Cours 6 : Programmation et complexité

1. Complexité des algorithmes

La complexité en temps d'un algorithme compte le nombre d'opérations élémentaires effectuées par l'algorithme. Cette complexité s'exprime en **fonction de la taille** n des données. On s'intéresse au coût exact si possible, mais également au coût moyen, au meilleur des cas, ou bien au pire des cas. Cette analyse peut se faire de façon formelle ou expérimentale.

Souvent, on s'intéresse à l'ordre de grandeur de la complexité avec la notation $O()$. On dit que la complexité de l'algorithme est $O(f(n))$ où f est généralement linéaire, quasi-linéaire ou polynomiale. La notation $O(f(n))$ signifie que le nombre d'opérations effectuées est borné par $cf(n)$, où c est une constante, quand n tend vers l'infini.

`Python` et `Sympy` permettent de programmer et de faire de l'analyse d'algorithmes (tant formellement qu'expérimentalement) que nous illustrons par quelques algorithmes classiques de L1. Regardons avant tout la librairie qui permet de mesurer un temps de calcul :`timeit`

---

2. La librairie `timeit`

La librairie `timeit` permet de mesurer le temps d'exécution d'un code `Python`. Pour cela, le code doit être répété un grand nombre de fois pour avoir une mesure pertinente. C'est ce que fait la librairie `timeit`: répéter le code pour mesurer la moyenne de son temps d'exécution.

In [15]:
```
import timeit
t=timeit.Timer('sin(1.2)',setup='from math import sin')
t.timeit(10000) #répète sin(1.2) 10000 fois
```

Out[15]: 0.0008780559996921511

Le premier argument au constructeur `Timer` contient le code qui doit être répété. Le second agument contient le code qui sert à l'initialisation. Regardons par exemple le même calcul en changeant le code d'initialisation auquel on fournit toute la librairie `math`:

In [11]:
```
import timeit
t=timeit.Timer('math.sin(1.2)',setup='import math')
t.timeit(10000) #répète sin(1.2) 10000 fois
```

Out[11]: 0.0014206179998836888

On constate que le 2e code est plus lent que le premier soit exprimé en pourcentage: $100(T_2 - T_1)/T_1$

In [12]:
```
((0.0011397519992897287-0.0008595699910074472)/0.0008595699910074472)*100
```

Out[12]: 32.59560143018697

Pour mesurer le temps d'exécution d'une fonction `f` il faut la passer en argument à `Timer` avec ses arguments. Il y a plusieurs méthodes. La plus simple est de faire appel à la méthode `partial` de la librairie de programmation fonctionnelle `functools` qui permet de transformer la fonction `f` et ses paramètres en une nouvelle fonction avec moins de paramètres. Dans l'exemple, on transforme l'addition en une addition d'arité 1 avec la constante 2.

In [16]:
```
from functools import partial

def addition(x,y): return x+y

add2=partial(addition,2)
print(add2(4))
```

```
6
```

Au moyen de `partial`, on peut fournir une fonction à `Timer`. On construit un dictionnaire des mesures du temps d'exécution.

In [14]:
```
releve={}
for i in range(5):
    mesure=timeit.Timer(partial(add2,i))
    t=mesure.timeit(100)
    releve[i]=t
releve
```

Out[14]:
```
{0: 1.8057000033877557e-05,
 1: 1.74489996425109e-05,
 2: 1.733799990688567e-05,
 3: 1.7299999854003545e-05,
 4: 1.7111000033764867e-05}
```

D'autres méthodes permettent de fournir une fonction et ses arguments à `Timer` mais sont plus compliqués dans leur mise en œuvre.

---

3. Recherche du maximum dans une liste

In [19]:
```
def chercheMax(L):
    localMax=L[0]
    for i in range(2,len(L)):
        if L[i]>localMax:
            localMax=L[i]
    return localMax
```

On construit une liste de 9 entiers tirés au hasard :

In [17]:
```
from random import randint
L=[randint(1,111) for i in range(9)]
L
```

Out[17]:
```
[108, 18, 69, 15, 99, 57, 60, 19, 76]
```

In [17]:
```
chercheMax(L)
```

Out[17]:
```
107
```

Pour mesurer le temps de calcul, on fait appel à `timeit`

In [28]:
```
import timeit
```

La librairie `timeit` peut être invoquée directement dans `jupyter` au moyen d'une commande "magique" qui renvoie la moyenne du temps d'exécution ainsi que l'écart-type:

In [19]:
```
%timeit chercheMax(L)
```

```
1.03 µs ± 3.61 ns per loop (mean ± std. dev. of 7 runs, 1000000 lo
ops each)
```

Nous l'utiliserons plutôt pour réaliser des séries de mesures pour des tailles de données croissantes.

On définit une fonction `randlist()` qui retourne une liste de n valeurs aléatoires de l'intervalle $[1, 100[$:

In [21]:
```
def randlist(n):
    return([randint(1,100) for i in range(n)])
```

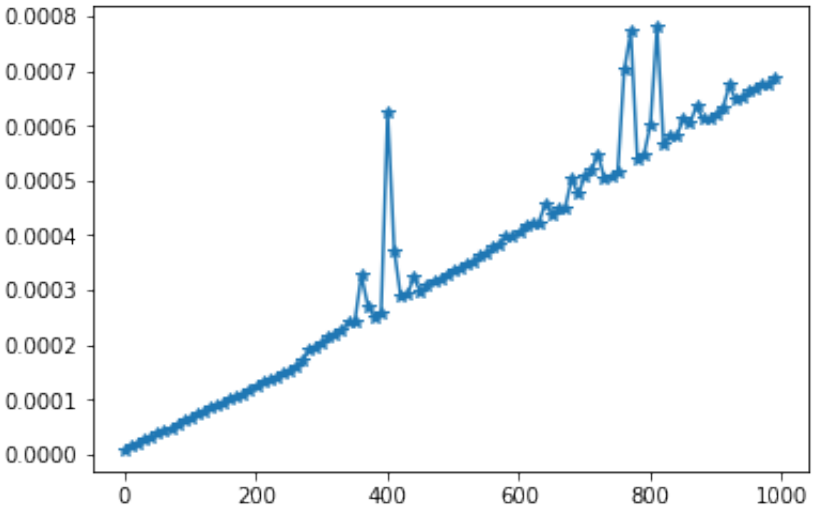
3.1 Etude expérimentale

In [22]:
```
from functools import partial
x,y=[],[]
for i in range(1,1000,10):
    testTimer=timeit.Timer(partial(chercheMax,randlist(i)))
    t=testTimer.timeit(number=10)
    x.append(i)
    y.append(t)
```

On crée 100 listes de valeurs aléatoires de taille croissant de 10 en 10. La liste en `x` conserve cette taille et celle en `y` la mesure du temps moyen de calcul de `chercheMax()` sur une liste de taille `x`. On affiche les points de mesure du temps en fonction de la taille

In [23]:
```
import matplotlib.pyplot as plt
plt.plot(x,y,marker='*')
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x10fa7bcf8>]
```



3.2 Chercher un modèle de la complexité

On intuite une complexité linéaire sur la taille de l'entrée, i.e. un polynôme de degré 1 dont on va chercher les coefficients par la méthode `polyfit` de `numpy` que nous avons déjà utilisée en séance 4:

In [24]:
```
import numpy as np
```

In [51]:
```
p=np.polyfit(x,y,1)
p
```

Out[51]:
```
array([ 6.63243105e-07, -4.17279989e-06])
```

In [52]:
```
poly=np.poly1d(p)
```

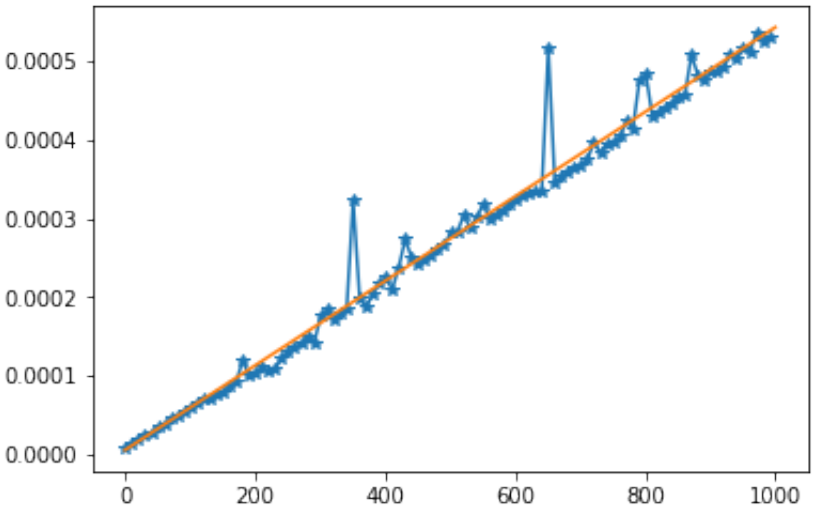
In [53]:
```
print(poly)
```

```
6.632e-07 x - 4.173e-06
```

In [54]:
```
xs=np.linspace(0,1000,100)
ys=poly(xs)
```

In [39]:
```
plt.plot(x,y,marker='*')
plt.plot(xs,ys)
```

Out[39]: [<matplotlib.lines.Line2D at 0x113c56ac8>]



4. Analyse du tri rapide

Vous avez rencontré le tri rapide dans le cours de L1 qui permet de trier efficacement une liste de valeurs. Etudions sa complexité de manière théorique et expérimentalement.

Voici l'algorithme du tri rapide tel que vous l'avez vu en L1:

In [25]:
```
def quicksort(L):
    if L==[] : return []
    else :
        p=L[0]
        L1 = [x for x in L[1:] if x<p]
        L2 = [x for x in L[1:] if x>=p]
        LT1=quicksort(L1)
        LT2=quicksort(L2)
        return LT1 + [p] +LT2
```

4.1 Pour une liste de valeurs uniformément distribuée

4.1.1 Etude théorique

Dans le cas d'une liste dont les valeurs sont uniformément distribuées, la complexité du tri rapide donne lieu à l'équation de récurrence (cf cours de L1):

$$\begin{cases} c_0 = c_1 = 0 \\ c_N = 2c_{n/2} + 3n \end{cases}$$

Essayons de résoudre cette équation de récurrence avec `Sympy` (vous apprendrez à résoudre à la main ce type d'équation en OFI):

In [37]:
```
from sympy import *
init_printing()
c=Function('c')
n=symbols('n',integer=True)
f=c(n)-c(n/2)-3*n
#solve(f,c(n),{c(0):0,c(1):1})
```

Cette tentative échoue. On essaye par un changement de variable. On pose $n = 2^k$ donc $k = \log_2(n)$ et, pour $c(n) = 2.c(n/2) + 3n$ on définit $d(k) = 2.d(k-1) + 3.2^k$ qu'on essaye de résoudre

In [38]:
```
d=Function('d')
k=symbols('k')
g=d(k)-2*d(k-1)-3*2**k
rsolve(g,d(k),{d(0):0})
```

Out[38]: $3 \cdot 2^k k$

Effectuons le changement de variable inverse sur le résultat pour obtenir la complexité du quicksort: $O(n \log(n))$

In [39]:
```
rsolve(g,d(k),{d(0):0}).subs(k,log(n,2))
```

Out[39]: $$\frac{3n \log(n)}{\log(2)}$$

4.1.2 Etude expérimentale

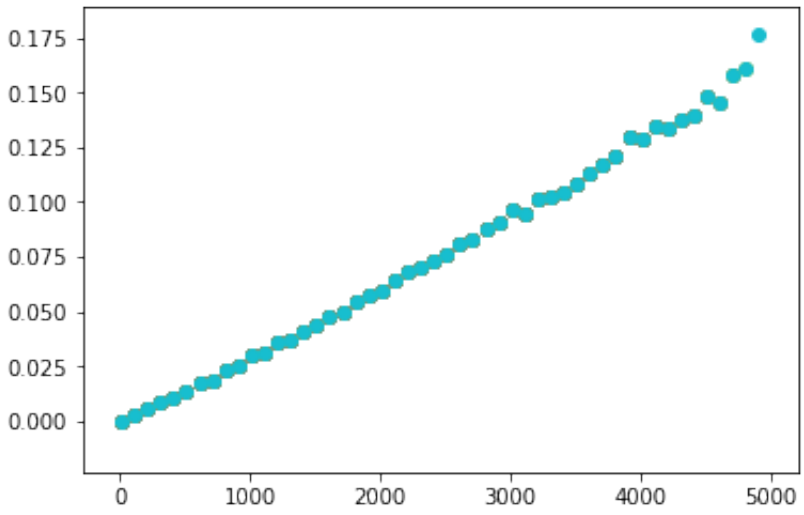
Essayons maintenant de mesurer expérimentalement cette complexité sur une liste de valeurs aléatoires.

In [26]:
```
def listeAlea(n,max):
    return [randint(1,max) for i in range(n)]
```

In [7]:
```
listeAlea(3,100)
```

Out[7]: [52, 64, 52]

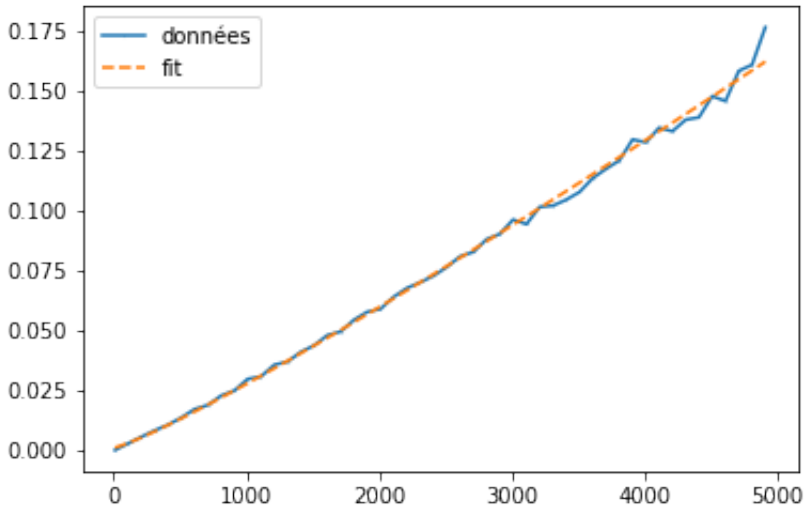
In [37]:
```
x,y=[],[]
for i in range(10,5000,100):
    t=timeit.Timer(partial(quicksort,listeAlea(i,1000)))
    t=t.timeit(10)
    x.append(i)
    y.append(t)
    plt.scatter(x,y)
```



4.1.3 Relier le théorique et l'expérimental

L'analyse théorique de la complexité nous donne une complexité en $O(n\log(n))$. Essayons de l'approcher avec `polyfit`. Pour cela, on considère les valeurs de $n\log(n)$ comme un polynôme de degré 1 dont les valeurs en x sont données par le calcul de $x\log(x)$:

In [38]:
```
import numpy as np
coefs=np.polyfit(x*np.log(x),y,1)
fit=np.poly1d(coefs)
plt.plot(x,y,marker='.',markersize=0.2,label="données")
plt.plot(x,fit(x*np.log(x)),'--',label='fit')
plt.legend()
plt.show()
```



Et nous obtenons la valeur de la constante pour obtenir un modèle exact de la complexité du tri rapide

In [38]:
```
print(fit)
```

```
7.191e-06 x + 0.001545
```

Ici encore, en remplaçant le `x` de la solution par $x\log(x)$, on obtient les coefficients: `7.191e-06 x log(x) + 0.00154`

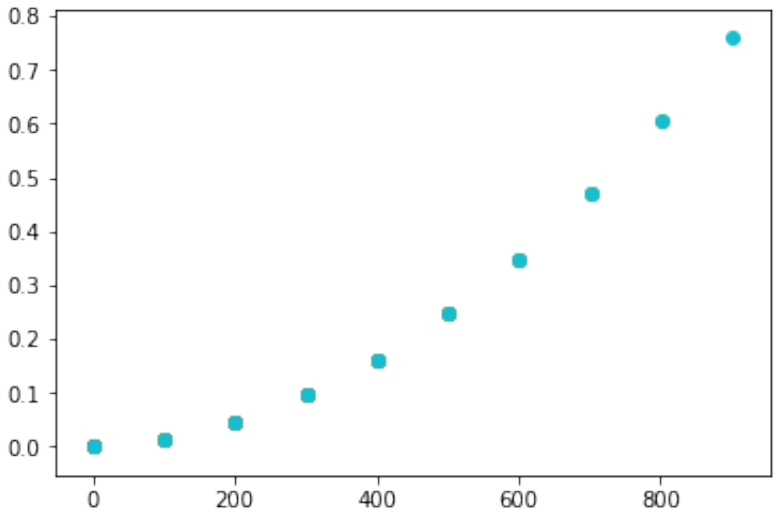
4.2 Pour une liste de valeurs qui est déjà triée

Lorsque les valeurs de la liste sont déja triées, le tri rapide fonctionne moins bien; il est en temps quadratique , c'est à dire en $O(n^2)$. C'est le pire des cas.

4.2.1 Etude expérimentale

Essayons de mesurer expérimentalement la complexité du tri rapide dans le pire des cas.

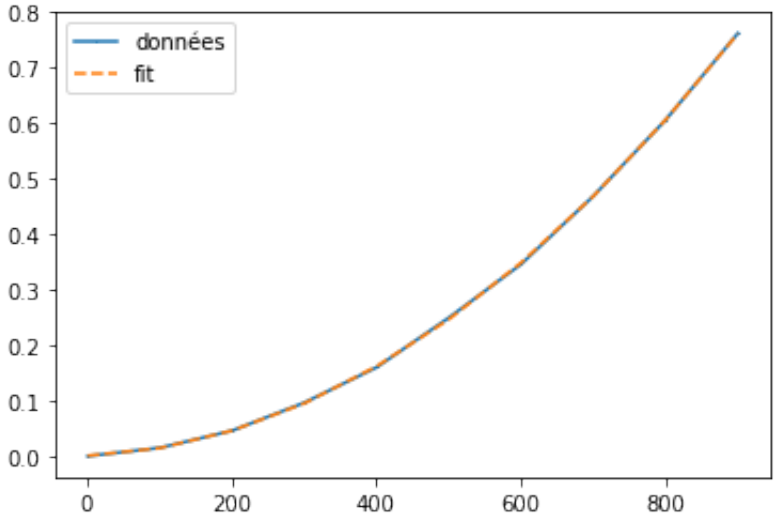
In [40]:
```
x,y=[],[]
for i in range(1,1000,100):
    t=timeit.Timer(partial(quicksort,range(i)))
    t=t.timeit(20)
    x.append(i)
    y.append(t)
    plt.scatter(x,y)
```



4.2.2 Relier le théorique et l'expérimental

Essayons de trouver les coefficients du polynôme de degré 2 qui approche les points mesurés pour le tri rapide sur une liste triée:

In [42]:
```
coefs=np.polyfit(x,y,2)
fit=np.poly1d(coefs)
plt.plot(x,y,marker='*',markersize=0.2,label="données")
plt.plot(x,fit(x),'--',label='fit')
plt.legend()
plt.show()
```



In [43]:
```
print(fit)
```

```
            2
8.841e-07 x + 4.762e-05 x + 0.0006804
```

5. Analyse de la recherche dichotomique

Comme on sait trier une liste de valeurs, on cherche à décider si un élément est dans une liste **triée**. Quand on dispose d'un grand nombre d'enregistrements ordonnés on utilise une recherche dichotomique qui procède d'une stratégie diviser pour régner

- la liste est coupée en deux parties (gauche et droite)
- on cherche dans quelle partie est l'élément
- on répète lance récursivement la recherche dans la partie concernée La fonction retourne la position de l'élément dans la liste Le premier programme présente la recherche dichotomique récursive et le second la recherche dichotomique itérative. Si l'élément n'est pas dans le tableau, la valeur -1 est retournée.

In [16]:
```
def dichotomie (T, min, max, e):
    from math import floor
    if max >= min: # le cas d'arrêt, on a croisé les indices
        milieu = min + floor((max - min)/2)
        if T[milieu] == e: # l'élément est au milieu
            return milieu
        elif T[milieu] > e: #l'élément est dans la martie gauche
            return dichotomie(T, min, milieu-1, e)
        else: # l'élément est dans la partie droite
            return dichotomie(T, milieu+1, max, e)
    else: # l'élément n'est pas dans T
        return -1
```

In [17]:
```
dichotomie(range(10),0,len(range(10))-1,5)
```

Out[17]: 5

Voici la version itérative de la recherche dichotomique de l'élément e dans la liste T

In [1]:
```
def dichoIter(T,e):
    from math import floor
    min,max = 0, len(T)-1
    while min<max:
        milieu = floor((min+max)/2)
        if T[milieu]<e:
            min=milieu+1
        else:
            max=milieu
    if T[min]==e: return min
    else: return -1
```

In [9]:
```
dichoIter(range(1,10,2),3)
```

Out[9]: 1

5.1 Etude théorique

Quelle est la complexité de la recherche dichotomique? Notons $d(n)$ la complexité de la recherche dichotomique d'une liste à n éléments. Elle donne lieu à la récurrence:

$$\begin{cases} d(1) = 0 \\ d(n) = d(n/2) + 1 \end{cases}$$

On pose $n = 2^k$ donc $k = \log_2(n)$ et, pour $d(n) = d(n/2) + 1$ on définit $e(k) = e(k-1) + 1$ qu'on essaye de résoudre

In [23]:
```
from sympy import *
init_printing()
e=Function('e')
k=symbols('k')
n=symbols('n',integer=True)
g=e(k)-e(k-1)-1
rsolve(g,e(k),{e(0):0})
```

Out[23]: k

Effectuons le changement de variable inverse sur le résultat pour obtenir la complexité de la recherche dichotomique: $O(\log(n))$

In [24]:
```
rsolve(g,e(k),{e(0):0}).subs(k,log(n,2))
```

Out[24]: $\frac{\log(n)}{\log(2)}$

On obtient une complexité logarithmique pour la recherche dichotomique d'un élément. Celle-ci est maximale lorsqu'on recherche un élément qui n'est pas dans la liste. On le vérifiera expérimentalement en TP.

5.2 Application de la recherche dichotomique

Théorème des valeurs intermédiaires : Pour toute application continue $f : [a, b] \to \mathbb{R}$ et tout réel u compris entre $f(a)$ et $f(b)$, il existe au moins un réel c compris entre a et b tel que $f(c) = u$.

Si on sait évaluer le signe de $f((a + b)/2)$ et pour f strictement croissante sur $[a, b]$ avec $f(a) < 0 < f(b)$, la recherche dichotomique nous permet de trouver y tq $f(y) = 0$

- on commence par le couple (a, b)
- on évalue $v = f((a + b)/2)$
- si $v < 0$ on remplace a par v sinon on remplace b par v
- on itère sur ce nouveau couple jusqu'à ce que la différence entre les valeurs soit inférieure à une précision donnée

6. Une erreur facile à commettre

Dans le TP, vous allez mesurer la complexité de différents tris pour les comparer. Certains d'entre eux travaillent sur place (il modifient la liste fournie en paramètre) d'autres non. Lorsque les tris travaillent sur place, comme la liste est modifiée, c'est celle-ci (donc triée) qui est réutilisée.

Illustration sur le tri de `Python` implémenté par la méthode `sort()`:

In [45]:
```
def sort(L):
    print(L)
    L.sort()

L=listeAlea(10,10)
t=timeit.Timer(partial(sort,L[:]))
t.timeit(5)
```

```
[8, 8, 3, 6, 4, 1, 1, 4, 1, 10]
[1, 1, 1, 3, 4, 4, 6, 8, 8, 10]
[1, 1, 1, 3, 4, 4, 6, 8, 8, 10]
[1, 1, 1, 3, 4, 4, 6, 8, 8, 10]
[1, 1, 1, 3, 4, 4, 6, 8, 8, 10]
```

Out[45]: `0.0002944839998235693`

La liste `L`, même si elle est copiée (`L[:]`) dans `partial` n'est évaluée convenablement qu'une seule fois (la première).

La complexité peut changer (pensez au quicksort sur une liste triée). Il faut changer la manière d'appeler la fonction et qu'elle travaille sur la **même** liste pour mesurer la complexité:

In [46]:
```
def timeSort(algo,L):
    algo(L[:])

L=listeAlea(10,10)
t=timeit.Timer(partial(timeSort,sort,L))
t.timeit(5)
```

```
[5, 7, 10, 3, 4, 2, 4, 7, 1, 4]
[5, 7, 10, 3, 4, 2, 4, 7, 1, 4]
[5, 7, 10, 3, 4, 2, 4, 7, 1, 4]
[5, 7, 10, 3, 4, 2, 4, 7, 1, 4]
[5, 7, 10, 3, 4, 2, 4, 7, 1, 4]
```

Out[46]: `0.0002092849999826285`