1 Pour bien démarrer

Le principe général de la programmation impérative est celui de la « recette de cuisine ». Il s'agit de donner une suite d'instructions élémentaires qui après exécution aveugle par le processeur produiront le résultat attendu. Le contenu d'une fonction impérative est donc une suite d'instructions qui effectue des calculs, des comparaisons, utilisent les résultats intérmédiaires pour des modifications de la mémoire (créations/modifications de variables) et terminent par le renvoi d'une valeur (de type arbitraire).

1.1 Variables et listes

Dans n'importe quel langage de programmation, la notion de variable tient une place essentielle. En quelques mots, une variable est une « boite » désignée par un nom et et dans laquelle on peut stocker une information (de nature très large, et donc pas seulement un entier ou un réel). Dans le langage python, la création d'une variable se fait sous la syntaxe générale :

```
ma_variable = contenu
```

et donc par exemple

```
a = 1; b = False; c = 8.21
```

Si l'on veut stocker plusieurs informations sous un même nom, on peut utiliser la notion de tuple. Il suffit alors de séparer les différents éléments par des virgules, et entre parenthèses. On accède alors au composant du tuple par des crochets :

```
a = (1,False,8.2); a[0];
```

La commande `len` permet d'obtenir le nombre de valeurs contenues dans le tuple. Les différents composants du tuple ne sont pas modifiables. Il faut donc redéfinir toute la variable si on souhaite en changer ne serait-ce qu'une valeur.
Pour finir, si l'on souhaite un stockage dynamique d'information, c'est-à-dire la possibilité de modifier et/ou de rajouter/-supprimer des éléments contenu dans la variable, on dispose de l'outil fourni par les listes. Elles se définissent de la même manière que les tuples, en remplaçant cette fois les parenthèses par des crochets.

```
a = [1,False,8.2]
```

On peut à nouveau accéder au nombre d'éléments par la commande `len`, et à la valeur de la i-ième donnée (on parle aussi de la i-ième case de la liste) par la commande `a[i]`. On dispose également des commandes suivantes :

- `append` pour ajouter un élément. D'une manière générale, l'exécution `s.append(x)` ajoute l'élement `x` à la liste `s`. L'insertion se fait systématiquement en fin de liste. Si on veut une insertion à l'interieur de la liste, il faut utiliser la fonction `insert` qui prend un argument supplémentaire (la position d'insertion). On écrira donc par exemple `s.insert(2,9)` pour insérer la valeur 9 en position 2 (ce qui décale les éléments suivants vers la droite).
- `remove` pour supprimer un élément : l'exécution `s.remove(i)` supprime la i-ième valeur et redécale donc tout le reste de la liste pour combler le trou. La fonction `pop` permet de supprimer le dernier élément de la liste et de le renvoyer comme résultat de la commande. Elle s'utilise sur une liste `s` par l'instruction `s.pop()`.
- Signalons pour finir `reverse` qu iinverse l'ordre de la liste, et `sort` qui la trie par ordre croissant dès lors que tous les éléments sont d'un même type.

Remarque 1 : Choix des noms de variable

Le nom d'une variable doit commencer par une lettre (majuscule ou minuscule) et ne pas comporter d'espace. En dehors de cela, il n' y pas de contrainte, juste des conventions pratiques :

- Il doit être court de préférence, puisqu'il est susceptible d'être réécrit plusieurs fois (mais aussi pour ne pas alourdir la lecture du code).
- Il doit être suffisamment explicite pour qu'on devine le rôle de la variable à son nom (lors d'une relecture ultérieure).

Pour ne pas surcharger le code, il faut si possible éviter d'introduire des variables inutiles. Par exemple, si `L` est une liste, ça ne sert strictement à rien de poser `a = L[0]`.

1.2 Structures de contrôle

En programmation impérative, un structure de contrôle est une commande qui contrôle l'ordre dans lequel les instructions d'un algorithme vont être exécutées. Dans les langages courants, il apparaît essentiellement deux grandes familles :

- les alternatives : on exéctue un bloc d'instructions si une certaine condition est réunie ;
- les boucles : on exécute un bloc d'instructions à plusieurs reprises.

Dans d'anciens langages de programmation, on trouvait également des commandes dites de sauts. Puisque les lignes d'une programme sont systématiquement numérotées, ces intructions permettaient d'avancer directement ou de revenir à une ligne particulière de la fonction. Toutefois, ces techniques ont globalement disparu de la plupart des langages modernes compte tenu des nombreux problèmes que ce type d'instruction soulevait (ne serait que si on rajoute une ligne au milieu d'un code, ce qui décale tout le reste !).

Branchements

En informatique, on appelle branchement toute structure de programmation effectuant un test logique sur une condition et permettant un choix entre divers blocs d'instructions suivant le résultat de ce test. En ce qui concerne la programmation en python, la syntaxe générale est la suivante :

```
if test1:
        instruction1
elif test2:
        instruction2
...
else:
        instruction_k
```

Remarque 2

- Les tests sont effectués dans l'ordre d'écriture. Si un test réussit, les instructions executées sont uniquement celles qui suivent le : et sont indentées à partir de ce dernier.
- Attention à ne pas oublier les deux points (:) pour ne pas risquer l'erreur de syntaxe. Tous les éditeurs interactifs de python prevoient automatiquement l'indentation après ces deux points, ce qui facilite l'écriture et permet d'éviter de traîner des erreurs.
- Le dernier `else` est facultatif si l'on a rien à faire lorsqu'aucun des tests ne réussit.
- Dans la mesure du possible, éviter d'utiliser un branchement pour renvoyer un booléen. L'instruction
  ```
  if test1:
      return True
  else:
      return False
  ```
  est complètement équivalente à `return test1` qui est beaucoup plus court.

A titre d'exemple, le programme suivant détermine si un entier donné en argument est pair :

```
def pair(n):
    if (n%2)==1:
        return False
    else:
        return True
```

ou tout simplement

```
def pair(n):
    return ((n%2)==0)
```

Boucles

La boucle est d'une manière générale une structure permettant de répéter les mêmes instructions plusieurs fois de suite. On distingue principalement deux types de boucles :

- les boucles `for`, qui s'exéctuent un nombre de fois déterminé à l'avance ;
- les boucles `while`, qui s'exécutent tant qu'une condition est remplie.

A noter pour ces dernières la possibilité de **boucles infinies** si l'on s'y prend mal, lorsque l'exécution de l'algorithme ne tombe jamais sur la condition de sortie de boucle. A ce moment, l'ordinateur ne permet plus d'interagir à moins d'interrompre le calcul en cours.

La syntaxe de la boucle `while` est simplement la suivante en python :

```
while condition:
        instruction1
        instruction2
        ...
```

A nouveau, il ne faut surtout pas oublier les (:), puisque leur absence entraînera une erreur de syntaxe. Seules les instructions indentées à partir des deux points sera exéctuée après chaque vérification de la condition.
A titre d'exemple, le programme suivant déterminer, pour un entier n donné la plus grande puissance de 2 inférieure ou égale à n.

```
def power_2_inf(n):
    p=1
    while p<=n:
        p=2*p
    return(p//2)
```

En ce qui concerne la boucle `for`, l'utilisation la plus classique consiste à appliquer la même suite d'instruction pour toutes les valeurs d'un compteur `i` appartenant à un intervalle d'entiers. C'est pourquoi la plupart du temps, on écrira pour répéter n fois un bloc avec $n \in \mathbb{N}$,

```
for i in range(n):
        instruction1
        instruction2
        ...
```

Cette syntaxe raccourcie masque cependant une utilisation plus souple de la boucle `for`, puisque cette dernière permet plus généralement d'exécuter les instructions du bloc pour toute valeur de `i` appartenant à la liste fournie après le `in` et ce, quel que soit la nature des éléments de l'ensemble (et donc pas nécessairement entiers, par nécessairement par ordre croissant, etc ...). L'instruction `range(n)` ne fait en pratique que créer une liste contenant les entiers de 0 à $n-1$. A titre d'exemple, on peut donc tester l'instruction

```
L=[1,2,True,3.8,"toto"]
for x in L:
    print(x)
```

Remarque 3 : Listes en compréhension

L'instruction `for` est également fort utile pour créer de nouvelles à l'aide d'une syntaxe typique à Python qui porte le nom de liste en compréhension. De la même manière qu'en mathématiques, on écrit

$$f(A) = \{f(a), a \in A\}$$

pour décrire littéralement « l'ensemble des $f(a)$ pour a appartenant à A », on peut écrire en Python

```
L'=[f(a) for a in L]
```

pour créer la liste `L'` des images des éléments d'une liste `L` par la fonction f. En particulier, pour copier une liste `L`, on écrit tout simplement

```
L'=[a for a in L]
```

et surtout pas `L' = L` qui ne crée pas une copie : toute modification de `L'` entraînera la même modification sur `L`.

En guise d'application, signalons pour finir le programme suivant qui permet, étant donné une suite définie par récurrence par

$$u_0 \in \mathbb{R} \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1} = f(u_n)$$

de renvoyer la valeur de u_n à partir des arguments f, n et u_0 :

```
def suite(f,n,u0):
    u=u0
    for i in range(n):
        u=f(u)
    return u
```

Si l'on veut récupérer toute la liste des valeurs $u_0, \dots, u_n$, on peut écrire

```
def liste_suite(f,n,u0):
    L=[u0]
    for i in range(n):
        L.append(f(L[len(L)-1]))
    return L
```

2 Eléments d'analyse des algorithmes

2.1 Notions de complexité

Principe général

La complexité temporelle est un outil théorique en algorithmique qui permet de mesurer l'efficacité d'un programme. Le principe général est de donner un ordre de grandeur du nombre d'opérations que devra effectuer le programme pour terminer son travail. Il se traduit par une relation entre le temps d'exécution et la taille des données en entrées. Les objectifs sont alors de pouvoir déterminer quel est le meilleur algorithme pour résoudre une problématique, d'identifier pour un algorithme les cas favorables, moyens et défavorables d'exécution.
On peut également mentionner la notion de complexité spatiale, qui mesure de son coté l'occupation mémoire nécessaire à l'exécution de l'algorithme, mais son étude n'est pas au programme des classes préparatoires.
Un point essentiel en analyse de complexité tient au fait que le travail doit être indépendant de tout contexte : le langage de programmation ne doit pas être pris en compte, pas plus que la nature de la machine qui exécute le travail. Pour ces raisons, l'unité de mesure utilisée ne saurait être exprimée en unité de temps standard (seconde/minute, etc ...).
Commençons par quelques définitions essentielles :

Opérations élémentaires

- On considère qu'une opération est élémentaire en informatique si son temps d'exécution est minimal parmi toutes les instructions effectuées par le programme. En pratique, il est quasiment impossible de donner une définition rigoureuse

de cette notion, on se contente de conventions d'usages. Une définition trop précise serait non seulement inutile, mais en plus nuisible en compliquant les calculs.

- En pratique, on considère donc que les opérations suivantes sont élémentaires :
  - addition, soustraction, multiplication, division, modulo, aussi bien sur les les entiers que sur les flottants.
  - comparaison, opérations booléennes (et, ou, non).
  - création, lecture, modifications simples de variables.

Notions de taille et de coût

Notons δ l'argument d'un programme (de nature quelconque : entier, liste, voire couple ou p-uplets). Le nombre d'opérations élémentaires effectuées par un algorithme lors de son exécution est généralement noté $T(\delta)$. L'objectif de la complexité n'est absolument pas d'obtenir l'expression de $T(\delta)$ pour toute entrée δ, ce qui n'aurait aucun intérêt. Pour simplifier, on associe à chaque entrée δ une notion de **taille**, la plupart du temps donné sous la forme d'un entier (le plus fréquent), ou de plusieurs entiers (rarement plus de deux). A titre d'exemple :

- Si δ est un entier, sa taille est généralement égale à lui-même ;
- Si δ est une liste ou un tableau, sa taille est égale à sa longueur. Si elle contient des entiers, on peut également dans certains cas de figure faire intervenir le maximum des valeurs.
- Si δ est un couple d'entiers, on prend suivant les cas pour la taille le maximum du couple, la somme des deux éléments, ou le couple lui-même !

Etant donné un entier n fixé, on définit alors Δ_n comme l'ensemble des données de taille n, ce qui permet ensuite de distinguer trois notions de complexité :

- Complexité dans le pire cas : $C_{\max}(n) = \max_{\delta\in\Delta_n} T(\delta)$
- Complexité dans le meilleur cas : $C_{\min}(n) = \min_{\delta\in\Delta_n} T(\delta)$
- Complexité moyenne : $C_{\text{moy}}(n) = \dfrac{1}{\text{Card}(\Delta_n)} \sum_{\delta\in\Delta_n} T(\delta)$

Toutes les analyses de complexité ont alors pour objectif de donner un **ordre de grandeur** de la quantité $C(n)$ sous la forme $C(n) = O(f(n))$ avec une fonction $f : \mathbb{R} \longmapsto \mathbb{R}$ la plus petite possible. Dans le cadre du programme des classes préparatoires, on s'intéresse essentiellement à la complexité dans le pire cas, les autres analyses de complexité pouvant l'object de présentations ponctuelles sur des exemples.

Classes de complexités

Définition 1

Soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ des suites de réels positifs. On note $u_n = \Theta(v_n)$ lorsque l'on a simultanément

$$u_n = O(v_n) \qquad \text{et} \qquad v_n = O(u_n)$$

On distingue plusieurs classes de complexité classiques :

- logarithmique $\Theta(\ln n)$ (très efficace)
  recherche dans un tableau trié, exponentiation rapide
- linéaire $\Theta(n)$ (efficace)
  recherche exhaustive dans un tableau non trié, calcul d'une suite récurrente
- quasi linéaire $\Theta(n \ln n)$ (assez efficace)
  tri fusion ou par tas, calcul d'une enveloppe convexe

- polynomiale $\Theta(n^k),\ k \in \mathbb{N}$ (moyennement efficace si $k = 2$, inefficace sinon)

  tri insertion/bulle/selection $O(n^2)$, multiplication matricielle, pivot de Gauss $O(n^3)$
- exponentielle $\Theta(a^n),\ a > 1$ (inutilisable, sauf pour n petit)

  satisfaisabilité d'une formule booléenne
- factorielle $\Theta(n!)$ (inutilisable, sauf pour n petit)

  calcul du déterminant par développement selon une colonne, voyageur de commerce naïf

Pour un programme quasi-linéaire, le terme $\log n$ ne dépasse en pratique pas quelques dizaines et peut donc être considéré quasiment constant, du moins quand on manipule des données à l'échelle humaine ($n \leq 10^{20}$). Les tableaux ci-dessous permettent d'apprécier l'importance de la complexité de l'algorithme

- le premier donne le temps de calcul nécessaire à un ordinateur effectuant 10^{12} opérations par secondes pour traiter une donnée en fonction de la complexité de l'algorithme. Un symbole ∞ désigne un temps supérieur à 10^{100} années.

	$\ln n$	n	$n \ln n$	n^2	n^3	2^n
10^2	$4,6 \cdot 10^{-6} \mu s$	$10^{-4}\,\mu s$	$4,6 \cdot 10^{-4}\,\mu s$	$0.01\,\mu s$	$1\,\mu s$	$4.02 \cdot 10^{22}\,ans$
10^3	$6,9 \cdot 10^{-6} \mu s$	$10^{-3}\,\mu s$	$6,9 \cdot 10^{-3}\,\mu s$	$1\,ms$	$1\,ms$	∞
10^6	$1.38 \cdot 10^{-5} \mu s$	$1\,\mu s$	$13,8\,\mu s$	$1 \cdot s$	$11,57\,jours$	∞
10^9	$2,07 \cdot 10^{-5} \mu s$	$1\,ms$	$20,7\,ms$	$11,57\,jours$	$31,68 \cdot 10^6\,ans$	∞

- le second donne la taille maximale d'une donnée que l'on peut traiter suivant le temps dont on dispose. Un symbole ∞ signifie une quantité supérieure à des quantités « humaines » ($> 10^{20}$).

	$\ln n$	n	$n \ln n$	n^2	n^3	2^n
$1\,s$	∞	10^{12}	$4,09 \cdot 10^{10}$	10^6	10^4	39
$1\,min$	∞	$6 \cdot 10^{13}$	$2,11 \cdot 10^{12}$	$7,74 \cdot 10^6$	39 148	45
$1\,heure$	∞	$3,6 \cdot 10^{15}$	$1,11 \cdot 10^{14}$	$6 \cdot 10^7$	153 261	51
$1\,jour$	∞	$8,6 \cdot 10^{16}$	$2,43 \cdot 10^{15}$	$2,93 \cdot 10^8$	442 083	56

Les informations essentielles à retenir de ces deux tableaux sont les suivantes :

- A l'échelle humaine, le gain de temps entre un algorithme de complexité quadratique et un algorithme quasi-linéaire est considérable.
- Tout algorithme de complexité supérieure à une complexité quadratique, et notamment de complexité exponentielle, est en pratique très vite inutilisable sauf pour de petites valeurs de n, et ce indépendamment du temps dont on dispose pour laisser l'ordinateur faire ses calculs.

Remarque : Le site *www.top500.org* recense les statistiques des ordinateurs les plus performants du monde, ainsi qu'un classement des machines suivant ces performances. En avril 2014, la machine la plus puissante s'appellait *Tianhe-2 (Milky-Way-2)*. Elle est située en Chine dans la ville de Guanzhou, est capable d'effectuer $61 \cdot 10^{15}$ multiplications en flottants par seconde, et consomme une énergie équivalente à celle d'une petite ville de 20 000 habitants. En Novembre 2022, cette machine n'est plus que la 7-ième du classement, la première étant *Supercomputer Fugaku*, située au Japon, qui émarge à $442 \cdot 10^{15}$ multiplications en flottants par seconde.

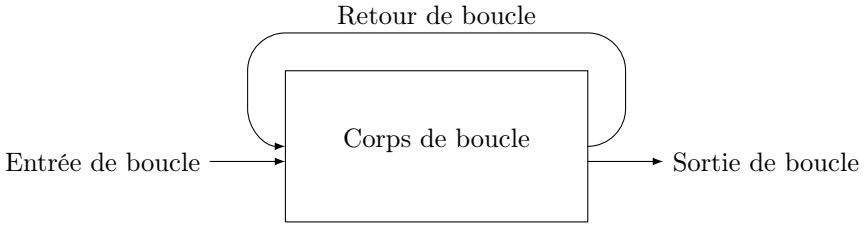
2.2 Preuves de programmes impératifs

Invariants de boucles

La preuve d'un programme a pour objectif de justifier rigoureusement qu'un programme fonctionne et fait bien ce que l'on attend de lui. Précisément, il faut

- prouver que le programme termine, c'est-à-dire que son exécution n'effectue qu'un nombre fini de calculs.
- que le résultat du programme (ou son action pour une fonction dont le type de retour est `unit`) est bien celui est attendu.

Ce travail en style impératif passe par la notion d'invariant de boucle. Qu'il s'agisse d'une boucle `for` ou d'une boucle `while`, le schéma général d'une boucle est le suivant :



Définition 2

Un invariant de boucle est un prédicat en i où i est le nombre de retour de boucles effectuées depuis le début de l'exécution de l'algorithme. Il décrit l'état de la mémoire juste après l'exécution de la i-ème boucle (pour $i = 0$, il décrit les conditions initiales).

La preuve d'un programme s'effectue alors de la manière suivante :

1. On définit un invariant de boucle décrivant l'état de la mémoire et on justifie qu'il est vrai à l'instant 1 (c'est-à-dire avant l'exécution de la première boucle), en se basant sur les initialisations des variables.
2. On prouve par récurrence que l'invariant de boucle reste vrai pour tout i (du moins jusqu'à la fin de l'exécution de la fonction).
3. La boucle s'effectue soit un nombre fini de fois (boucle `for`), soit lorsqu'une condition est vérifiée (auquel cas il faut justifier que ce cas fini par se produire). On se sert alors de l'invariant et s'il y a lieu de la condition de sortie de boucle pour prouver que le résultat est correct.

Précisions également que l'invariant de boucle peut non seulement servir à la preuve de la terminaison d'une boucle `while`, mais il permet également de majorer le nombre d'exécutions de cette boucle, ce qui permet notamment d'évaluer sa complexité.

2.3 Quelques exemples

Tri bulle

Le tri bulle est une méthode de tri naïve particulièrement simple à implémenter. On parcourt le tableau de gauche à droite en comparant les éléments consécutifs. S'ils ne sont pas dans le bon ordre, on les permute. On répète l'opération n fois où n est la longueur du tableau mais à chaque nouveau parcours, on fait une comparaison de moins.

```
1  def echange(L,i,j):
2      a=L[i]
3      L[i]=L[j]
4      L[j]=a
5
6  def tri_bulle(L):
7      n=len(L)
8      for i in range(n):
9          for j in range(n-i-1):
10             if L[j]>L[j+1]:
11                 echange(L,j,j+1)
```

Le nom du tri bulle vient (probablement ?) du fait que l'algorithme fait remontrer successivement le plus grand élément du tableau vers la droite du tableau (comme les bulles remontent le long du verre). En effet, dès lors que l'indice `j` atteint le maximum du tableau, le test de la ligne 10 est toujours vrai et l'élement en position `j` est systématiquement échangé avec son voisin de droite. Pour une analyse rigoureuse, on peut rédiger de la manière suivante :

- **Preuve du programme :** On utilise l'invariant de boucle suivant :

  $$\mathcal{P}(i) : \quad \text{Après } i \text{ tours de boucles, } \quad \{t.(0), t.(1), \ldots, t.(n-i-1)\} \leq t.(n-i) \leq \cdots \leq t.(n-1)$$

  Autrement dit, les i derniers éléments du tableau sont rangés par ordre croissant et supérieurs aux $n-i$ éléments en tête du tableau. La preuve se fait par récurrence (laissée au lecteur).

- **Complexité :** La boucle sur `j` s'exécute $n-i+1$ fois et n'effectue qu'un nombre borné d'opérations élémentaires (quel que soit ce que l'on décide de compter, entre les accès au tableau, les comparaisons, et les modifications du tableau). Par conséquent, son coût est un $O(n-i-1)$ et la complexité totale est

$$\sum_{i=0}^{n-2} O(\,n-i-1) = O\left(\sum_{i=0}^{n-2}(n-i-1)\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

  c'est-à-dire quadratique. On notera qu'on aboutit au même ordre de grandeur si on majore grossièrement le coût de la boucle sur `j` par un $O(n)$. De plus, cet algorithme effectue le même ordre de grandeur d'opérations élémentaire quel que soit le tableau. Autrement dit, les complexités dans le pire cas, le meilleur cas, et en moyenne sont identiques.

Recherche dichotomique dans un tableau trié

La recherche dichotomique utilise l'hypothèse selon laquelle un tableau `t` est trié pour couper en deux à chaque étape la zone de recherche d'un élément `x`. En effet, si la valeur `x` est supérieure à une valeur `t.(i)`, elle se situe nécessairement dans les cases d'indices supérieures à `i` ou nulle part dans le tableau. On en déduit le code suivant.

```
def recherche_dico(L,x):
    a=0; b=len(L)-1
    while (b>=a):
        m=(a+b)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            a=m+1
        else:
            b=m-1
    return False
```

Justifions proprement la correction et donnons une estimation de la complexité de cette fonction.

- **Terminaison et preuve du programme :** La présence d'une boucle `while` impose en plus de la preuve de correction d'inclure une preuve de terminaison de l'algorithme. Une rédaction rigoureuse peut se faire par l'invariant de boucle suivant :

  $\mathcal{P}(i)$: Après i tours de boucles, l'élément `x` ne peut se trouver qu'entre les indices `a` et `b` du tableau (inclus). De plus, $|b-a| \leq n/2^i$ où n est la longueur du tableau.

  Là encore, la preuve de cet invariant par récurrence est laissée au lecteur. Dès lors, pour $i \geq \lfloor \log_2(n) \rfloor + 1$, on a nécessairement $b \leq a$ et l'exécution de l'algorithme s'achève au plus tard à la fin de cette i-ième boucle. En effet, soit `x` est dans la case d'indice `a` auquel cas `r` passe à `true`, soit `b < a` à la fin de cette boucle.
- **Complexité :** On se sert à nouveau de l'invariant précédent. Celui garantit que la boucle `while` s'exécute au plus $O(\log_2 n)$ fois. Or, il n'y a qu'un nombre borné d'opérations élémentaires au sein de celle-ci. Par conséquent, la complexité globale est logarithmique.

Crible d'Eratosthène

Le crible d'Eratosthène est une méthode pour déterminer la liste des nombres premiers inférieurs ou égaux à un entier n donné. L'algorithme procède par élimination successive des entiers composés en éliminant d'un tableau les indices multiples de 2, puis de 3 et ainsi de suite. L'implémentation ci-dessous utilise un tableau de booléens de $n+1$ cases. La fonction renvoie un tableau dont la case d'indice `i` vaut `true` si et seulement si `i` est un entier premier.

```
def crible(n):
    L=[True for i in range(n+1)]
    L[0]=False; L[1]=False
    for i in range(2,n+1):
        if L[i]:
            k=2
            while (k*i <= n):
                L[k*i]=False
```

```
9                       k+=1
10      return(L)
```

Remarque : Un entier k non premier admet nécessairement un diviseur plus petit que $\sqrt{k}$. On pourrait donc remplacer la boucle `for` par une boucle `while` qui s'arrête lorsque `i*i` dépasse `n`. Mais cela n'améliore pas significativement la complexité de la méthode.

- **Preuve du programme :** La terminaison de la boucle `while` est évident dès lors que `k` est incrémenté de 1 à chaque tour de boucle. Pour la preuve de cet algorithme, on peut proposer l'invariant de boucle suivant :

  $\mathcal{P}(k)$: A la fin de la boucle où $i = k$, pour tout $j \leq k$, `t.(j)` vaut `true` si et seulement si `j` est un nombre premier. De plus, pour tout $j > k$, `t.(j)` vaut `true` si et seulement si `j` n'est divisible par aucun nombre premier inférieur ou égal à `k`.

- **Complexité :** Pour une analyse fine du crible d'Eratosthène, il convient de remarquer que le test ligne 5 n'est en pratique effectué que si `i` est un nombre premier. Par ailleurs, la boucle `while` s'exécute alors au plus $\lfloor n/i \rfloor + 1$ fois. La complexité globale est donc majorée par une quantité de l'ordre de

$$\sum_{\substack{i \leq n \\ i \text{ premier}}} O\left(\frac{n}{i}\right) + \sum_{\substack{i \leq n \\ i \text{ non premier}}} O(1) = n \cdot O\Big(\sum_{\substack{i \leq n \\ i \text{ premier}}} \frac{1}{i} \Big) + O(n)$$

  On peut démontrer que la somme des inverses des nombres premiers inférieur ou égaux à n a une croissance en $O(\ln(\ln n))$. Par conséquent, l'algorithme du crible d'Eratosthène calcule la liste des nombres premiers inférieurs ou égaux à n en $O(n \cdot \ln(\ln n))$ opérations élémentaires. Autant dire qu'à l'échelle des données humaines, il s'agit d'un programme de complexité linéaire.