de tri est le suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def etape_tri_bulle( t ):
    est_trie = True
    for i in range( len(t)-1 ):
        if( t[i]>t[i+1] ):
            echanger( t, i, i+1 )
            est_trie = False
    return est_trie

def tri_bulle( t ):
    est_trie = False
    while not( est_trie ):
        est_trie = etape_tri_bulle( t )
```

Si le tableau t est déjà trié, cet algorithme peut se terminer très rapidement, en réalisant juste une itération de `etape_tri_bulle(t)`.

Cependant, les performances chutent lorsque le tableau est quelconque. On peut évaluer le nombre d'opérations effectuées par cet algorithme, dans le pire cas. Soit n le nombre d'éléments contenus dans un tableau. Commençons par évaluer le nombre d'opérations effectuées par `etape_tri_bulle` : cette fonction réalise 1 opération en dehors de la boucle `for` et entre 3 et 8 opérations par boucle à l'intérieur de la boucle `for`. Le programme `etape_tri_bulle` réalise donc entre $1+3(n-1)$ et $1+8(n-1)$ opérations. Étudions maintenant la complexité de `tri_bulle`. À la fin du $k^{\text{ème}}$ appel de `etape_tri_bulle(t)`, le tableau t contient les k plus grand éléments de t dans les k dernières cases, triées par ordre croissant. Ainsi, à la $n-1^{\text{ème}}$ exécution de `etape_tri_bulle(t)` les $n-1$ plus grands éléments sont triés et situés à la fin, donc le tableau est complètement trié. On en déduit que `etape_tri_bulle(t)` est exécuté, dans le pire cas, $n-1$ fois. Ce pire cas est atteint pour le tableau $[n, n-1, n-2, \dots, 1]$. Ainsi, dans le pire cas, l'algorithme réalise entre $1+(n-1).(1+1+3.(n-1)) = 3.n^2-5.n+3$ et $1+(n-1).(1+1+8(n-1)) = 8.n^2-14.n+7$ opérations.

Ce nombre d'opération est un polynôme de degrés 2 en n. On dit que cet algorithme est de complexité quadratique dans le pire cas.

Dans le meilleur cas, si le tableau est déjà trié, cet algorithme réalise $1+3(n-1)$ opérations qui est un polynôme de degrés 1 en n. On dit alors que ce programme est de complexité linéaire dans le meilleur des cas.

Quand on parle de complexité d'un programme, on parle généralement de sa complexité dans le pire des cas. Ainsi, l'algorithme de tri à bulle est de complexité quadratique.

4.2.2 Tri par insertion ou le tri du joueur de carte

Le tri par insertion, appelé aussi tri du joueur de carte, est le tri réalisé généralement par les joueurs de cartes. Habituellement, les joueurs de cartes trient ses cartes au fur et à mesure qu'ils les obtiennent

en insérant la nouvelle carte à la bonne position parmi les cartes déjà triées. Cet algorithme s'écrit en Python de la façon suivante :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def inserer( t, i ):
    j = i
    while j > 0 and t[j] < t[j-1] :
        echanger( t, j, j-1 )
        j = j - 1

def tri_insertion( t ):
    for i in range( len(t) ):
        inserer( t, i )
```

Nous allons étudier la complexité dans le pire cas de `tri_insertion`. Soit $c(t,i)$ le nombre d'opérations réalisées par `inserer(t,i)`. Dans le pire cas, la boucle de `inserer(t,i)` est exécuté i fois. Ainsi, dans le pire cas, $c(t,i) = 1+10i$. Lors de l'exécution de `tri_insertion(t)`, ce pire cas est atteint pour toutes les exécutions de `inserer(t,i)` lorsque le tableau t initial vaut [n, n-1, n-2, ..., 3, 2, 1].

Ainsi, dans le pire cas, le programme `tri_insertion(t)` réalise :

$$\sum_{i=0}^{n-1}(1+c(t,i)) = \sum_{i=0}^{n-1}(1+1+10i) = 2n+10\frac{n(n-1)}{2} = 5n^2-3n$$

opérations (voir propriété 2 pour le détail des calculs).

Cet algorithme est donc un algorithme quadratique.

4.2.3 Tri par sélection

Le tri par sélection est un tri qui consiste à itérer n étapes où la $k^{\text{ème}}$ étape consiste à chercher le $k^{\text{ème}}$ plus petit élément et à l'échanger avec l'élément situé en position k dans le tableau.

On obtient ainsi l'algorithme suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def recherche_min( t, i ):
    position_min = i
    for k in range( i+1, len(t) ):
        if( t[ k ] < t[ position_min ] ):
            position_min = k
    return position_min

def tri_insertion( t ):
    for i in range( len(t) ):
        echanger( t, i, recherche_min( t, i ) )
```

Si l'on note par $c(t,i)$ la complexité de `recherche_min(t,i)`, alors on obtient l'encadrement $2+2(n-i-1) \leq c(t,i) \leq 2+3(n-i-1)$. Le nombre d'opérations réalisées par le programme `tri_insertion` est donc compris antre $\sum_{i=0}^{n-1}(6+2(n-i-1)) = 6+2\frac{n(n-1)}{2}$ et $\sum_{i=0}^{n-1}(6n+3(n-i-1)) = 6n+3\frac{n(n-1)}{2}$.

L'algorithme de tri par sélection est donc quadratique.

4.2.4 Tri rapide (hors programme)

Le principe du tri rapide consiste à choisir un élément p du tableau, appelé pivot, puis à trier le tableau en mettant les éléments plus petits que p à gauche de p et les éléments plus grands que p à droites de p, puis a recommencer le processus, jusqu'à ce que le tableau soit entièrement trié, sur le tableau de gauche d'une part et sur le tableau de droite d'autre part.

On obtient ainsi l'algorithme suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def partitioner( t, min, max ):
    pivot = max
        i = min
    j = max - 1
    while( i<j ):
        if t[i] < t[pivot] :
             i = i + 1
        else:
            echanger( t, i, j )
            j = j - 1
    if t[i] < t[pivot] :
        echanger( t, i+1, pivot )
        pivot = i+1
    else :
        echanger( t, i, pivot )
        pivot = i
    return pivot

def tri_rapide_recursif( t, i, j ):
    if( i < j):
        pivot = partitioner( t, i, j )
        tri_rapide_recursif( t, i, pivot-1 )
        tri_rapide_recursif( t, pivot+1, j )

def tri_rapide( t ):
    tri_rapide_recursif( t, 0, len(t)-1 )
```

La complexité de ce programme est quadratique dans le pire cas. Cet algorithme est tout de même très utilisé car il est rapide dans beaucoup de cas usuels.

4.2.5 Tri fusion (hors programme)

Le tri fusion consiste à couper le tableau en 2, à trier le tableau de gauche à l'aider d'un autre tri fusion, à trier le tableau de droite avec le même algorithme, puis à fusionner les deux tableaux.

On obtient ainsi l'algorithme suivant :

```
def fusion( t, min, milieu, max ):
    tmp = []
    i = min
    j = milieu + 1
    while i <= milieu or j <= max :
        if i > milieu :
            tmp.append( t[j] )
```

```
            j = j + 1
        elif j > max :
            tmp.append( t[i] )
            i = i + 1
        elif t[i] < t[j] :
            tmp.append( t[i] )
            i = i + 1
        else :
            tmp.append( t[j] )
            j = j + 1
    for i in range( len(tmp) ):
        t[min+i] = tmp[i]

def tri_fusion_recursif( t, i, j ):
    if( i < j ):
        milieu = (i+j)/2
        tri_fusion_recursif( t, i, milieu )
        tri_fusion_recursif( t, milieu+1, j )
        fusion( t, i, milieu, j )

def tri_fusion( t ):
    tri_fusion_recursif( t, 0, len(t)-1 )
```

La complexité de ce programme est majoré par $c.n.\ln(n)$, où n est la taille du tableau et c un constante. Pour des tableaux contenant beaucoup d'éléments, cet algorithme est le le plus rapide parmi les algorithmes précédement présentés.

A Prérequis Mathématiques

A.1 Quelques symboles classiques

On définit les symboles $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{R}_+, \mathbb{R}_-, \mathbb{C}$ par :

- $\mathbb{N} = \{1, 2, \ldots\}$ est l'ensemble des entiers positifs ;
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ est l'ensemble des entiers relatifs ;
- $\mathbb{R}$ est l'ensemble des réels ;
- $\mathbb{R}_+$ est l'ensemble des réels positifs ;
- $\mathbb{R}_-$ est l'ensemble des réels négatifs ;
- $\mathbb{C}$ est l'ensemble des nombres complexes.

Soit A un sous-ensemble de $\mathbb{C}$, alors A^* est l'ensemble A privé de l'élément neutre 0. Par exemple, $\mathbb{R}_+^*$ est l'ensemble des réèls strictement positifs.

A.2 Les sommes et les produits

Soit m et n deux entiers et f une fonction de $\mathbb{N}$ dans $\mathbb{R}$. On définit par $\sum_{i=m}^{n} f(i)$ la somme définie par :

$$\begin{cases} \sum_{i=m}^{n} f(i) = 0 & \text{si } n < m \\ \sum_{i=m}^{n} f(i) = \sum_{i=m}^{n-1} f(i) + f(n) & \text{sinon.} \end{cases}$$

L'expression $\sum_{i=m}^{n} f(i)$ correspond à la somme :

$$f(m) + f(m+1) + f(m+2) + \cdots + f(n-1) + f(n).$$

Par exemple,

$$\sum_{i=3}^{6} i^2 + 2 \times i \quad = \quad (3^2 + 2 \times 3) \quad + \quad (4^2 + 2 \times 4) \quad + \quad (5^2 + 2 \times 5) \quad + \quad (6^2 + 2 \times 6).$$

On définit par $\prod_{i=m}^{n} f(i)$ le produit défini par :

$$\begin{cases} \prod_{i=m}^{n} f(i) = 1 & \text{si } n < m \\ \prod_{i=m}^{n} f(i) = \prod_{i=m}^{n-1} f(i) \times f(n) & \text{sinon.} \end{cases}$$

L'expression $\prod_{i=m}^{n} f(i)$ correspond au produit :

$$f(m) \times f(m+1) \times f(m+2) \times \cdots \times f(n-1) \times f(n).$$

Par exemple,

$$\prod_{i=3}^{6} i^2 + 2 \times i \quad = \quad (3^2 + 2 \times 3) \quad \times \quad (4^2 + 2 \times 4) \quad \times \quad (5^2 + 2 \times 5) \quad \times \quad (6^2 + 2 \times 6).$$

Pour tout entier $n \geq 0$, on défini la factorielle de n par :

$$n! = \prod_{i=1}^{n} i$$

Par exemple $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ et $0! = 1$.

A.3 Les binômiaux

Soit m et n deux entiers tels que $m \leq n$. Les binomiaux C_n^m sont des entiers définis par :

$$C_n^m = \frac{n!}{m!(n-m)!}.$$

Les binomiaux vérifient la prorpriété suivante :

$$C_n^m = C_{n-1}^{m-1} + C_{n-1}^m \tag{2}$$

En effet,

$$C_{n-1}^{m-1} + C_{n-1}^m = \frac{(n-1)!}{(m-1)!(n-m)!} + \frac{(n-1)!}{m!(n-1-m)!} = \frac{(n-1)!m + (n-1)!(n-m)}{m!(n-m)!} = \frac{n!}{m!(n-m)!} = C_n^m.$$

On aime bien généralement représenter cette propriéte sous la forme d'un triangle appelé triangle de Pascal qui est présenté à la figure 5.

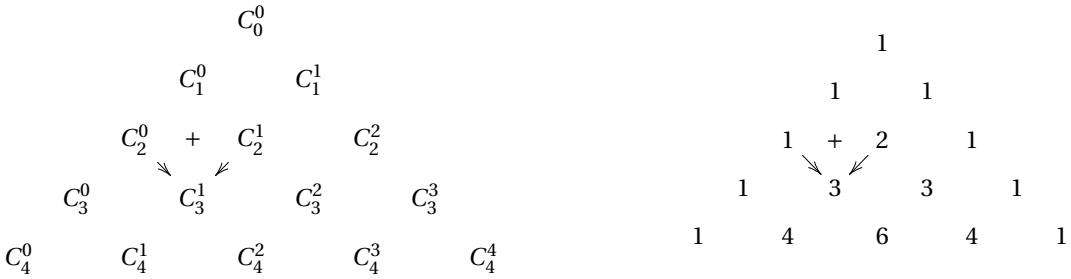


FIGURE 5 – Le triangle de Pascal

Les binomiaux jouent un rôle important en mathématiques car il apparaissent dans le développement de

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k} \tag{3}$$

Cette formule se démontre par récurence sur n de la façon suivante :

- Si $n = 0$, alors, $(a+b)^0 = 1$ et $C_0^0 a^0 b^0 = 1$, donc la formule est vrai ;
- Soit $M > 0$ un entier. Supposons que la formule soit vraie pour $n \leq M$, est-elle vrai pour $n = M+1$? Develloppons $(a+b)^{M+1}$:

$$\begin{aligned}
(a+b)^{M+1} &= (a+b).(a+b)^M = (a+b)\sum_{k=0}^{M} C_M^k a^k b^{M-k} \\
&= \sum_{k=0}^{M} C_M^k a^{k+1} b^{M-k} + \sum_{k=0}^{M} C_M^k a^k b^{M+1-k} \\
&= a^{M+1} + \sum_{k=1}^{M} \left(C_M^{k-1} + C_M^k\right).a^k b^{M+1-k} + b^{M+1}
\end{aligned}$$

D'après l'équation 2, on sait que $C_{M+1}^k = C_M^{k-1} + C_M^k$, on a alors :

$$\begin{aligned}
(a+b)^{M+1} &= a^{M+1} + \sum_{k=1}^{M} C_{M+1}^k.a^k b^{M+1-k} + b^{m+1} \\
&= \sum_{k=0}^{M+1} C_{M+1}^k a^k b^{M+1-k}.
\end{aligned}$$

Ce qui finit de démontrer par récurence la formule 3.

A.4 Calcul de sommes classiques

Proposition 1. *Soit n un entier et r un réel différent de 1, alors*

$$\sum_{k=0}^{n} r^k = \frac{1-r^{n+1}}{1-r}$$

Démonstration.

$$\begin{array}{rcl}(1-r).\sum_{k=0}^{n} r^k & = & (1+r+r^2+\cdots+r^n)-(r+r^2+\cdots+r^n+r^{n+1}) \\ & = & 1-r^{n+1}\end{array}$$

□

Proposition 2. *Soit n un entier, alors*

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \qquad \sum_{k=0}^{n} k^2 = \frac{(2n+1)(n+1)n}{6} \qquad \sum_{k=0}^{n} k^3 = \frac{n^2(n+1)}{4}$$

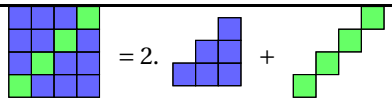
Démonstration. La formule $S_n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ est célèbre. Elle a été démontré par Carl Friedrich Gauss de la façon suivante :

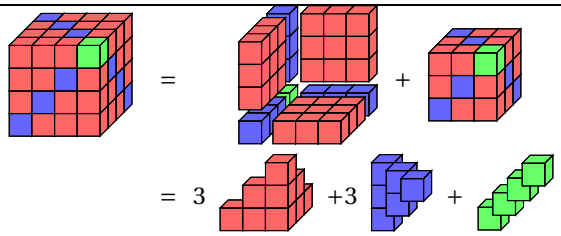
$$\begin{array}{rcccccccccc} S_n & = & 1 & + & 2 & + & \ldots & + & n-1 & + & n \\ S_n & = & n & + & n-1 & + & \ldots & + & 2 & + & 1 \\ \hline 2.S_n & = & n+1 & + & n+1 & + & \ldots & + & n+1 & + & n+1 \end{array}$$

$$2.S_n = n(n+1)$$

Cependant, cette preuve ne se généralise pas pour le calcul des sommes $\sum_k k^d$ où d est un entier quelconque.

En fait, cette identité remarquable est un cas particulier d'une identité géométrique qui est la suivante :

dimension 1	dimension 2
	
$n+1 = \sum_{k=0}^{n} k^0$	$(n+1)^2 = 2\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$

dimension 3	dimension 4
	?
$(n+1)^3 = 3\sum_{k=0}^{n} k^2 + 3\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$	$(n+1)^4 = 4\sum_{k=0}^{n} k^3 + 6\sum_{k=0}^{n} k^2 + 4\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$

En regardant les formules des tableaux précédents, on reconnait les premières lignes du triangle de Pascal, présenté à la figure 5. On peut donc facilement en déduire une formule conjecturale pour la dimension 4, (voir tableau précédent), et on peut traduire ces observations mathématiquement par, pour tout entier n et toute dimension d,

$$(n+1)^d = \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i. \tag{4}$$

Il ne reste plus qu'à démontrer cette dernière identité en procédant de la façon suivante : On sait que $(k+1)^d = \sum_{i=0}^{d} C_d^i k^i$. Ainsi, si l'on somme cette dernière identité pour k allant de 0 à n, on obtient que :

$$\sum_{k=0}^{n} (k+1)^d = \sum_{k=0}^{n} \sum_{i=0}^{d} C_d^i k^i = \sum_{i=0}^{d} \sum_{k=0}^{n} C_d^i k^i = \sum_{i=0}^{d} C_d^i \sum_{k=0}^{n} k^i.$$

Si l'on extrait le terme $i = d$ de la somme du dernier terme, on obtient :

$$\sum_{k=0}^{n} (k+1)^d = \sum_{k=0}^{n} k^d + \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i$$

qui se simplifie de part et d'autre de l'égalité pour donner :

$$(n+1)^d = \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i.$$

On peut maintenant utiliser l'équation 4 pour déterminer récursivement les sommes cherchées :

$$(n+1) = \sum_{k=0}^{n} k^0 \qquad \Longrightarrow \qquad \sum_{k=0}^{n} k^0 = n+1$$

$$(n+1)^2 = 2.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \qquad \Longrightarrow \qquad \sum_{k=0}^{n} k = \frac{(n+1)^2 - (n+1)}{2}$$

$$(n+1)^3 = 3.\sum_{k=0}^{n} k^2 + 3.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \qquad \Longrightarrow \qquad \sum_{k=0}^{n} k^2 = \frac{1}{3}\left((n+1)^3 - 3.\frac{n(n+1)}{2} - (n+1)\right)$$

$$(n+1)^4 = 4.\sum_{k=0}^{n} k^3 + 6.\sum_{k=0}^{n} k^2 + 4.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \qquad \Longrightarrow \qquad \sum_{k=0}^{n} k^3 = \frac{(n+1)^4 - 6.\frac{(2n+1)(n+1)n}{6} - 4.\frac{n(n+1)}{2} - (n+1)}{4}.$$

□