Note de cours sur la complexité

https://www.youtube.com/watch?v=P66h8D5Lkwk (début → 3'30)

De nos jours, on peut associer :
Un ruban → Disque Dur
Table d'action → Un programme
Machine de Turing → un ordinateur
Enfin pas tout à fait, la machine de Turing telle que manipulé est spécifique à un programme donné.
Turing a conceptualisé une Machine de Turing Universelle, qui serait valable quel que soit le programme.
C'est un peu plus complexe que prévu par le programme de 1ère, je vous laisse poursuivre la lecture de la vidéo chez vous, si vous le souhaitez.

Pourquoi présenté la machine de Turing dans le programme de NSI,
D'abord parce que Turing est un personnage important de l'histoire de l'informatique et il a, avec Church (thèse de Church-Turing) conceptualisé la notion de calculabilité.

On ne retiendra ici, que le modèle de la machine de Turing, sert d'étalon pour mesurer la complexité d'un algorithme : C'est l'ordre de grandeur du nombre d'actions élémentaires (lire, déplacer la tête de lecture) qu'effectuerait une machine de Turing pour effectuer l'algorithme.

C'est quoi la complexité ?

En informatique, la question de la performance des programmes est centrale. Si un problème est traité en un temps raisonnable, l'utilisateur est satisfait. De manière générale, le traitement d'un certain volume de données requiert un temps d'exécution lié à ce volume de données.
On se contente en général d'une estimation de ce temps.

Si en doublant un volume de n données, un programme renvoie une réponse au même problème avec un temps multiplié par 2 et un autre programme renvoie une réponse au même problème avec un temps multiplié par 4, on peut raisonnablement penser que le premier programme est plus efficace !
On dit que sa complexité temporelle est meilleure.

Ajoutons qu'il existe également la notion de complexité spatiale, qui prend en compte l'espace mémoire occupé au cours de l'exécution du programme.

<u>Revenons à la complexité temporelle :</u>
Exemple :

```
N = 100
For i in range(n)
        Print (i)
```

Le nombre d'opérations est de 1 + n * 1. Soit une complexité de l'ordre de n : O(n)

Que pensez-vous des programmes suivants :

```python
def fonction1(n):
    a=0
    for _ in range(n):
        a=a+1

def fonction2(n):
    a=0
    for _ in range(n):
        a=a+1
    for _ in range(n):
        a=a+1

def fonction3(n):
    a=0
    for _ in range(n):
        for _ in range(n):
            a=a+1
```

La fonction magique %timeit, qui mesure le temps moyen d'un code sur un très grand nombre d'exécutions.

Ou sinon

```
from time import *
debut = perf_counter()
#code ici
fin = perf_counter()
print("temps passé : ".fin-debut) :
```

Classification :

Ces outils de comparaison permettent de classer les algorithmes selon leur complexité :

- Complexité **constante** en O(1)
- Complexité **logarithmique** en $O(\log_2 n)$
- Complexité **linéaire** en O(n)
- Complexité **quasi-linéaire** en $O(n\log_2 n)$
- Complexité **polynomiale** en $O(n^k)$
- Complexité **exponentielle** en $O(2^n)$

Exemple :

```python
def moyenne(uneListe):
    """
    Calcul de la moyenne d'une liste de nombres passée en argument
    """
    #Calcul de la somme des éléments de la liste
    somme =0                        #initialisation
    for elt in uneListe:            #Boucle sur les éléments de la liste
        somme = somme + elt         # ajout de l'élément courant
    #Division de la somme par le nombre de termes
    return somme/len(uneListe)
```

Une affectation somme = 0
Une affectation + une addition pour chaque itération : soit 2n opérations
Une division pour le calcul final soit une opération

Le cout total est de f(n)=2n+n
On dit que la complexité est de l'ordre de O(n)

Preuve d'un algorithme

Il est important de pouvoir montrer qu'un programme termine, afin de savoir si son exécution se fera sans problème.

→ Parmi les instructions ci-dessous, quelles sont celles qui ne peuvent pas engendrer d'exécution infinie d'un algorithme ? Quelle est/sont la/les seule(s) instruction(s) problématique(s) ?

- Affectation
- Instruction conditionnelle (if … Then …else …)
- Boucle Pour (for)
- Boucle Tant que (While)

On suppose maintenant que l'on est capable d'écrire un algorithme, appelé Terminator, dont la fonction est de répondre vrai si un programme termine et faux si un programme ne termine pas. Ainsi, l'instruction `Terminator(P)` renvoie vrai si le programme P termine toujours et faux si P est capable de boucler.

Algorithme 4 : SarahConnor

```
tant que Terminator(SarahConnor) faire
    rien
fin
```

À quelle condition le programme `SarahConnor` termine-t-il ?

On comprend donc que le problème de la terminaison des algorithmes n'est pas si simple. On peut énoncer le théorème suivant :

Théorème 1 Problème de l'arrêt

Il n'existe pas de programme permettant de dire si un algorithme termine toujours ou non.

On dit en théorie de l'informatique que le problème de l'arrêt est indécidable. En fait, le théorème de Rice dit même que toute propriété non triviale sur les programmes est indécidable. Bien que ce soit vrai dans le cas général, on peut cependant s'intéresser à des cas particuliers et utiliser des propriétés mathématiques pour démontrer que des algorithmes simples terminent.

1.1 Définition

Définition :

On appelle preuve d'un algorithme, la propriété qui assure à ce dernier :

- *de se terminer. On appelle cela la* **terminaison** *de l'algorithme*
- *de réaliser ce qu'on attend de lui. On appelle cela la* **correction** *de l'algorithme.*

1.2 Comment ≪prouver≫ ?

a. Terminaison

Il est fréquent dans l'établissement d'un algorithme, qu'un programmeur ait recours a une structure de boucle. Lorsque cette dernière est conditionnelle (while), et que l'algorithme exécute une première fois les instructions contenues dans la boucle, il est important de s'assurer que l'algorithme sortira de la boucle et se terminera. Cette propriété de l'algorithme s'appelle la terminaison

Ainsi :

Le groupe d'instructions de la boucle doit permettre une modification de la condition de boucle.

Définition :

*On appelle **convergent (ou variant de boucle)** une quantité qui prend ses valeurs dans un ensemble bien fondé et qui diminue strictement à chaque passage dans une boucle.*

Remarque : Un ensemble bien-fondé est un ensemble totalement ordonné dans lequel il n'existe pas de suite infinie strictement décroissante.

Propriété :

L'existence d'un convergent pour une boucle garantit que l'algorithme finit par en sortir.

Propriété :

*On assure la **terminaison** d'un algorithme lorsque toutes les structures de boucles conditionnelles de celui-ci ≪terminent≫*

Pour la boucle For en Python, on utilise la fonction range qui est strictement décroissante et positive. La boucle For se termine donc toujours.

En effet, on peut toujours construire un variant simple.

Si la boucle est donnée par la structure : Pour i allant de a à b, un variant simple est b – i

b. Correction

En outre, la seconde préoccupation du programmeur sera d'assurer que son algorithme réalise bien le travail demande. Cette propriété de l'algorithme s'appelle la correction. Elle est généralement plus délicate à établir.

Propriété :

*On assure la **correction** d'un algorithme avec boucle en dégageant une propriété vérifiée avant l'entrée dans la boucle et qui le restera durant chaque itération i de boucle ; soit $\mathcal{P}_i$ cette propriété au rang i. Cette propriété doit permettre de renvoyer le résultat attendu au dernier rang de boucle. On l'appelle **l'invariant de boucle**.*

Définition :

***L'invariant de boucle** est une formule logique qui :*

— est vérifiée à l'initialisation de la boucle

— reste vraie à chaque itération de la boucle

1.3 Quelques exemples

a. Factorielle

On note n! le nombre entier défini par n! = n x (n - 1) x (n - 2) x ... x 3 x 2 x 1. (on dira « n factorielle »)

On considère le script python suivant :

```
n=input("Veuillez saisir un entier : " )
if type(n)==int and n>=0 :
    k=1
    f=1
    while k<=n :
        f=f*k
        k=k+1
    print(f)
else :
    print (" Impossible ")
```

A faire : Calculer factorielle(4) et 4!, puis factorielle(7) et 7!

• Terminaison:

- Si *n* entre au clavier est négatif, le programme termine sur un message ("impossible").
- Si *n* est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
- Si $n > 0$ k étant initialement a 1, la boucle est exécutée. A chaque itération, *k* est incrémenté de 1 et finit par être supérieur a *n* donc pour k=n+1, on sort de la boucle, le programme renvoie *f*, et termine.

CONCLUSION : la terminaison est assurée.

• Correction:

Un invariant de boucle P_i est par exemple :

≪après la i_{ieme} itération k contient i + 1 et f contient i!≫

Cette propriété est vraie au rang 0. Supposons la vraie au rang *i*, et montrons qu''elle est héréditaire :

- Au rang $i + 1$, on a : *k* qui contient $i + 1$ en début d'itération et $f = i! \times (i + 1) = (i + 1)!$
- En fin d'itération *k* contient $i + 2$

Ceci est bien la propriété au rang $i + 1$

CONCLUSION : la correction est assurée.

A retenir :

Voici une méthode pour prouver la terminaison et la correction d'un algorithme ayant une boucle while :

- Définir clairement les **préconditions** – état des variables initial (avant la boucle)
- Déterminer la variant de boucle et prouver la terminaison de la boucle
- Définir un invariant de boucle
- Prouver que l'invariant est vrai au début de la boucle et à chaque itération
- Montrer qu'en sortie de boucle l'invariant est vrai et que combiné à la condition de sortie de boucle il permet de prouver que l'algorithme est correct

Exercice : Puissance de 2

On considère le code python calculant la puissance n_{ieme} de 2 :

```
n=input ("Veuillez saisir un entier : " )
if type (n)==int and n>=0 :
    p=1
    while n>0 :
        p=2*p
        n=n□1
    print (p)
else :
    print (" Impossible ")
```

A Faire : Donner la preuve de cet algorithme

• Terminaison:

- Si n entre au clavier n'est pas un entier positif ou nul le programme termine sur un message ("impossible").
- Si n est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
- Si $n > 0$, la boucle est exécutée. A chaque itération, n est décrémente de 1 et finit par être nul, on sort de la boucle, le programme renvoie p, et termine.

Conclusion : la terminaison est assurée.

• Correction:

Un invariant de boucle est par exemple :

$\ll$ *apres la* i^{eme} *iteration* p *contient* $2^{n0-(n0-i)} = 2^i$ *et* n *contient* $n_i = n_0 - i \gg$

Les conditions initiales assurent qu''au rang 0 la propriété est vraie. Supposons la vraie au rang i, et montrons Qu'elle est héréditaire :

- Au rang $i + 1$, on a : p qui contient $2 \times 2^{n_0-(n_0-(i+1))} = 2^{i+1}$
- En fin d'itération n contient $n_{i+1} = n_0 - (i + 1)$

Ceci est bien la propriété au rang $i + 1$

Conclusion : la correction est assurée.