Chapitre 6. Algorithmes fondamentaux

Table des matières

Remplissez le jupyter notebook suivant en vous aidant de votre livre de Première NSI de Serge BAYS .

- Pour répondre, double-cliquez sur **Réponse** et complétez la zone en-dessous. Puis cliquez sur le bouton *Exécuter*.
- **Important : pour fermer votre jupyter notebook, cliquez sur :**

Fichier / Créer une nouvelle sauvegarde

puis sur :

Fichier / Fermer et Arrêter

- Ecrivez ci-dessous votre prénom et votre nom :

Réponse :

Chapitre 6. Algorithmes fondamentaux

1. Les algorithmes élémentaires

1.1 Point histoire

Lisez la p. 275

1) Quel mathématicien perse du 9 ème siècle a écrit l'ouvrage 'Kitab al jabr wa muqabala' et dont le nom latinisé est à l'origine du mot algorithme ?

Réponse :

1.2 Introduction

Lisez le paragraphe **Introduction** et **Algorithme d'Euclide** p. 277 et en haut de la p. 278

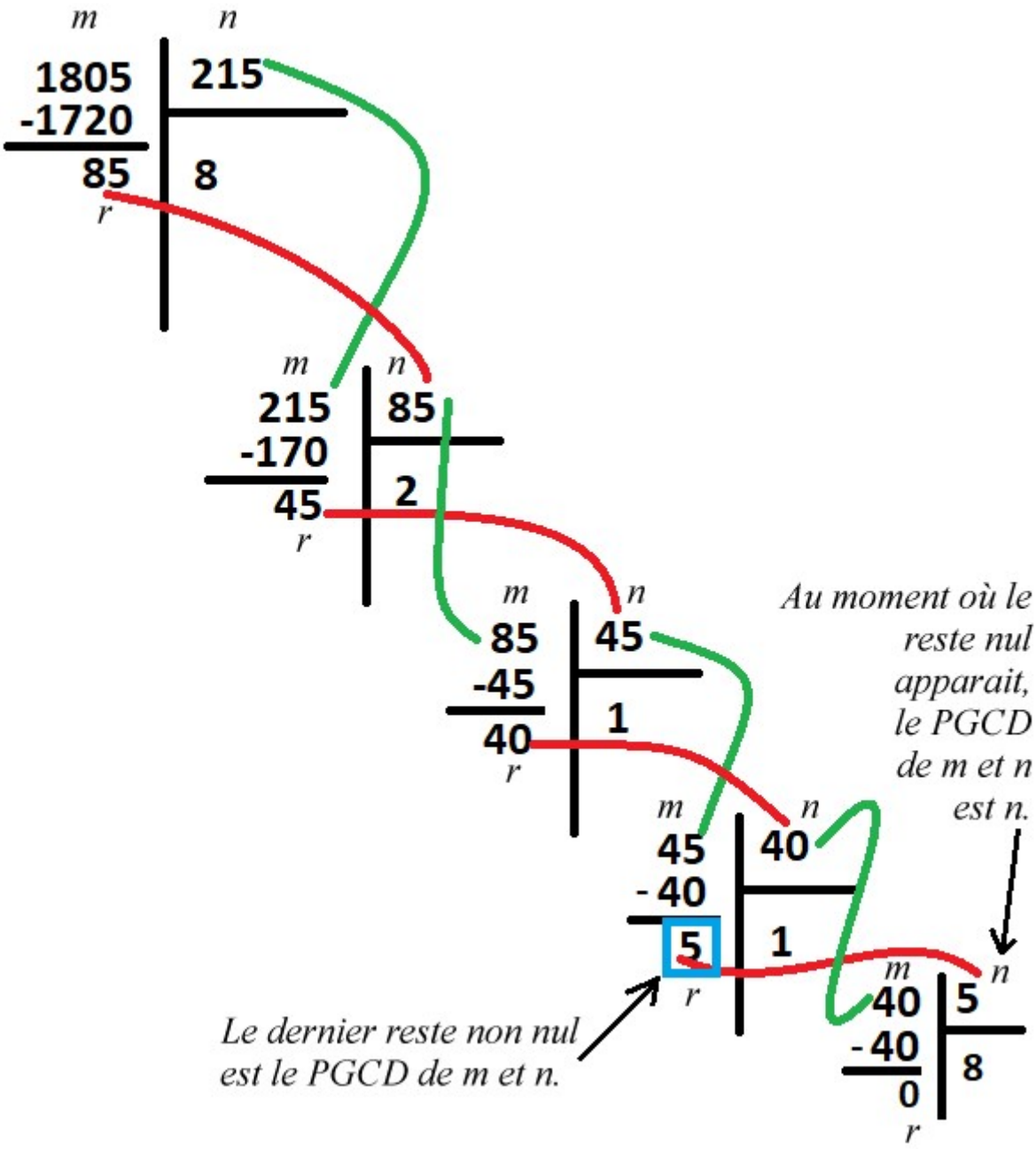
2) Quel type de problèmes servent à résoudre les algorithmes d'Al-Khwarizmi ?

Réponse :

3) Le mathématicien de la Grèce antique Euclide, dans le livre VII de son ouvrage 'Elements' expose le premier algorithme non trivial (c'est à dire non évident) pour trouver le PGCD de deux entiers m et n. Cet algorithme a été écrit il y a combien d'années ?

Réponse :

- Principe de l'algorithme d'Euclide : Exemple de la recherche du plus grand entier qui divise à la fois 1805 et 215.



On *implémente* (c'est à dire on *met en oeuvre*) l'algorithme de façon itérative :

In [ ]:
```
def pgcd(m, n):
    """
    Retourne le PGCD des entiers m et n

    Parametres nommes
    -----------------
        m et n de type int

    Retourne
    --------
        n de type int
        A la fin, n est le PGCD de m et n donnes en parametres.

    """

    r = m % n  # r est le reste de la division entiere de m par n.
    while r != 0:  # La condition d'arret est 'le reste r est nul'.
        m, n = n, r  # On passe a la division suivante par transfert de n
et r.
        r = m % n  #  On calcule le nouveau reste.
    return n
```

Testez dans la cellule ci-dessous la fonction pgcd avec des valeurs de m et n de votre choix :

In [ ]:
```
m =
n =
pgcd(m, n)
```

4) Quelle est la profession de Donald Knuth ? (voir l'article qui lui est consacré sur Wikipedia).

Réponse :

5) Quelle récompense Donald Knuth offre-t-il à quiconque trouve une erreur dans l'un de ses ouvrages ? (voir la rubrique 'personnalité' dans l'article sur Wikipedia).

Réponse :

6) Qu'est-ce qu'un algorithme doit toujours faire après un nombre fini d'étapes ?

Réponse :

7) Combien de paramètres donnés avant ou pendant son exécution un algorithme peut-il avoir ?

Réponse :

8) Combien de valeurs un algorithme peut-il retourner ?

Réponse :

9) Un algorithme doit avoir des instructions suffisamment basiques pour pouvoir, en principe, être exécuté de quelle façon ?

Réponse :

1.3 Les outils

1.3.1 Compteurs et accumulateurs

Lisez le paragraphe **Compteurs et accumulateurs** p. 278 et p. 279

10) Compteur dans une boucle while (sans test if) : écrivez ci-dessous une fonction taille(n) qui compte le nombre de chiffres en base 10 d'un entier n donné en paramètre.

In [ ]:
```
def taille(n):
    """
    Renvoie le nombre de chiffres de n écrit en base 10.
    Principe : calculer les quotients successifs dans la division entière par 10.

    Parametres nommes
    -----------------
        n de type int

    Retourne
    --------
        cpt de type n
        Le nombre de chiffres en base 10.


    """

    cpt = 0
    while n > 0:
        cpt = cpt + 1
        n = n // 10

    return cpt
```

Testez dans la cellule ci-dessous la fonction taille avec un entier n de votre choix :

In [ ]:
```
n =
taille(n)
```

11) Compteur dans une boucle while (avec un test) : écrivez ci-dessous une fonction nombre_de_7(n) qui compte le nombre de chiffres '7' d'un entier n écrit en base 10 donné en paramètre.

In [ ]:
```
def nombre_de_7(n):
    """
    Renvoie le nombre de chiffres 7 de n ecrit en base 10.
    Principe : calculer les restes successifs dans la division entière par
10.

    Parametres nommes
    -----------------
        n de type int

    Retourne
    --------
        cpt de type n
        Le nombre de chiffres 7.


    """
    cpt = 0
    while n > 0:



    return cpt
```

Testez dans la cellule ci-dessous la fonction taille avec un entier n de votre choix :

In [ ]:
```
n =
nombre_de_7(n)
```

12) Compteur dans une boucle for (sans un test) : Exécutez le programme ci-dessous.

In [ ]:
```
def taille(n):
    cpt = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                cpt = cpt +1
    return cpt

taille(5)
```

Comment expliquez-vous le résultat ?

Réponse :

13) Compteur dans une boucle for (avec test if) : La fonction diviseurs(n) compte le nombre de diviseurs positifs de l'entier n donné en argument.

In [ ]:
```
def diviseurs(n):
    """
    Renvoie le nombre de diviseurs positifs de n.
    Principe : tester si le reste de la division de n par tous les entiers de 1 à n est nul.

    Parametres nommes
    -----------------
        n de type int
        n est un entier naturel non nul dont on cherche le nombre de diviseurs positifs.

    Retourne
    --------
        cpt de type n
        Le nombre de diviseurs.


    """
    assert n != 0, "0 a une infinité de divisuers !"  #  Affiche un message d'erreur lorsque n = 0.

    cpt = 0  #  Initialisation du nombre de diviseurs de n.
    for d in range(1, n + 1):
        if n % d == 0:  # Calcul du reste de la division de n par d
            cpt = cpt + 1
    return cpt
```

Testez dans la cellule ci-dessous la fonction diviseurs(n) avec un entier n de votre choix :

In [ ]:
```
n =
diviseurs(n)
```

14) Ecrivez un programme qui affiche la liste des tuples [(1, 1), (2, 2), (3, 2), ... , (100, 9)] où les premiers éléments des tuples sont les entiers de 1 à 100 et où le deuxième élément est le nombre de diviseurs positifs du premier élément. Ce programme devra utiliser la fonction diviseurs(n) de la question précédente.

In [ ]:
```
#  Dans un premier temps création de f1 la fonction qui affiche
#  la liste de stuples [(1, 1), (2, 2_, (3, 3), ... , (100, 100))]

def f1():
    """
    Renvoie la liste des tuples [(1, 1), (2, 2), (3, 3), ... ,(100, 100)]

    Parametre nommes
    ----------------
        aucun

    Retourne
    --------
        L de type list

    """
```

In [ ]:
```
#  Dans un deuxième temps création de f2 la fonction
#  qui renvoie la liste des tuples (i, diviseurs(i))

def f2():
    """
    Renvoie la liste des tuples [(1, 1), (2, 2), (3, 2), ... ,(100, 9)]

    Parametre nommes
    ----------------
        aucun

    Retourne
    --------
        L de type list

    """
```

1.3.2 Permutations de valeurs

Lisez le paragraphe **Permutation de valeurs** en bas de la p. 279 et p. 280

15) Lorsqu'on a besoin d'échanger les valeurs des variables var1 et var2, on a deux méthodes. Méthode n°1 : elle consiste à utiliser une troisième variable temporaire temp pour stocker l'ancienne valeur de var1 avant de l'affecter à var2. Complétez le code dans la cellule ci-dessous et testez son bon fonctionnement.

In [ ]:
```
var1 = 17
var2 = 23
temp = ...
var1 = var2
var2 = temp
print("var1 = ", var1, " et var2 = ", var2)
```

16) Méthode n°2, lorsqu'on a besoin d'échanger les valeurs des variables var1 et var2, on peut aussi utiliser les tuples. Complétez le code dans la cellule ci-dessous et testez son bon fonctionnement.

In [ ]:
```
var1 = 17
var2 = 23
var1, var2 = ..., ...
print("var1 = ", var1, " et var2 = ", var2)
```

1.3.3 Tests et boucles

Lisez le paragraphe **Tests et boucles** p. 280, p. 281 et en haut de la p. 282

17) Complétez le programme suivant pour qu'il retourne " 2 racines distinctes " ou " 1 racine double " ou " Pas de racine ". Vous utiliserez une structure if ... elif ... else ...

In [ ]:
```
def nombre_de_racines(a, b, c):
    delta = ...
    if ...:
        return "2 racines distinctes"
    elif ...:
        return "1 racine double"
    else:
        return "Pas de racine"
```

- Testez, dans la cellule ci-dessous, votre programme nombre_de_racines() en choisissant des valeurs de a, b, c prévues pour avoir 2 racines distinctes :

In [ ]:
```
print(nombre_de_racines(...))
```

- Testez, dans la cellule ci-dessous, votre programme nombre_de_racines() en choisissant des valeurs de a, b, c prévues pour avoir 1 racine double :

In [ ]:
```
print(nombre_de_racines(...))
```

- Testez, dans la cellule ci-dessous, votre programme nombre_de_racines() en choisissant des valeurs de a, b, c prévues pour qu'il n'y ait pas de racine :

In [ ]:
```
print(nombre_de_racines(...))
```

18) Testez les fonctions test1 et test2 en prenant comme valeur initiale 5 pour la variable x.

In [ ]:
```
def test1(x):
    if x > 0:
        x = x - 3
    elif x < 0:
        x = x + 5
    else:
        x = x + 2
    return x

def test2(x):
    if x > 0:
        x = x - 3
    if x < 0:
        x = x + 5
    else:
        x = x + 2
    return x
```

In [ ]:
```
print("L'image de 5 par la fonction test1 est ", test1(5))
print("L'image de 5 par la fonction test2 est ", test2(5))
```

- Expliquez pourquoi la fonction test2 donne un résultat différent :

Réponse :

19) Testez la fonction test3 en prenant comme valeur initiale 5 pour la variable x. Expliquez le résultat obtenu.

In [ ]:
```
def test3(x):
    if x > 0:
        x = x - 3
    if True:
        x = x + 5
    else:
        x = x + 2
    return x
```

In [ ]:
```
print("L'image de 5 par la fonction test3 est ", test3(5))
```

Réponse :

20) Testez la fonction test4 en prenant comme valeur initiale 5 pour la variable x. Expliquez le résultat obtenu.

In [ ]:
```
def test4(x):
    if x > 0:
        x = x - 3
    elif True:
        x = x + 5
    else:
        x = x + 2
    return x
```

In [ ]:
```
print("L'image de 5 par la fonction test4 est ", test4(5))
```

Réponse :

1.4 Validité et coût d'un algorithme

Lisez le paragraphe **Validité et coût** en haut de la p. 282

21) Pour vérifier la validité d'un algorithme on doit vérifier deux points : La correction et la terminaison. Expliquez ce que signifient ces deux mots :

Réponse :

1.4.1 Validité d'un algorithme itératif

Lisez le paragraphe **Validité d'un algorithme itératif** p. 282 et la première moitié de la p. 283

22) Un algorithme itératif est un algorithme qui contient au moins une boucle. De quelle notion disposons-nous pour prouver qu'un algorithme itératif est correct ?

Réponse :

23) Un invariant de boucle est une propriété qui est vraie à quels endroits par rapport à une boucle ?

Réponse :

Etude de la validité d'un algorithme

Il faut étudier :

1) La correction de l'algorithme.

2) La terminaison de l'algorithme.

- Exemple : Soit l'algorithme qui calcule le produit $p = a \times b$ (a est le multiplicateur et b un entier).

```
m = 0  #  Initialisation de la variable "compteur de boucles" m.
p = 0  #  Initialisation du produit p.


tant que m < a :
    m = m + 1  #  Compteur de boucles.
    p = p + b  #  Accumulateur de b a chaque tour de boucle.
fin du tant que
```

Pour comprendre l'algorithme, faisons le fonctionner sur un exemple.

avec $a = 4$ et $b = 3$:

a	b	m	p	m < a
4				
	3			
		0		
			0	
				Vrai
		1		
			3	
				Vrai
		2		
			6	
				Vrai
		3		
			9	
				Vrai
		4		
			12	
				Faux

Il y a eu $a = 4$ accumulations de nombres $b = 3$ sur la variable p qui valait au départ 0.

Un exemple n'est pas suffisant pour prouver la validité d'un algorithme

1) Etudions la correction de l'algorithme grâce à un invariant de boucle.

Montrons que pour chaque tour de boucle la relation $p = m \times b$ est vraie (ce sera l'invariant de boucle pour cet algorithme).

On procède en trois étapes : initialisation, hérédité, conclusion.

- Initialisation :

Avant la boucle, b = 3, m = 0 et p = 0. Donc $p = m \times b$ est vraie.

- Hérédité :

Hypothèse : pour un certain tour de boucle, on a $p = m \times b$.

Montrons qu'alors, au tour suivant, on a encore $p' = m' \times b'$ en utilisant les données de l'algorithme b' = 3, m' = m + 1, p' = p + b.

Partons de l'hypothèse et transformons-la à l'aide des données :

$p = m \times b.$

$p + b = m \times b + b.$

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

$p' = m' \times b'.$

- Conclusion : La propriété $p = m \times b$ est vraie au départ et elle est héréditaire. Donc elle est vraie quel que soit le nombre de tours de boucles.

Donc après $m = a$ tours de boucles, la propriété et vraie, c'est à dire qu'en sortie de boucle on a :

$p = a \times b.$

On vient de prouver la correction de l'algorithme (autrement dit que l'algorithme est correct).

2) Justifions la terminaison de l'algorithme.

- Examinons la condition de maintien dans la boucle while : m < a.
  - Si m > = a au départ alors la boucle while n'est jamais exécutée et l'algorithme se termine.
  - Si m < a au départ, puisque dans le corps de la boucle il y a l'instruction m = m + 1, alors la condition de maintien dans la boucle while devient fausse après un nombre fini de tours et l'algorithme s'arrête.

On vient de prouver la terminaison de l'algorithme (autrement dit que l'algorithme se termine).

- En conclusion :

```
- L'algorithme est correct.

- L'algorithme se termine.
```

Donc l'algorithme de la multiplication est *valide.*

1.4.2 Coût d'un algorithme

Lisez le paragraphe **Coût** en bas de la p. 283, p. 284 et p.285

- Le *coût* en temps d'un algorithme (on dit aussi la *complexité*) dépend du type d'instructions qu'il contient. On se place "dans le pire des cas" pour l'évaluer. Par exemple pour un algorithme de tri, on se place dans le cas où la liste à trier est totalement désordonnée.

- Exemple le l'algorithme de la multiplication $p = a \times b$. Voyez le code ci-dessous.

In [ ]:
```
def produit(a, b):
    """
    Fait le produit de a par b

    Parametres nommes
    -----------------
        a, b: de type int
        Les opérandes de la multiplication

    Retourne
    --------
        p: de type int
        Le résultat du produit

    """

    m = 0
    p = 0
    while m < a:
        m = m + 1
        p = p + b
    return p
```

- Faites le produit de a = 1e6 par b = 50 dans la cellule ci-dessous.

In [ ]:
```
from time import time
time_start = time()
produit(...)
time_end = time()
print("duree = ", time_end - time_start)
```

- Faites le produit de a = 1e7 par b = 50 dans la cellule ci-dessous.

In [ ]:
```
from time import time
time_start = time()
produit(...)
time_end = time()
print("duree = ", time_end - time_start)
```

Complexité linéaire : La durée mise pour faire 10 fois plus de tours de boucle est environ 10 fois plus grande. On dit que le coût en temps de cet algorithme est **linéaire** en fonction de la taille du nombre a.

Dire qu'un algorithme est de **complexité linéaire**, signifie que s'il a n éléments à traiter (par exemple une liste de longueur n) alors il mettra un temps de l'ordre de n, ce qui signifie une fonction affine de n. On dit que la complexité est en $O\left(n\right)$.

Cas d'un algorithme qui a deux boucles imbriquées

- Exemple 1 :

On veut créer une fonction qui calcule les produits $i \times j$ pour toutes les cellules d'un tableau de taille $\left(n+1\right) \times \left(n+1\right)$:

	j = 0	**j = 1**	**j = 2**	**j = 3**	**j = 4**	**. . .**	**. . .**	**j = n**
i = 0	0	0	0	0	0	...	...	...
i = 1	0	1	2	3	4	...	...	...
i = 2	0	2	4	6	8	...	...	...
i = 3	0	3	6	9	12	...	...	...
i = 4	0	4	8	12	16	...	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
i = n	...	...	...	...	...	...	...	...

In [ ]:
```
def tableau_carre(n):
    """
    Calcule les produits i * j pour i allant de 0 à n et pour j allant de
0 à n.
    Ainsi, le tableau contient (n + 1) * (n + 1) valeurs calculees.

    Parametres nommes
    -----------------
        n de type entier
        La taille du tableau carre est (n + 1) * (n + 1).

    Retourne
    --------
        L de type list
        La liste de tous les produits i * j

    """
    L = []
    for i in range(0, n + 1):
        for j in range(0, n + 1):
            L.append(i * j)
    return L
```

- Exécutez dans la cellule de code ci-dessous la fonction tableau_carre pour n = 500. On n'affichera pas la liste L, mais on mesurera le temps mis pour ce calcul.

In [ ]:
```
from time import time
time_start = time()
tableau_carre(...)
time_end = time()
print("duree = ", time_end - time_start)
```

- Exécutez dans la cellule de code ci-dessous la fonction tableau_carre en multipliant par 10 la valeur de n. On n'affichera pas la liste L, mais on mesurera le temps mis pour ce calcul.

In [ ]:
```
from time import time
time_start = time()
tableau_carre(...)
time_end = time()
print("duree = ", time_end - time_start)
```

Complexité quadratique : La durée mise pour un nombre n 10 fois plus grand est environ 10 au carré fois plus grande (100 fois plus grande). On dit que le coût en temps de cet algorithme est **quadratique** c'est à dire, en gros, proportionnel au carré en fonction de la taille du nombre n. Plus précisément, la durée de calcul est une fonction polynôme du second degré de n.

Dire qu'un algorithme est de **complexité quadratique**, signifie que s'il a n éléments à traiter (par exemple une liste de longueur n) alors il mettra un temps de l'ordre de n^2. On dit que la complexité est en $O\left(n^2\right)$.

- Exemple 2 :

On veut créer une fonction qui calcule les produits $i \times j$ pour *certaines cellules* d'un tableau de taille $(n+1) \times (n+1)$.

i est le numéro de ligne et va de 0 à n.

Mais pour une cellule (i, j) donnée, son numéro de colonne ne peut pas être supérieur à son numéro de ligne. Par exemple (3, 2) et (3, 3) sont possibles, mais pas (3, 4). Autrement dit, on a un tableau triangulaire :

	j = 0	**j = 1**	**j = 2**	**j = 3**	**j = 4**	**...**	**...**	**j = n**
i = 0	0							
i = 1	0	1						
i = 2	0	2	4					
i = 3	0	3	6	9				
i = 4	0	4	8	12	16			
...	...	...	...	...	...	...		
...	...	...	...	...	...	...	...	
i = n	...	...	...	...	...	...	...	...

In [ ]:
```
def tableau_triangulaire(n):
    """
    Calcule les produits i * j pour i allant de 0 à n et j allant de 0 à
i.


    Parametres nommes
    -----------------
        n de type entier
        La taille du tableau triangulaire.

    Retourne
    --------
        L de type list
        La liste de tous les produits i * j

    """
    L = []
    for i in range(0, n + 1):
        for j in range(0, i):
            L.append(i*j)
    return L
```

- Exécutez dans la cellule de code ci-dessous la fonction tableau_triangulaire pour n = 500. On n'affichera pas la liste L, mais on mesurera le temps mis pour ce calcul.

In [ ]:
```
from time import time
time_start = time()
tableau_triangulaire(...)
time_end = time()
print("duree = ", time_end - time_start)
```

- Exécutez dans la cellule de code ci-dessous la fonction tableau_triangulaire en multipliant par 10 la valeur de n. On n'affichera pas la liste L, mais on mesurera le temps mis pour ce calcul.

In [ ]:
```
from time import time
time_start = time()
tableau_triangulaire(...)
time_end = time()
print("duree = ", time_end - time_start)
```

Cette fois la boucle intérieure n'est pas pour j allant de 0 à n. Nous sommes dans le cas où la boucle intérieure est pour j allant de 0 à i. Le calcul est plus rapide.

Mais la complexité est **quadratique** comme pour le tableau carré de l'exemple 1. La durée mise pour un nombre n 10 fois plus grand est *environ* 10^2 fois plus grande (100 fois plus grande). La complexité est en $O\left(n^2\right)$. C'est normal puisque l'aire du triangle dépend du produit $base \times hauteur$. Si on multiplie par 10 la base et par 10 la hauteur alors l'aire est multipliée par 10^2.

1.5 Parcours séquentiel

1.5.1 Calcul d'une moyenne

Lisez le paragraphe **Calcul d'une moyenne** en bas de la p. 285 et en haut de la p. 286

- Calcul de la moyenne des éléments d'une liste

24) Saisissez dans la cellule ci-dessous le code donné en haut de la page 286. Puis, dans la cellule encore en-dessous, vérifiez que le coût de la fonction moyenne(liste) est de l'ordre de n

In [ ]:
```
def moyenne(liste):
    n = ...
    s = ...
    for u in ...:
        s = s + u  #  Accumule la somme des elements de la liste.
    return ...
```

In [ ]:
```
# Creation de la liste des n premiers entiers naturels non nuls
n = int(1e6)
ma_liste = [i for i in range(1, n + 1)]

# Mesure de la duree du calcul de la moyenne
from time import time
time_start = time()
moyenne(ma_liste)
time_end = time()
print("duree = ", time_end - time_start)
```

Réponse :

Le temps mis pour $n = 10^6$ est $t_{10^6} = \ldots s$

Le temps mis pour $n = 10^7$ est $t_{10^7} = \ldots s$

La complexité en temps de l'algorithme de calcul de moyenne est ...

1.5.2 Recherche d'une occurrence

Lisez le paragraphe **Recherche d'une occurrence** p. 286 et en haut de la p. 287

25) a) Saisissez dans la cellule ci-dessous le code donné en bas de la page 286. Puis, dans la cellule encore en-dessous, vérifiez que le coût de la fonction recherche(x, t) où x est un élément et t une liste (ou éventuellement un tuple ou une chaîne) est de l'ordre de n

In [ ]:
```
def recherche(x, t):
    n = ...
    i = ...
    while ... and ...:
        i = i + 1  #  Incremente l'indice.
    if i < n:  #  Si on est sorti de la boucle parce que x != t[i] est Fal
se.
        return i  #  Retourne l'indice de l'element qu'on cherchait.
```

In [ ]:
```
# Creation de la liste des n premiers entiers naturels non nuls.
n = int(1e6)
ma_liste = [i for i in range(1, n + 1)]

# Mesure de la duree de la recherche d'une occurrence d'un element.
from time import time
time_start = time()
recherche(int(0.5 * n), ma_liste)  #  Recherche un element en milieu de li
ste.
time_end = time()
print("duree = ", time_end - time_start)
```

Réponse :

Le temps mis pour $n = 10^6$ est $t_{10^6} = \ldots\ s$

Le temps mis pour $n = 10^7$ est $t_{10^7} = \ldots\ s$

La complexité en temps de l'algorithme de recherche de l'indice d'un élément est ...

25) b) On cherche maintenant un élément x qui n'est certainement pas dans la liste. Vérifiez que le coût de la fonction recherche(x, t) est encore de l'ordre de n

In [ ]:
```
# Creation de la liste des n premiers entiers naturels non nuls.
n = int(1e6)
ma_liste = [i for i in range(1, n + 1)]

# Mesure de la duree de la recherche d'une occurrence d'un element.
from time import time
time_start = time()
recherche(-1, ma_liste)  #  Recherche de -1 qui n'est pas dans la liste.
time_end = time()
print("duree = ", time_end - time_start)
```

Réponse :

Le temps mis pour $n = 10^6$ est $t_{10^6} = \ldots\ s$

Le temps mis pour $n = 10^7$ est $t_{10^7} = \ldots\ s$

La complexité en temps de l'algorithme de recherche de l'indice d'un élément est ...

1.5.3 Recherche d'un extremum

Lisez le paragraphe **Recherche d'un extremum** p. 287 et en haut de la p. 288

- Une fonction qui remplit une liste de n entiers aléatoires (ces entiers sont entre 0 et 10000)

26) Observez le code de la fonction remplit_liste dans la cellule ci-dessous. Dans la cellule encore en-dessous, faites la fonctionner pour quelques valeurs de n de votre choix.

In [ ]:
```
def remplit_liste(n):
    """
    Retourne une liste de n entiers aleatoires entre 0 et 10000.

    Parametres nommes
    -----------------
        n : de type int
        La longueur de la liste aleatoire retournee.

    Retourne
    --------
        L : de type list
        Une liste de n entiers

    """

    from random import randint
    L = []
    for i in range(n):
        L.append(randint(0, 10000))
    return L
```

In [ ]:
```
remplit_liste(...)
```

27) Maintenant, on utilise la fonction remplit_liste. On veut trouver son maximum. Dans la cellule ci-dessous, recopiez la fonction maximum(liste) qui est au milieu de la page 287. Puis dans la cellule encore après, recherchez le maximum de la liste ma_liste en donnant à n différentes valeurs.

In [ ]:
```
def maximum(liste):
    maxi = ...
    for x in ...:
        if x > ...:
            maxi = ...
    return maxi
```

In [ ]:
```
ma_liste = remplit_liste(100)
print(maximum(ma_liste))
```

28) Quelle est la complexité de l'algorithme de la fonction de recherche du maximum ? Vous répondrez sans faire de mesures de durées, mais en donnant une justification théorique.

Réponse :

29) En vous inspirant de la fonction maximum, écrivez ci-dessous une fonction minimum qui retourne le plus petit élément d'une liste. Faites-la fonctionner sur quelques exemples de listes ma_liste(n) en donnant à n des valeurs de votre choix

In [ ]:
```
def minimum(liste):
    mini = ...
    for x in ...:
        if x < ...:
            mini = ...
    return mini
```

In [ ]:
```
ma_liste = remplit_liste(100)
print(minimum(ma_liste))
```

- L'algorithme utilise encore un parcours séquentiel de la liste de longueur n (c'est à dire avec des indices qui se suivent de 1 en 1 depuis 0). Donc sa complexité est en $O(n)$ (c'est à dire linéaire).

Recherche du maximum d'un phénomène continu

- On va voir qu'on peut tracer un graphique en important la bibliothèque matplotlib.
- On va voir aussi qu'on peut traiter les problèmes " continus " où la variable x évolue sur un intervalle de l'ensemble des nombres réels en " échantillonnant " les valeurs de x. On se ramène ainsi au problème déjà vu des valeurs " discrètes " qui sont dans les listes.

- Exécutez le code ci-dessous :

In [ ]:
```
#  Importation des bibliotheques permettant les graphiques
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#  Definition de la fonction
def f(x):
    y = -3*x**2 + 8*x + 1
    return y

# Choix de la grille, du nom de la figure et des axes
plt.style.use('seaborn-whitegrid')
plt.title(" Fonction continue sur [0 ; 2]")
plt.xlabel("x")
plt.ylabel("y")


# Choix de la fenetre et execution du graphe
x = np.linspace(0, 2, 1000)
plt.plot(x, f(x))
```

- Comment trouver une valeur approchée du maximum sur l'intervalle [0 ; 2] ?

La fonction f est continue sur l'intervalle [0 ; 2]. On se ramène au cas d'une liste en prenant n + 1 points sur l'intervalle[0 ; 2].

Par exemple, choisissons $n = 10$.

On place donc sur l'intervalle [0 ; 2] 11 points séparés par des petits intervalles d'amplitude $dx = \frac{2-0}{10} = 0,2$.

- Exécutez le code ci-dessous qui affiche en **rouge** les valeurs de x pour lesquelles on va prendre des échantillons.

In [ ]:

```
#  Importation des bibliotheques permettant les graphiques
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#  Definition de la fonction
def f(x):
    y = -3*x**2 + 8*x + 1
    return y

# Choix de la grille, du nom de la figure et des axes
plt.style.use('seaborn-whitegrid')
plt.title(" Fonction continue sur [0 ; 2]")
plt.xlabel("x")
plt.ylabel("y")


# Choix de la fenetre et execution du graphe
x = np.linspace(0, 2, 1000)
plt.plot(x, f(x))

# Trace des 11 points sur l'axe (Ox) espaces de dx = 0,2
x = [i/5 for i in range(11)]
y = [0 for i in range(11)]
plt.scatter(x, y, s = 3, c = 'red')
```

- On utilise une boucle *for* comme pour la recherche du maximum d'une liste. Sauf qu'à chaque tour de boucle, on calcule f(x). *maxi* est initialisé avec f(0). Si f(x) > *maxi* alors ce f(x) devient le nouveau *maxi*. A la fin de la boucle, *maxi* représente le maximum de toutes les valeurs testées.

- Exécutez le code ci-dessous qui affiche en **vert** les points dont les ordonnées sont les valeurs des échantillons prélevés.

In [ ]:

```
#  Importation des bibliotheques permettant les graphiques
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#  Definition de la fonction
def f(x):
    y = -3*x**2 + 8*x + 1
    return y

# Choix de la grille, du nom de la figure et des axes
plt.style.use('seaborn-whitegrid')
plt.title(" Fonction continue sur [0 ; 2]")
plt.xlabel("x")
plt.ylabel("y")


# Choix de la fenetre et affichage de la courbe.
x = np.linspace(0, 2, 1000)
plt.plot(x, f(x))

# Creation des coordonnees des 11 points rouges sur l'axe (Ox) espaces de
dx = 0,2
x = [i/5 for i in range(11)]
y = [0 for i in range(11)]

# Creation de la liste y2 des valeurs des echantillons
y2 = [f(j) for j in x]

# Affichage des points rouges sur (Ox) et des points verts sur la courbe.
plt.scatter(x, y, s = 3, c = 'red')
plt.scatter(x, y2, s = 30, c = 'green')
```

- Pour connaitre la valeur du maximum, avec plus de précision, il suffit de diviser l'intervalle [0 ; 2] en davantage de petits intervalles. Par exemple $n = 100$.

- Exécutez le code ci-dessous :

In [ ]:
```
#  Importation des bibliotheques permettant les graphiques
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#  Definition de la fonction
def f(x):
    y = -3*x**2 + 8*x + 1
    return y

# Choix de la grille, du nom de la figure et des axes
plt.style.use('seaborn-whitegrid')
plt.title(" Fonction continue sur [0 ; 2]")
plt.xlabel("x")
plt.ylabel("y")


# Choix de la fenetre et execution du graphe
x = np.linspace(0, 2, 1000)
plt.plot(x, f(x))

# Trace des 11 points sur l'axe (Ox) espaces de dx = 0,02
x = [i/50 for i in range(101)]
y = [0 for i in range(101)]
y2 = [f(j) for j in x]
plt.scatter(x, y, s = 3, c = 'red')
plt.scatter(x, y2, s = 3, c = 'green')
```

30) Recopiez et exécutez ci-dessous le code Python de la fonction maximum(f, a, b, n) donnée en bas de la page 287 :

In [ ]:
```
def maximum(f, a, b, n):
    maxi = ...
    dx = ...
    x = a
    for k in range(n):
        x = ...
        y = ...
        if y > ...:
            maxi = ...
    return maxi
```

31) Exécutez dans la fenêtre ci-dessous cette fonction avec comme valeurs de paramètres (f, 0, 2, 10).

In [ ]:
```
maxi = maximum(...)
print(maxi)
```

- Quelle valeur de maxi obtenez-vous ?

Réponse avec n = 10 :

32) Exécutez dans la fenêtre ci-dessous cette fonction avec comme valeurs de paramètres (f, 0, 2, 100).

In [ ]:
```
maxi = maximum(...)
print(maxi)
```

- Quelle valeur de maxi obtenez-vous ?

Réponse avec n = 100 :

33) Des deux réponses, laquelle est la plus précise ? celle avec n = 10 ou celle avec n = 100 ? Justifiez.

Réponse :

2. L'algorithme de recherche dichotomique

2.1 Le principe

Lisez le paragraphe **Recherche dichotomique** p. 288 et tout en haut de la p. 289

- Commençons par un exemple :

Soit la liste de cent nombres pris au hasard parmi les mille premiers entiers naturels :

In [1]:
```
ma_liste =  [852, 262, 876, 633, 602, 292, 927, 155, 804, 702, 358, 557, 3
0, 259, 975, 572, 971, 557, 747, 727, 627, 540, 128, 66, 272, 449, 98, 23
8, 887, 351, 810, 416, 601, 996, 999, 513, 615, 682, 368, 270, 397, 633, 2
54, 861, 725, 966, 777, 873, 590, 285, 138, 447, 302, 516, 174, 662, 93, 1
60, 711, 410, 806, 856, 931, 893, 178, 352, 800, 984, 250, 613, 448, 111,
161, 471, 42, 68, 160, 11, 85, 127, 43, 471, 686, 771, 935, 208, 382, 495,
487, 852, 309, 780, 544, 635, 658, 698, 381, 265, 447, 231]
print("ma_liste = ", ma_liste)
```

```
ma_liste =  [852, 262, 876, 633, 602, 292, 927, 155, 804, 702, 358, 5
57, 30, 259, 975, 572, 971, 557, 747, 727, 627, 540, 128, 66, 272, 44
9, 98, 238, 887, 351, 810, 416, 601, 996, 999, 513, 615, 682, 368, 27
0, 397, 633, 254, 861, 725, 966, 777, 873, 590, 285, 138, 447, 302, 5
16, 174, 662, 93, 160, 711, 410, 806, 856, 931, 893, 178, 352, 800, 9
84, 250, 613, 448, 111, 161, 471, 42, 68, 160, 11, 85, 127, 43, 471,
686, 771, 935, 208, 382, 495, 487, 852, 309, 780, 544, 635, 658, 698,
381, 265, 447, 231]
```

- Comment savoir si le nombre 207 figure dans cette liste ?

1) Trier la liste par ordre croissant :

- Pour le moment, nous n'avons pas encore vu les algorithmes de tri.

On va donc utiliser la *fonction sorted(ma_liste)* intégrée de Python qui s'appelle par :

```
ma_liste_triee = sorted(ma_liste)
```

34) Dans la zone de code ci-dessous affichez la liste ma_liste_triee :

In [3]:
```
ma_liste_triee = sorted(ma_liste)
print("ma_liste_triee = ", ...)
print("\nlongueur de ma_liste_triee = ", ...)
```

```
ma_liste_triee =  Ellipsis

longueur de ma_liste_triee =  Ellipsis
```

Voici l'algorithme. Ensuite après une brève explication théorique, on donne un exemple pour expliquer le fonctionnement.

In [4]:
```
def dichotomie(x, liste):
    """
    Recherche par dichotomie dans liste triee.

    Parametres nommes
    -----------------
        x : de type int
        C'est l'element qu'on cherche

        liste : de type list
        C'est la liste dans laquelle on cherche x.

        Retourne
        --------
        L'indice de x si x est un element de liste.
        False si x n'est pas un element de liste.

    """

    g = 0
    d = len(liste)
    assert liste[g] <= x <= liste[d-1], "x est en dehors"

    while g < d - 1:
        k = (g + d) // 2 #  Calcul de l'indice du milieu.
        if x < liste[k]:
            d = k
        else:
            g = k

    if x == liste[g]:
        return g
    else:
        return False  #  False lorsque x n'est pas trouvé.
```

2) Recherche de l'élément x par la méthode de dichotomie dans la liste triée que nous appelons "liste" :

Une brève explication théorique

Dichotomie signifie "couper en deux".

- On travaille avec une boucle while.

- A chaque tour de boucle, on considère les éléments de liste[g] à liste[d - 1] où g est l'indice de l'élément situé à gauche et d - 1 est l'indice de l'élément situé à droite et on compare x avec l'élément d'indice k = (g + d) // 2 c'est à dire l'élément du centre.

- Si x < liste[k] alors au tour suivant on a d = k. Sinon on a g = k.

- Ainsi à chaque tour, on a coupé en deux la partie de liste du tour précédent.

Un exemple pour expliquer le fonctionnement : Recherche de x = 207 dans ma_liste_triee :

Commencez par visualiser ici (http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/dichotomie/dichotomie.html) le déroulement de la recherche dichotomique pour un élément **qui n'est pas** dans la liste.

Etape 1

L'indice du premier élément (situé à gauche) est g = 0.

L'indice du dernier élément (situé à droite) est d - 1 = 99 avec d = len(ma_liste_triee) = 100.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 50.

Que vaut l'élément ma_liste_triee[50] ?

In [5]:
```
#  Si on fait un for i in ma_liste,
#  alors i prend tour à tour comme valeurs les éléments de ma_liste.

#  Si on fait un for i in enumerate(ma_liste),
#  alors i prend tour à tour comme valeurs les couples (indice, élément)

liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[50] = ...

- La condition x < liste[k] est vraie car 207 < 513 donc d = k = 50.

On déduit que si x = 207 est dans la liste alors c'est dans la partie depuis liste[0] jusqu'à liste[49] inclus.

Etape 2

L'indice du premier élément (situé à gauche) est g = 0.

L'indice du dernier élément (situé à droite) devient d - 1 = 49 avec d = k = 50.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 25.

Que vaut l'élément ma_liste_triee[25] ?

In [6]:
```
liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[25] = ...

- La condition x < liste[k] est vraie car 207 < 262 donc d = k = 25.

On déduit que si x = 207 est dans la liste alors c'est dans la partie liste[0] et liste[24].

Etape 3

L'indice du premier élément (situé à gauche) est g = 0.

L'indice du dernier élément (situé à droite) est d - 1 = 24 avec d = k = 25.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 12.

Que vaut l'élément ma_liste_triee[12] ?

In [7]:
```
liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[12] = ...

- La condition x < liste[k] est fausse car 207 >= 138 donc g = k = 12.

On déduit que si x = 207 est dans la liste alors c'est dans la partie liste[12] et liste[24].

Etape 4

L'indice du premier élément (situé à gauche) est g = k = 12.

L'indice du dernier élément (situé à droite) est d - 1 = 24 avec d = 25.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 18.

Que vaut l'élément ma_liste_triee[18] ?

In [8]:
```
liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[18] = ...

- La condition x < liste[k] est fausse car 207 >= 178 donc g = k =18.

On déduit que si x = 207 est dans la liste alors c'est dans la partie liste[18] et liste[24].

Etape 5

L'indice du premier élément (situé à gauche) est g = k = 18.

L'indice du dernier élément (situé à droite) est d - 1 = 24 avec d = 25.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 21.

Que vaut l'élément ma_liste_triee[21] ?

In [9]:
```
liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[21] = ...

- La condition x < liste[k] est vraie car 207 < 238 donc d = k = 21.

On déduit que si x = 207 est dans la liste alors c'est dans la partie liste[18] et liste[20].

Etape 6

L'indice du premier élément (situé à gauche) est g = 18.

L'indice du dernier élément (situé à droite) est d - 1 = 20 avec d = 21.

L'indice de l'élément situé au milieu est k = (g + d) // 2 = 19.

Que vaut l'élément ma_liste_triee[19] ?

In [10]:
```
liste_triee_avec_index = []
for i in enumerate(ma_liste_triee):
    liste_triee_avec_index.append(i)
print("liste_triee_avec_index = ", liste_triee_avec_index)
```

```
liste_triee_avec_index =  [(0, 11), (1, 30), (2, 42), (3, 43), (4, 6
6), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 12
8), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174),
(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (2
4, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30,
292), (31, 302), (32, 309), (33, 351), (34, 352), (35, 358), (36, 36
8), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (4
9, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55,
557), (56, 572), (57, 590), (58, 601), (59, 602), (60, 613), (61, 61
5), (62, 627), (63, 633), (64, 633), (65, 635), (66, 658), (67, 662),
(68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725), (7
4, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80,
804), (81, 806), (82, 810), (83, 852), (84, 852), (85, 856), (86, 86
1), (87, 873), (88, 876), (89, 887), (90, 893), (91, 927), (92, 931),
(93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 996), (9
9, 999)]
```

On voit que ma_liste_triee[19] = ...

- La condition x < liste[k] est vraie car 207 < 208 donc d = k = 19.

On déduit que si x = 207 est dans la liste alors c'est dans la partie allant depuis liste[18] jusqu'à liste[18] inclus.

Fin de la boucle while

L'indice du premier élément (situé à gauche) est g = 18.

L'indice du dernier élément (situé à droite) est d - 1 = 18 avec d = 19.

- La condition de maintien dans la boucle while est $d - g > 1$. Elle est désormais fausse.
- La boucle while est terminée.

- On teste si liste[g] vaut x = 207. C'est faux.

Conclusion : Donc 207 n'est pas dans la liste.

Testez dans la zone de code ci-dessous si x = 207 est dans la liste *ma_liste_triee* :

In [13]:
```
x =  207

print(dichotomie(x, ma_liste_triee))
```

```
False
```

Maintenant visualisez [là (http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/dichotomie2/dichotomie2.html)](http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/dichotomie2/dichotomie2.html) le déroulement de la recherche dichotomique d'un élément **qui est** dans la liste. Le déroulement est le même sauf à la fin où l'indice de l'élément recherché est renvoyé.

Testez dans la zone de code ci-dessous si x = 208 est dans la liste *ma_liste_triee* :

In [14]:
```
x =  208

print(dichotomie(x, ma_liste_triee))
```

```
19
```

- Examinons la validité de l'algorithme de recherche dans une liste triée par la méthode de dichotomie.

2.2 Preuve de la terminaison

Lisez le paragraphe **Preuve de la terminaison** en haut de la p. 289

35) Quel variant de boucle permet de prouver la terminaison ?

Réponse :

36) Si une liste a moins de 2^n éléments, au bout de combien de tours de boucles est-on certain que l'algorithme de recherche par dichotomie s'arrête ?

Réponse :

37) Dans l'exemple de ma_liste_triee de longueur 100, l'algorithme s'est terminé après 7 tours de boucles. Expliquez pourquoi c'était prévisible.

Réponse :

38) Si la liste avait une longueur 10 fois plus grande (longueur 1000), faudrait-il 10 fois plus de tours de boucles ? Justifiez.

Réponse :

Remarque :

- Un algorithme de recherche séquentielle, c'est à dire élément par élément, a un coût linéaire. C'est à dire qui si on multiplie par 10 la longueur n de la liste, le temps de recherche moyen est aussi multiplié par 10 puisqu'il faut explorer toute la liste.

- L'algorithme de recherche par dichotomie ne fonctionne que sur une liste préalablement triée. Si cette condition est vérifiée, son coût en temps est nettement inférieur au coût linéaire en $O(n)$ d'une recherche séquentielle.

2.3 Preuve de la correction

Lisez le paragraphe **Preuve de la correction** en deuxième moitié de la p. 289 et première moitié de la p. 290

39) Quel invariant de boucle permet de prouver la correction ?

Réponse :

Pour finir, on peut présenter l'invariant de boucle dans l'algorithme ainsi :

```
    - L'invariant est vrai avant la boucle : c'est le premier commentaire. On u
tilise des accolades.

    - L'invariant est vrai après la boucle : c'est le deuxième commentaire. On
utilise des accolades.

    - L'invariant est vrai à la fin de l'algorithme et la seule possibilité est
d - g = 1 ce qui garantit que ou bien x = liste[g], ou bien x n'est pas dans
la liste. Ceci est présenté en commentaire et entre accolades à la fin de l'a
lgorithme.
```

In [ ]:

```
def dichotomie(x, liste):
    """
    Recherche par dichotomie si x est dans liste.

    Parametres nommes
    -----------------
        x : de type int
        C'est l'element qu'on cherche

        liste : de type list
        C'est la liste dans laquelle se trouve ou non l'element x.

        Retourne
        --------
        L'indice de x si x est un element de liste.
        False si x n'est pas un element de liste.

    """

    g = 0
    d = len(liste)
    assert liste[g] <= x <= liste[d-1]

    # {liste[g] <= x < liste[d]}
    while g < d - 1:
        k = (g + d) // 2
        if x < liste[k]:
            d = k
        else:
            g = k
        # {liste[g] <= x < liste[d]}
    if x == liste[g]:
        return g
    else:
        return False
    # {liste[g] <= x < liste[d] et d - g = 1 => x = liste[g] ou x n'est pa
s dans la liste}
```

Recherche d'une valeur de x0 pour laquelle un phénomène continu s'annule.

In [ ]:
```
#  Importation des bibliotheques permettant les graphiques
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#  Definition de la fonction
def f(x):
    y = x**3 - x**2 + 3*x - 4
    return y

# Choix de la grille, du nom de la figure et des axes
plt.style.use('seaborn-whitegrid')
plt.title(" Fonction continue sur [0 ; 2]")
plt.xlabel("x")
plt.ylabel("y")


# Choix de la fenetre et execution du graphe
x = np.linspace(0, 2, 1000)
plt.plot(x, f(x))
```

- Comment trouver une valeur approchée de la valeur x0 sur l'intervalle [0 ; 2] pour laquelle la fonction f s'annule ?

In [ ]:
```
print("f(0) = ", f(0))

print("f(2) = ", f(2))
```

La fonction f est continue sur l'intervalle [0 ; 2].

f est strictement croissante sur l'intervalle [0 ; 2].

On a f(0) = -4 donc f(0) < 0.

On a f(2) = 6 donc f(2) > 0.

Donc on l'équation $f(x_0) = 0$ a une unique solution x_0 est dans [0 ; 2]

Sur le graphique ci-dessus, on peut estimer que $x_0 = 1,2$ environ.

- Utilisons l'algorithme de dichotomie du haut de la page 290 pour donner une valeur approchée de x_0 avec epsilon (l'écart entre a et b) $= 10^{-10}$.

- Principe :

- On travaille avec une boucle while.

- A chaque tour de boucle, on considère un intervalle [a ; b] dans lequel f change de signe.

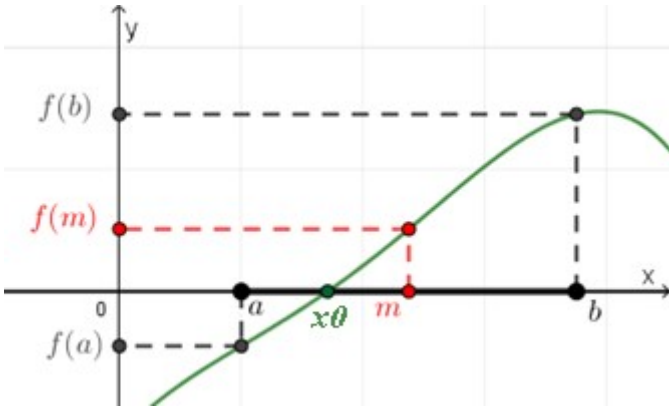
- Au départ, on passe en paramètres a = 0 et b = 2 puisqu'on est certain que f s'annule sur cet intervalle.

- On calcule le milieu m de l'intervalle.

- On examine si f(a) * f(m) est négatif.

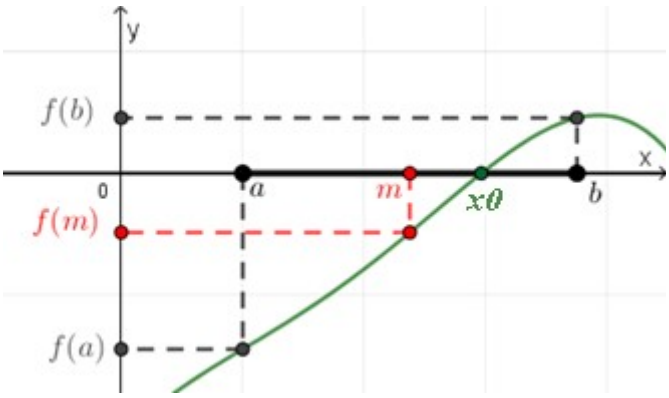
- Si oui (cas n°1), cela signifie que f(m) est positif et donc que x0 se trouve sur [a ; m]. Alors au tour suivant on aura b = m. Le nouvel intervalle [a ; b] sera deux fois plus petit que le précédent intervalle [a ; b].

Cas n°1 :



- Si non (cas n°2), cela signifie que f(m) est négatif aussi et donc que x0 se trouve sur [m ; b]. Alors au tour suivant on aura a = m. Le nouvel intervalle [a ; b] sera deux fois plus petit que le précédent intervalle [a ; b].

Cas n°2 :



- Ainsi à chaque tour, on remplace l'une des bornes de l'intervalle [a ; b] par m le milieu. L'intervalle suivant est donc deux fois plus petit que le précédent.

- On s'arrête dès que l'écart entre a et b est inférieur ou égal à la valeur epsilon (petite) qu'on a choisie au départ.

In [ ]:
```
def dichotomie(f, a, b, epsilon):
    while b - a > ... :
        m = ...
        if f(a) * f(m) ...:
            b = m
        else:
            a = m
    return (a + b) / 2
```

Recherchez dans la cellule de code ci-dessous une valeur approchée de x_0 sur $[0\,;2]$ pour la fonction définie par $f(x) = x^3 - x^2 + 3x - 4$ avec epsilon = 10^{-10}.

In [ ]:
```
x0 = dichotomie(f, 0, 2, 1e-15)
print(x0)
```

Lisez le paragraphe **Note** en deuxième moitié de la p. 290

40) Dans la cellule de code ci-dessous, saisissez le programme donné au bas de la page 290 qui utilise la fonction bisect du module optimize de la bibliothèque scientifique Python scipy

In [ ]:
```
import scipy.optimize

def f(x):
    y = x**3 - x**2 + 3*x - 4
    return y

x0 = scipy.optimize.bisect(f, 1, 2)

print("x0 = ", x0)
```

41) Dans la cellule de code ci-dessous, saisissez le programme donné au bas de la page 290 qui utilise la fonction newton du module optimize de la bibliothèque scientifique Python scipy.

In [ ]:
```
import scipy.optimize

def f(x):
    y = x**3 - x**2 + 3*x - 4
    return y

x0 = scipy.optimize.newton(f, 1)

print("x0 = ", x0)
```

- Cette méthode de dichotomie permet de résoudre des équations pour lesquelles on n'a pas de formule exacte pour les solutions. Par exemple ici, résoudre l'équation du troisième degré $x^3 - x^2 + 3x - 4 = 0$.