1ère NSI — Thème 6: Algorithmique

Algorithmique & Mise au Point de Programmes

Lycée Fustel de Coulanges, Massy

Marc Biver, février 2024, *v0.1*

Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.

Table des matières

1 Point d'étape – où est-on / où va-t-on ?

1.1 Ce qu'on a couvert jusqu'à présent

- Rudiments de l'architecture physique d'un ordinateur – le modèle de Von Neumann.
- Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
- Introduction à Python, et plus spécifiquement :
  - Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
  - Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
  - Un type dit "construit" – les listes.
- Un peu de théorie : la représentation des types et valeurs de base en machine (les entiers naturels et relatifs, les réels, les alphanumériques).
- Un peu plus de théorie : introduction à la logique booléenne (que l'on n'a couverte que très rapidement – on y reviendra en fin d'année).
- Un retour à la pratique : le traitement de données en table, la manipulation de fichiers dans Python, et des types de données plus complexes — les dictionnaires, les listes de dictionnaires...

1.2 Ce dont on va parler dans ce nouveau chapitre

On a donc fait des "sauts" de la théorie vers la pratique, puis vers la théorie de nouveau — pour finalement atterrir ici, dans ce chapitre sur l'algorithmique qui se trouve presque exactement au "milieu" de cet axe théorie–pratique.

Comme je vous l'ai dit à plusieurs reprises dans les parties précédentes – et spécialement dans celle portant sur le traitement de données en table, l'algorithmique, vous en faites déjà : je vous pose des problèmes par le biais de notebooks Jupyter dans Capytale v2 et vous réfléchissez à comment les résoudre. Dit autrement, *vous concevez des algorithmes.*

Ce que l'on va faire dans ce nouveau chapitre c'est formaliser cette démarche, la structurer, puis étudier quelques exemples d'algorithmes beaucoup plus avancés que ce que l'on a vu jusqu'à présent.

Spécifiquement, on va parcourir le chemin suivant :

- Introduction à l'algorithmique – étapes de conception et de rédaction d'un algorithme ;
- Pratique : tests d'algorithmes / de programmes ;
- Preuve d'algorithmes ;
- Complexité d'algorithmes ;
- Etude de certains algorithmes spécifiques :
  - Algorithmes de tri - par sélection, par insertion ;
  - Algorithme de recherche dichotomique dans un tableau ;
  - Algorithmes gloutons ;
  - Algorithmes des k plus proches voisins – algorithme d'apprentissage, "machine learning".

— Pratique : mise au point de programmes ; programmation défensive.

1.3 Comment on va procéder

Comme dit plus haut, on est ici à la frontière entre la théorie et la pratique – on va donc avoir un fonctionnement hybride en classe :

— **Prise de notes essentielle** — vous commencez à connaître les cours que je vous fournis, il sont *très* longs. Ils doivent vous servir de référence, vous permettre surtout de bien revoir les corrections d'exercices, mais votre savoir, lui, doit venir de votre prise de notes ;

— Plusieurs exercices sur papier qu'on fera en classe et dont il sera très important que vous gardiez une trace ;

— En parallèle et en complément, quelques applications / exercices sur machine.

2 Introduction à la conception d'algorithmes

Quelques questions pour commencer...

→*Qu'est-ce qu'un algorithme (indice : c'est constitué de trois parties) ?*

Un algorithme est une ***suite finie d'instructions*** *permettant de* ***résoudre un problème****. Il est structuré en trois parties :*

a. L'entrée des données ;
b. Le traitement des données ;
c. La sortie des données

→*Parmi les éléments suivants, qu'est-ce qui est un algorithme, qu'est-ce qui n'en est pas ?*

a. Une recette de gâteau au chocolat ;
b. La liste des présidents de la V^{ème} République ;
c. Les règles du jeu d'échecs ;
d. Les règles à appliquer pour résoudre une équation du 2nd degré ;
e. Les instructions de montage d'un meuble Ikea.

a. Oui c'en est un – ingrédients en entrée, étapes de confection, gâteau en sortie. C'est bien la résolution d'un problème (en l'occurrence "comment fabriquer un gâteau au chocolat ?").
b. Non ce n'en est pas un – c'est une liste d'informations, pas des étapes à suivre.
c. Ce n'en est pas un non plus – ce n'est qu'une liste de principes. En revanche on pourrait les utiliser pour mettre en œuvre un algorithme qui joue aux échecs (et résoud le problème "comment jouer – et gagner – une partie d'échecs ?).
d. Oui c'en est un – c'est même écrit dans la description ("résoudre").
e. Oui c'en est un également.

Définition:

L'algorithmique est :

— La conception (et la production) d'algorithmes ;
— Leur étude – leur analyse, la mesure de leur fiabilité (est-ce qu'il répond vraiment au problème posé ?) et de leur efficacité (est-ce qu'il le fait en un temps acceptable ?).

Nous allons dans ce cours nous intéresser à ces deux composantes, en commençant, dans ce chapitre, par la première dont les étapes peuvent se résumer ainsi :

1. Énoncé d'un problème à résoudre ;
2. Spécification de l'algorithme – nom, entrées, sorties ;
3. Explicitation de la démarche en pseudo-code – description du traitement des données qui va permettre de résoudre le problème ;
4. Traduction en langage de programmation.

La 1ère et la 4ème étape sont à la marge de notre propos ici :

— L'énoncé d'un problème à résoudre par le biais d'un programme informatique est une activité à part entière (souvent appelée "expression de besoin" dans le monde professionnel). Vous y avez un petit peu touché dans le cadre de vos projets, mais dans l'ensemble, dans le contexte de la NSI, les problèmes vous sont posés – et votre rôle consiste à savoir les résoudre ;
— La traduction en langage de programmation, que dans le contexte de la NSI nous réalisons en Python, a fait l'objet d'un pan du cours distinct ; nous allons évidemment y revenir en partie ici, mais la syntaxe Python n'est pas l'objet de notre étude ici.

2.1 Écrire un algorithme : spécification

Avant d'écrire un algorithme il faut bien définir ce que l'on veut faire et à partir de quoi ; il s'agit de donner **une spécification au problème**. Pour cela on doit :

— Donner un nom explicite à l'algorithme — *par exemple* `CuissonGateauChocolat` ;
— Décrire les conditions d'utilisation de l'algorithme, les données qu'il attend en entrée et les conditions dans lesquelles il va pouvoir être exécuté, sa ***précondition*** — *par exemple* *"Beurre et Lait non périmés"* :
— De même, décrire le résultat attendu, sa ***postcondition***, la nature des données renvoyées et à quoi elles correspondent — *par exemple* *"gâteau rond, moelleux, et succulent"*.

Cette étape de spécification est fondamentale – c'est en quelque sorte la "carte d'identité" de notre algorithme, ce qui va permettre à quelqu'un qui ne le connait pas de le comprendre sans avoir besoin de lire son code. On va donc la transcrire dans notre code Python en tête de la fonction lui correspondant – et c'est exactement ce que je vous demande de faire dans vos projets.

Méthode:

La transcription de la spécification d'un algorithme en tête de la fonction Python lui correspondant s'appelle "***le docstring***" ou "***la documentation***" de la fonction. Il est inscrit entre deux séries de trois apostrophes – par exemple :

```python
def MaxNombre(n1, n2):
    '''
    Fonction dont les paramètres sont entiers ou réels.
    Elle renvoie la plus grande de ces deux valeurs ou, en cas
    d'égalité, la première valeur.
    '''
    if n1 < n2:
        return n2
    else:
        return n1
```

Exercice 1: Rédaction d'une spécification de fonction

Considérez la fonction suivante :

```python
def Fonction(n1, n2, n3):
    if n1 < n2 < n3 or n3 < n2 < n1:
        return n2
    elif n1 < n3 < n2 or n2 < n3 < n1:
        return n3
    elif n2 < n1 < n3 or n3 < n1 < n2:
        return n1
    elif n1 == n2 and n2 == n3:
        return n1
    else:
        return None
```

Est-ce que ce qu'elle fait est clair d'entrée de jeu ?
Rédigez la docstring de cette fonction pour remédier à cela.

On aura noté deux problèmes ici – l'absence de docstring, mais également l'absence d'un nom explicite à la fonction ("Fonction", ce n'est franchement pas génial...). Remédions à tout cela :

```python
def Mediane(n1, n2, n3):
    '''
    Fonction qui prend en entrée trois valeurs numériques (int ou float).
    Elle renvoie la médiane de ces trois valeurs quand celle-ci existe.
    Si elle n'existe pas (deux valeurs égales, la troisième différente),
    elle renvoie None.
    Par ex: Mediane(0, 2, 1) renverra 1;
    Mediane(1, 1, 1) renverra 1;
    Mediane(0, 2, 2) renverra None.
    '''
    if n1 < n2 < n3 or n3 < n2 < n1:
        return n2
    elif n1 < n3 < n2 or n2 < n3 < n1:
        return n3
    elif n2 < n1 < n3 or n3 < n1 < n2:
        return n1
    elif n1 == n2 and n2 == n3:
        return n1
    else:
        return None
```

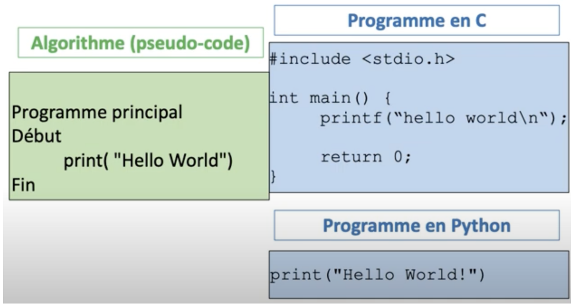
Vous remarquerez l'inclusion d'exemples dans la docstring – il ne faut surtout pas hésiter à y recourir, c'est ce qu'il y a de plus parlant pour quelqu'un qui découvre votre code !

2.2 Ecrire un algorithme : pseudo-code

Si l'on se réfère aux étapes listées plus haut, on en est maintenant au moment où l'on sait *ce que va réaliser* notre programme, *ce qu'il va prendre en entrée*, et *ce qu'il va retourner en sortie*. Il s'agit à présent d'expliciter le ***comment*** – quelles sont les étapes qui vont être effectuées pour résoudre le problème ? Quel traitement va-t-on appliquer aux données en entrée pour produire les données en sortie ?

On a déjà utilisé à de multiples reprises le pseudo-code dans ce cours – donc (*en théorie*) vous devriez déjà être convaincus de son intérêt et savoir l'utiliser. Nous allons donc passer directement à quelques exercices d'application – en rappelant tout de même au préalable les principes et règles suivants :

a. Le pseudo-code est une façon de décrire un algorithme pour qu'il soit compréhensible "entre humains".
b. Le pseudo-code est indépendant du langage de programmation – un algorithme convenablement écrit devrait en théorie pouvoir être implémenté aussi aisément en Python qu'en C ou qu'en JavaScript[1]. A titre d'exemple, voici un "Hello World" en deux langages de programmation distincts, mais partant du même pseudo-code :



c. Conséquence : les règles de syntaxe de pseudo-code sont inspirées des éléments communs à la plupart des langages de programmation.
d. Il n'y a pas de pseudo-code universel – le seul principe à respecter, c'est que les règles de syntaxe appliquées soient bien définies, comprises et partagées par tous·tes celles et ceux qui seront amené·e·s à lire les algorithmes.

Ce dernier point implique qu'il y ait quand même une ossature de règles minimales dans un contexte donné – comme pour ce cours par exemple :

Règles pseudo-code 1ère NSI:

a. On spécifie explicitement en début d'algorithme les entrées attendues et les sorties prévues ;
b. On utilise une flèche vers la gauche ("←" ou "◁") pour affecter des variables ;
c. On utilise l'indentation pour délimiter les fonctions, les conditions, les boucles...[a]
d. On explicite la fin de toute structure (fonction, condition, boucle...) débutée ;
e. On n'hésite pas à inclure des commentaires pour expliquer les étapes – et dans ce cas, on les préfixe d'une flèche vers la droite ("→" ou "▷")
f. ... et c'est tout !

a. Ici on triche un peu puisque l'indentation est spécifique à Python – mais pas tant que ça puisque cela restera compréhensible même pour une implémentation dans un autre langage.

1. C'est évidemment un peu simpliste d'écrire ceci ainsi, mais en théorie le principe est vrai : lorsque vous rédigez un algorithme en pseudo-code, vous devriez ne pas avoir de langage de programmation spécifique en tête.

Pour illustrer ces principes, voici le pseudo-code d'une fonction (qu'on a déjà vue dans un chapitre précédent, d'ailleurs) prenant une liste de réels positifs en entrée et qui en renvoie le maximum :

Entrée: *liste* dont tous les éléments $\in \mathbb{R}+$
Sortie: $Max \in \mathbb{R}+$

```
fonction TrouveMax(liste)
    Max ← 0
    pour tout Element de liste faire
        si Element > Max alors          ▷ On a trouvé un nouveau max
            Max ← Element
        fin si
    fin pour
    retourner Max
fin fonction
```

Exercice 2: Rédaction d'algorithmes en pseudo-code

En appliquant les principes énoncés ci-dessus, rédigez les algorithmes suivants :

a. Un algorithme qui prend deux nombres en entrée et affiche leur somme.

b. Un algorithme qui calcule la somme des N premiers entiers naturels.

c. Un algorithme qui génère les N premiers termes de la séquence de Fibonacci. (rappel : c'est une suite de nombres dont les deux premiers sont 0 et 1 et dont chaque élément est la somme des deux précédents – donc : 0, 1, 1, 2, 3, 5, 8, 13, etc...).

d. Un algorithme qui vérifie si une chaîne de caractères est un palindrome (se lit de la même manière dans les deux sens).

e. [**BONUS**] — Supposez que vous avez une liste Lst de nombres réels ordonnée (c'est-à-dire où $Lst[0] \leq Lst[1] \leq (...) \leq Lst[n]$) et que vous cherchez à savoir si elle contient un nombre N en particulier (la conclusion sera donc booléenne - Vrai ou Faux). Rédigez puis comparez deux algorithmes :
   - Un qui fait une recherche "normale", en commençant à un bout et en parcourant toute la liste ;
   - Un autre qui effectue une recherche dite "dichotomique" :
     - Il regarde le milieu de la liste :
       - S'il est supérieur à N il élimine la partie supérieure de la liste ;
       - S'il est inférieur à N il élimine la partie inférieure de la liste ;
     - Il recommence l'opération jusqu'à soit trouver N soit aboutir à une impossibilité.

   Que peut-on dire des efficacités relatives de ces deux algorithmes ?

Il n'y a jamais (ou rarement) une solution algorithmique unique à un problème – ce qui suit n'est donc que des propositions de solution (qui sont, évidemment, valides – mais pas uniques).

a. ***Entrée:*** *deux nombres a et $b \in \mathbb{R}$*
Sortie: $a + b$

```
fonction Somme(a, b)
```

```
    Resultat ← a + b
    Afficher Resultat
fin fonction
```

b. **Entrée:** $N \in \mathbb{N}$
Sortie: *la somme des N premiers entiers naturels*

```
fonction SommeEntiers(N)
    Resultat ← 0                          ▷ On initialise le résultat à 0
    pour i allant de 0 à N faire
        Resultat ← Resultat + i
    fin pour
    retourner Resultat
fin fonction
```

c. **Entrée:** $N \in \mathbb{N}$ *avec* $N > 2$ ▷ *On précise bien les valeurs acceptables en entrée*
Sortie: *Liste des N premiers termes de la Suite de Fibonacci*

```
fonction Fibonacci(N)
    Resultat ← [0,1]       ▷ On initialise avec les deux premiers termes
    pour i allant de 2 à N-1 faire       ▷ N premiers termes, donc on s'arrête à N-1
        Resultat[i] ← Resultat[i − 1] + Resultat[i − 2]
    fin pour
    retourner Resultat
fin fonction
```

d. **Entrée:** *chn, une chaine de caractères*
Sortie: *Vrai ou Faux selon si la chaine est un palindrome ou non*

```
 1: fonction EstPalindrome(chn)
 2:     Long ← longueur(chn)
 3:     si Long est pair alors
 4:         Milieu ← Long/2
 5:     sinon
 6:         Milieu ← (Long − 1)/2
 7:     fin si
 8:     pour i allant de 0 à Milieu faire
 9:         si chn[i] ≠ chn[Long − i] alors
10:             retourner Faux
11:         fin si
12:     fin pour
13:     retourner Vrai
14: fin fonction
```

Quelques remarques sur cet algorithme :

- *Lignes 8 et 9 : le but ici est de bien comprendre l'algorithme – avec cette formule on comprend bien ce qu'on fait : on part du début pour aller jusqu'au milieu. Strictement parlant les formules ne marchent pas (en Python chn[Long] donnerait une erreur) – mais on s'en fiche : la démarche est claire ici, et c'est le but.*
- *Ligne 3 : on vérifie la parité de la longueur puisque dans le cas des longueurs impaires on ignorera le caractère du milieu. Vous noterez*

qu'on dit ici "si Long est pair" sans expliquer comment on fait – la difficulté de l'algorithme n'étant pas là, c'est tout à fait acceptable (on imagine qu'il y aura un algorithme pour une fonction "EstPair" explicité ailleurs).

— *Lignes 10 et 13 : c'est une démarche qu'on a déjà utilisée ailleurs pendant notre cours, qui est très classique, et qu'il faut impérativement maîtriser : on part du principe que quelque chose est vrai — ici "chaîne est un palindrome" — et dès qu'on trouve une preuve du contraire (ici : un caractère est différent de son homologue de l'autre côté) on retourne "Faux", c'est-à-dire qu'on arrête immédiatement la fonction puisqu'on connaît son résultat. Si on arrive au bout de la boucle, c'est qu'on a pas trouvé de preuve que c'est faux – donc c'est vrai et c'est ce qu'on retourne.*

e. *La recherche dichotomique faisant l'objet d'un chapitre de ce cours, on y reviendra plus tard...*

Exercice 3: Traduction de pseudo-code en Python

a. Traduisez en une fonction Python l'algorithme de la suite de Fibonacci ;

b. Traduisez en une fonction Python l'algorithme de vérification qu'un mot est un palindrome ;

c. **Expliquez comment vous allez tester votre fonction. Comment choisissez-vous les cas que vous allez tester ?**

Pensez bien à remplir correctement la documentation (ou docstring) de votre fonction. Que renvoie la fonction `help(nom_de_votre_fonction)` ?

Pour la suite de Fibonacci, on pourra utiliser le code suivant :

```python
def Fibonacci(N):
    '''
    Fonction qui prend en entrée un entier N > 2.
    Elle renvoie les N premiers termes de la séquence de Fibonacci.
    Par ex: Fibonacci(5) renverra [0, 1, 1, 2, 3].
    '''
    Resultat = [0, 1]
    for i in range(2, N):
        Resultat.append(Resultat[i - 1] + Resultat[i - 2])
    return Resultat
```

Pour la tester, il suffira de la lancer avec différentes valeurs de N et de vérifier que le résultat est correct. Assez rapidement on constatera que si l'on passe une valeur de N qui n'est pas dans le champ des possibles (1, par exemple, ou -2) alors on obtient un résultat faux – nous reviendrons plus tard dans ce cours sur comment gérer cela.

Pour le palindrome, on pourra utiliser le code suivant :

```python
def EstPalindrome(chn):
    '''
    Fonction qui prend en entrée une chaine de caractères.
    Elle renvoie Vrai si la chaine est un palindrome, Faux sinon.
    Par ex: EstPalindrome("abba") renverra Vrai;
    EstPalindrome("abbac") renverra Faux.
    '''
    Long = len(chn)
    if Long % 2 == 0:
        Milieu = Long // 2
        # Utilisation de la division entière ("//" et non "/" pour la division simple) de manière à avoir des entiers et non des float
    else:
        Milieu = (Long - 1) // 2

    for i in range(Milieu):
        if chn[i] != chn[Long - 1 - i]: # Sans le -1 on aura une erreur
            return False

    return True
```

Pour la tester, il faudra réfléchir un peu plus à tous les cas de figures possibles :

- *Chaine de longueur paire qui est un palindrome ;*
- *Chaine de longueur paire qui n'en est pas un ;*
- *Chaine de longueur impaire qui est un palindrome ;*
- *Chaine de longueur impaire qui n'en est pas un.*

Le choix des données à tester (les "jeux de tests") fait l'objet de la section suivante de ce cours.

Vous aurez remarqué que la fonction help affiche à l'écran la docstring de la fonction que vous avez programmée – utile, vous ne pensez pas ?

2.3 Tester un algorithme

Cette étape est cruciale dans le développement d'un programme informatique car les erreurs dans les phases de rédaction de l'algorithme et de traduction en langage de programmation sont plus que fréquentes – elles sont systématiques dès lors qu'un programme atteint un certain niveau de complexité.

Un test permet de vérifier que l'algorithme fonctionne sur une donnée précise. Pour programmer efficacement il faut concevoir des *jeux de tests* permettant de vérifier que l'algorithme renvoie, dans des cas particuliers bien choisis, ce que l'on attend de lui. Il est impossible d'écrire un ensemble de tests permettant d'exclure toutes les erreurs possibles, mais on peut cependant essayer d'en construire un en respectant déjà les règles suivantes :

Règles pour tester une fonction:

Les **jeux de tests** (ensembles de données que l'on va tester) à préparer pour tester une fonction donnée doivent :

— Si la spécification de l'algorithme mentionne plusieurs cas possibles, les tester tous (ex : chaines de caractères paires et impaires pour le palindrome) ;

— Si l'algorithme doit renvoyer une valeur booléenne, construire des tests permettant d'obtenir les deux valeurs de vérité (toujours l'exemple du test de palindrome) ;

— Si l'algorithme s'applique à une liste / un tableau, effectuer un test avec un tableau vide ;

— Si l'algorithme s'applique à un nombre, effectuer des tests avec des valeurs positives, négatives, et avec zéro.

Ces règles :

— Ne sont pas exhaustives – vous devez plus les voir comme des principes à appliquer ;

— **NE SONT PAS A CONNAITRE PAR CŒUR** – les principes qui les sous-tendent, en revanche, doivent être bien compris et vous devez être capable, pour les fonctions que vous allez développer (en exercice, projet, ou contrôle), de proposer des jeux de tests qui les respectent.

3 Preuves d'algorithmes

On vient de voir comment tester un algorithme – démarche qui va nous permettre de nous rendre compte, sur des cas concrets, s'il se comporte de la manière que l'on souhaite. On ne peut en revanche jamais tester *l'intégralité* des situations possibles, et il est parfois vital d'être certain, malgré cela, que l'algorithme se termine et produit le résultat attendu à tous les coups – on peut penser par exemple à l'informatique embarquée dans le pilotage automatique de voitures ou de trains...

Pour atteindre ce but on va (comme on le ferait en mathématiques) ***démontrer*** qu'un algorithme est correct, et ce en deux étapes :

— La ***terminaison*** : on montre que l'algorithme se termine.

— La ***correction partielle*** : on montre que l'algorithme produit bien le résultat attendu.

3.1 Terminaison : variant de boucle

Dans ce cours on va limiter le champ de cette étude à la vérification que toute boucle `tant que` (ou boucle conditionnelle, ou boucle `while`) se termine bien et n'est pas infinie. En effet, dans le contexte du programme de 1$^{\text{ère}}$, c'est le seul cas où la terminaison ne serait pas garantie puisque, d'une part, les seules structures se répétant que nous envisageons sont les boucles[2], et que, d'autre part, les boucles `pour` (ou boucles itératives, ou boucles `for`) sont conçues pour se terminer au bout d'un nombre d'itérations donné, fixé, et connu à l'avance[3].

Pour prouver qu'un algorithme s'arrête, il faut donc démontrer que pour chaque boucle `tant que`, la condition d'entrée **sera invalidée dans un temps fini** – ou, dit autrement, après un nombre fini de passages dans cette boucle. La plupart du temps, cela revient à montrer qu'il existe un **variant** pour chacune de ces boucles.

Définition:

Un **variant de boucle** est une quantité entière positive qui décroît strictement à chaque itération de la boucle (une suite d'entiers naturels strictement décroissante est nécessairement finie).

Exemple, pour la boucle suivante, on peut aisément se convaincre que la quantité $5-i$ est un variant de boucle selon la définition ci-dessus : elle commence à 5 puis décroît jusqu'à atteindre 0 – on est donc bien certains que la boucle se terminera.

```
i ← 0
tant que i < 5 faire
    i ← i + 1
fin tant que
```

Attention ! Pour qu'une quantité soit un variant de boucle il faut bien qu'elle décroisse, ***mais aussi*** qu'elle soit ***toujours*** positive – en d'autres termes que la condition d'arrêt de la boucle ne soit pas étroite au point de permettre au variant de "dépasser" 0. Considérez la boucle suivante par exemple :

2. En terminale on s'intéressera à la récursivité qui induisent des répétitions potentiellement infinies sans l'utilisation de boucles.

3. même si on peut en théorie imaginer une boucle allant "de 1 à n" dans laquelle on augmente n... Mais c'est en dehors du périmètre de ce cours.

i ← 0

tant que $i \neq 5$ **faire**
    i ← i + 2
fin tant que

On voir bien que dans ce cas $5 - i$ décroît bien... mais devient assez rapidement négatif !

Exercice 4: Variant de boucle

Considérez les deux algorithmes suivants :

Afficher "entrez un entier positif"
lire Nb
i ← 0
tant que $Nb > 0$ **faire**
    $Nb \leftarrow Nb$ // 0
    i ← i + 1
fin tant que
Afficher i

Afficher "entrez un entier positif"
lire Nb
$k \leftarrow 1$
$i \leftarrow 0$
tant que $k < (Nb + 1)$ **faire**
    $k \leftarrow k \times 10$
    $i \leftarrow i + 1$
fin tant que
Afficher i

a. Que sont censés faire ces algorithmes ?
b. Utilisez la technique du variant pour justifier que le premier se termine.
c. Que se passerait-il si on remplaçait la condition "$Nb > 0$" par "$Nb \neq 0$" ?
d. Utilisez la technique du variant pour justifier que le second se termine.
e. Que se passerait-il si on remplaçait "$k < (Nb + 1)$" par "$k \neq (Nb + 1)$" ?

a. Les deux sont censés afficher le nombre de chiffres que comporte le nombre entré par l'utilisateur :
- *Le premier effectue des divisions entières successives du nombre jusqu'à atteindre 0 ;*
- *Le second procède "dans l'autre sens" en partant de 1 et en multipliant par 10 jusqu'à dépasser le nombre.*

b. Le variant ici est le nombre Nb : en effet, pour tout nombre strictement positif, sa division entière par 10 lui est strictement inférieure – $Nb//10 < Nb$ *; donc Nb est strictement décroissant.*

c. Si l'on remplace "$Nb > 0$*" par "*$Nb \neq 0$*" le variant demeure valide. En effet, à terme, les divisions entières successives par 10 atteignent exactement 0 (et ne peuvent en aucun cas devenir négatives).*

d. Le variant ici est la quantité $Nb+1-k$ *– il est facile de se convaincre qu'à chaque itération de la boucle, k étant multiplié par 10, la quantité décroît strictement.*

e. Si on remplaçait "$k < (Nb + 1)$*" par "*$k \neq (Nb + 1)$*" en revanche, l'arrêt n'aurait plus lieu que pour les valeurs de Nb égales à 9, 99, 999, etc... – pour toutes les autres, le variant passera en négatif sans "s'arrêter" à la valeur 0.*

3.2 Correction partielle : invariant de boucle

On va s'intéresser ici (dans le cadre du programme de 1$^{\text{ère}}$) à démontrer la correction des boucles dont on conçoit les algorithmes et, pour ce faire, on va se poser des questions du type :

— Les variables sont-elles bien initialisées *avant* le début de la boucle ?
— Le nombre de tours de la boucle est-il correct ?
— S'il y en a un, est-ce que l'indice est bien choisi ?
— Et, *in fine*, les valeurs obtenues en sortie de boucle sont-elles les bonnes ?

Toutes ces questions vont être abordées au moyen de la notion d'*invariant de boucle*.

> **Définition:**
> Un **invariant** est une propriété d'un algorithme qui reste vraie tout au long de son exécution.

Un invariant de boucle est une proposition toujours vraie à chaque fois que l'on entre dans la boucle. La démarche que nous allons adopter se déroule en quatre étapes :

1. Choix de l'invariant : partir "de la fin", c'est-à-dire du résultat attendu et identifier quelle quantité est "construite" au fur et à mesure des itérations de la boucle pour construire ce résultat – et ce sera votre invariant ;
2. On montre que l'invariant est vérifié avant la boucle (initialisation) ;
3. On montre que si l'invariant est vérifié *avant* un passage dans la boucle, alors il est préservé *après* le passage dans la boucle ;
4. On peut conclure sur la valeur finale à la sortie de la boucle.

Et ainsi, *par récurrence*, on démontre la correction partielle.

▷ Ca vous semble très abstrait ? Vous avez notamment l'impression que le choix de l'invariant est *très, très, très* flou ? C'est normal ! Le seul moyen d'expliquer ça clairement est de s'appuyer sur des exemples...

Considérons la fonction suivante :

Entrée: $a, b \in \mathbb{R}$ avec
Sortie: Le produit $a \times b$

```
fonction PRODUIT(a, b)
    m ← 0
    p ← 0
    tant que m < a faire
        m ← m + 1
        p ← p + b
    fin tant que
    retourner p
fin fonction
```

On commence par noter qu'on a bien un variant de boucle... Lequel ? [4] La terminaison est donc prouvée.

Etapes de la démarche :

4. $a - m$ bien sûr !

1. Choix de l'invariant : le but est de renvoyer le produit p qui, à la fin, vaudra $a \times b$; il est construit dans cette boucle par ajouts successifs de b, m fois. On peut donc avoir l'intuition que l'invariant est $p = m \times b$. Vérifions cela avec les deux étapes suivantes.
2. Avant la boucle on a p et m tous les deux à 0 – donc l'invariant est vérifié.
3. Supposons qu'au début d'une itération de la boucle l'invariant est vérifié, avec m et p ; à la fin de cette itération, les valeurs respectives de m et p seront de $m' = m + 1$ et $p' = p + b$. On aura alors :

   $$p' = p + b = m \times b + b = (m + 1) \times b = m' \times b$$

   Et donc l'invariant est bien vérifié à la fin de la boucle.
4. En fin de boucle on a $m = a$ et donc à la sortie de la boucle on a bien $p = a \times b$.

On a donc bien démontré la correction de la boucle.

Exercice 5: Calcul d'invariant de boucle

Donner un invariant de boucle pour la fonction suivante qui calcule x à la puissance n :

Entrée: $x, n \in \mathbb{N}$
Sortie: x^n

fonction PUISSANCE(x, n)
    r ← 1
    pour i allant de 0 à n - 1 **faire**
        $r \leftarrow r \times x$
    fin pour
    retourner r
fin fonction

La démarche est extrêmement proche de celle qu'on a adoptée pour le produit dans l'exemple précédent :

1. *On peut choisir comme invariant "$r = x^i$" ;*
2. *Au début de la boucle on a $r = 1$ et $i = 0$, donc l'invariant $r = x^i$ est bien vérifié, quel que soit x.*
3. *Si au début d'une itération de la boucle on a l'invariant vérifié, notons $r' = r \times x$ et $i' = i + 1$ les valeurs respectives de r et i à la fin de cette itération. On a :*

   $$r' = r \times x = x^i \times x = x^{i+1} = x^{i'}$$

4. *En fin de boucle on est passé n fois dans la boucle (de 0 à (n-1)) donc on a bien $r = x^n$.*

Exercice 6: Et un autre pour la route...

Montrer que $r = a - b \times q$ est bien un invariant de boucle de la fonction suivante, qui réalise une division euclidienne :

Entrée: $a, b \in \mathbb{N}; b \neq 0$
Sortie: q, r le quotient et le reste de la division euclidienne de a par b

```
fonction DivEuclid(a, b)
    r ← a
    q ← 0
    tant que r ≥ b faire
        r ← r − b
        q ← q + 1
    fin tant que
    retourner q, r
fin fonction
```

La démarche est extrêmement proche de celle qu'on a adoptée pour le produit dans l'exemple précédent :

1. *Le choix de la quantité identifiée comme invariant est donné par l'énoncé ;*
2. *Au début de la boucle on a $r = a$ et $q = 0$, donc l'invariant $r = a - b \times q$ est bien vérifié, quel que soit b.*
3. *Si au début d'une itération de la boucle on a l'invariant vérifié, notons $r' = r - b$ et $q' = q + 1$ les valeurs respectives de r et q à la fin de cette itération. On a :*

$$r' = r - b = a - b \times q - b = a - b \times (q + 1) = a - b \times q'$$

4. *En fin de boucle l'invariant est nécessairement encore vrai, par récurrence (le point 2 ci-dessous étant l'initialisation et le point 3 la preuve de la récurrence).*

4 Complexité d'algorithmes

Lorsque l'on commence à traiter d'importants volumes de données, que l'on commence à considérer des traitements complexes, ou tout simplement longs en nombre d'instructions exécutées se pose la question de la *performance* du traitement. On évalue cette performance suivant deux axes :

- **Performance temporelle** (que nous allons aborder ici) – il s'agit du temps d'exécution du programme.
- **Performance spatiale** (qui n'est pas au programme) – il s'agit de la mémoire nécessaire à l'exécution du programme (pour stocker notamment l'intégralité des variables qu'il manipule).

On ne s'intéresse pas ici à la durée exacte d'exécution d'un programme – celle-ci est trop dépendante de la machine sur laquelle on l'exécute, des autres traitements en cours le cas échéant, du langage de programmation... Ce à quoi on va s'intéresser c'est à la mesure du **nombre d'opérations élémentaires** que va effectuer un programme en fonction des données qu'il va recevoir en entrée. Par "opération élémentaire" on entend "étape" du programme (ligne de code, le plus souvent) dont, pour simplifier, on va considérer qu'elles prennent toutes la même temps à exécuter. Ainsi, en estimant grossièrement le nombre d'opérations élémentaires en fonction du volume de données en entrée on pourra commencer à se faire une idée de la performance d'un algorithme donné : c'est ce qu'on appelle un **calcul de complexité**.

Le but principal d'un calcul de complexité est de pouvoir comparer l'efficacité entre différents algorithmes répondant à un même problème. En d'autres termes, de répondre à la question : "*Quelle que soit la machine et le langage de programmation utilisé, l'algorithme A est-il plus performant que l'algorithme B pour de grands volumes de donénes ?*"

Définition:

La **complexité d'un algorithme** est une mesure intrinsèque à l'algorithme qui est indépendante de toute implémentation. Elle est calculée en fonction d'un *paramètre représentatif des entrées* (la taille) et à l'aide d'une *mesure élémentaire* (nombre de comparaisons, nombre d'opérations arithmétiques, etc.).

Exercice 7: Comptage d'opérations élémentaires

Pour chacun des deux algorithmes suivants, calculer le nombre d'opérations élémentaires effectué. Dépend-il des données en entrée ?

```
fonction PRODUIT1(n, b)
    p ← n × b
    retourner p
fin fonction
```

```
fonction PRODUIT2(n, b)
    m ← 0
    p ← 0
    tant que m < n faire
        m ← m + 1
        p ← p + b
    fin tant que
    retourner p
fin fonction
```

Pour Produit1, c'est vite vu : 2 opérations (le produit et le "retourner") quelles que soient les données en entrée.

Pour Produit2 c'est un peu plus compliqué :

- *On a trois opérations effectuées quoi qu'il arrive (les deux affectations avant la boucle, et le "retourner").*
- *Dans la boucle, on a deux opérations également : le calcul du nouveau m et le calcul du nouveau p.*
- *On a également une opération "cachée" dans la boucle : la comparaison "$m < n$" qui doit être effectuée à chaque itération – ce qui nous fait un total de trois opérations pour la boucle.*
- *La boucle est effectuée n fois – de $m = 0$ à $m = n - 1$.*
- *Le nombre total d'opérations élémentaires pour Produit2 est donc de :* $3 \times n + 2$.

Je vous laisse donc deviner lequel de ces deux algorithmes est le meilleur pour déterminer le produit de deux entiers n et b (imaginez le calcul de 10 milliards $\times 2$ par exemple).

Exercice 8: Comptage d'opérations élémentaires – somme des premiers entiers

Reprenons un algorithme qu'on avait écrit dans un exercice précédent et qui calculait la somme des n premiers entiers :

Entrée: $n \in \mathbb{N}$
Sortie: la somme des N premiers entiers naturels

```
fonction SommeEntiers(N)
    Resultat ← 0
    pour i allant de 1 à N faire
        Resultat ← Resultat + i
    fin pour
    retourner Resultat
fin fonction
```

Quel est son nombre d'opérations élémentaires ?

- *On a deux opérations en-dehors de la boucle (que l'on va assez rapidement apprendre à ignorer – il est évident qu'elles n'ont aucune influence sur la performance globale de l'algorithme).*
- *On a une opération évidente dans la boucle : l'ajout de l'entier en cours au résultat.*
- *On a en fait deux opérations cachées en plus : l'incrémentation de i, d'une part, et sa comparaison avec n d'autre part.*

On arrive donc à un total de $3 \times n + 2$ *opérations élémentaires.* [a]

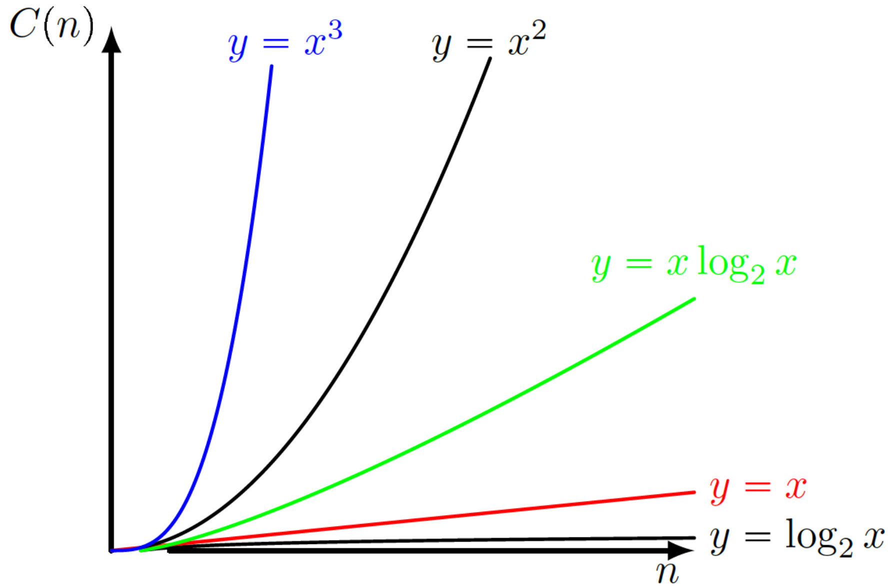
a. Pour prouver que les maths sont parfois utiles : on sait que $1 + 2 + ... + n = \frac{n(n+1)}{2}$. Il est donc possible de faire le même calcul en exactement quatre opérations contre plus de 3 millions par exemple si on utilise l'algorithme de l'énoncé sur les un million premiers entiers...

On commence à voir que les algorithmes ont souvent un lien direct entre nombre d'opérations et une quantité qu'ils reçoivent en entrée. Dans les exemples précédents on a vu deux cas :

- Aucun lien entre les deux (le calcul direct de $n \times b$ par exemple dont le nombre d'opérations ne dépend ni de n ni de b) : on parle de **complexité constante**.

— Multiple d'une quantité en entrée – l'exemple des n premiers entiers par exemple où la complexité est proche d'un multiple de n : on parle de **complexité linéaire**.

Il existe d'autres relations de ce type, nous allons en voir quelques unes dans les chapitres à venir – et leur importance est cruciale si l'on regarde le graphique ci-dessous de croissance des principales fonctions utilisées dans les calculs de complexité :



Le lien entre la complexité et les performances temporelles d'un algorithme devraient sembler évidents à la lecture de ce graphique, mais pour s'en convaincre davantage encore, il suffit de considérer les tables ci-dessous (*source : cours NSI Charles Poulmaire*). L'unité utilisée ("FLOPS") signifie "Floating-point operations per second" ou opération sur nombre flottant par seconde – c'est donc bien une unité de mesure de la performance d'un ordinateur qui se rapporte directement à la complexité qui, elle, est un mesurée comme une estimation des opérations élémentaires.

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 ps	6,64 ps	0,1 ns	0,66 ns	0,01 μs	1 μs	4×10^{10} a
	$n = 10^3$	1 ps	9,96 ps	1 ns	9,96 ns	1 μs	1 ms	∞
	$n = 10^4$	1 ps	13,28 ps	10 ns	132,8 ns	10 ms	1 s	∞
	$n = 10^5$	1 ps	16,6 ps	0,1 μs	1,6 μs	0,01 s	> 16 min	∞
	$n = 10^6$	1 ps	19,93 ps	1 μs	19,93 μs	1 s	> 11 j	∞

(a) Puissance de l'unité de calcul : 1 téraFLOPS (ordinateurs actuels)

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 fs	6,64 fs	0,1 ps	0,66 ps	0,01 ns	1 ns	4×10^{8} a
	$n = 10^3$	1 fs	9,96 fs	1 ps	9,96 ps	1 ns	1 μs	∞
	$n = 10^4$	1 fs	13,28 fs	10 ps	132,8 ps	10 μs	1 ms	∞
	$n = 10^5$	1 fs	16,6 fs	0,1 ns	1,6 ns	0,01 ms	1 s	∞
	$n = 10^6$	1 fs	19,93 fs	1 ns	19,93 ns	1 ms	> 16 min	∞

(b) Puissance de l'unité de calcul : 1 pétaFLOPS (ordinateurs les plus puissants du monde)

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 as	6,64 as	0,1 fs	0,66 fs	0,01 ps	1 ps	4×10^{5} a
	$n = 10^3$	1 as	9,96 as	1 fs	9,96 fs	1 ps	1 ns	∞
	$n = 10^4$	1 as	13,28 as	10 fs	132,8 fs	10 ns	1 μs	∞
	$n = 10^5$	1 as	16,6 as	0,1 ps	1,6 ps	0,01 μs	1 ms	∞
	$n = 10^6$	1 as	19,93 as	1 ps	19,93 ps	1 μs	1 s	∞

(c) Puissance de l'unité de calcul : 1 exaFLOPS (certains réseaux actuels ; ordinateurs attendus d'ici fin des années 2020)

Les unités de temps utilisées ici – qui permettent également de mieux se rendre compte de la vitesse à laquelle évoluent les ordinateurs – sont :

- ∞ Infini – temps que par convention on considère comme infini car irréalisable dans la réalité.
- **a** Année
- **j** Jour
- **min** Minute
- **s** Seconde
- **ms** Milliseconde – 10^{-3} secondes – un millième de seconde
- **μs** Microseconde – 10^{-6} secondes – un millième de seconde
- **ns** Nanoseconde – 10^{-9} secondes – un milliardième de seconde
- **ps** Picoseconde – 10^{-12} secondes – un mille milliardième de seconde
- **fs** Femtoseconde – 10^{-15} secondes – un million de milliardième de seconde
- **as** Attoseconde – 10^{-18} secondes – un milliard de milliardième de seconde

5 Algorithmes de tri