TP N°2 : CORRECTION -TERMINAISON ET COMPLEXITE TEMPORELLE D'UN ALGORITHME

PARTIE 1 : MULTIPLICATION EGYPTIENNE

Considérons le programme suivant, qui implémente un ancien algorithme égyptien.

```
def egyptienne(a:int, b:int) ->int :
      t=0
      while a>0 :
            if a%2 == 1 :
                  t = t + b
            b = 2*b
            a = a //2
      return t
```

Q1. Détailler l'exécution de `egyptienne(41,3)`

	a	b	t
Initialisation	41	3	0
	20	6	3
	10	12	3
	5	24	3
	2	48	27
	1	96	27
	0	192	123
Fin de la boucle while			**123**

Q2. Montrer que la fonction `egyptienne` termine toujours.

On choisit a en variant de boucle

Initialisation : a est un entier strictement positif

On note a_k le contenu de la variable a à la fin de la $k^{ième}$ itération du bloc d'instructions.

Supposons $a_k > 0$, alors si a_k est pair $a_{k+1} = \frac{a}{2} < a_k$. si a_k est impair $a_{k+1} = \frac{a-1}{2} < a_k$

La suite (a_k) est positive et décroissante donc l'algorithme termine.

Q3. Montrer à l'aide d'un invariant de boucle que la fonction renvoie le produit entre les deux arguments.

On choisit comme invariant de boucle la propriété : « $I_k = a_k \times b_k + t_k$ est constante. »

On note x_k la valeur stockée dans la variable x à la fin de la $k^{ième}$ itération.

Initialisation : $I_0 = a_0 \times b_0$

Hérédité : On suppose que $I_k = a_k \times b_k + t_k$ vrai.

Procédons par disjonction de cas :

- 1er cas : a_k%2 == 1 est vrai (a est impair) :

alors $a_{k+1} = \frac{a_k - 1}{2}$ $b_{k+1} = 2 \times b_k$ $t_{k+1} = t_k + b_k$

D'où $I_k = (2a_{k+1} + 1) \times \frac{b_{k+1}}{2} + t_{k+1} - b_k = a_{k+1} \times b_{k+1} + \frac{b_{k+1}}{2} + t_{k+1} - b_k = I_{k+1}$

- $2^{ème}\ cas :\ a_k\%2 == 1\ est\ faux$ (a est pair) :

Alors $a_{k+1} = \frac{a_k}{2}$ $b_{k+1} = 2 \times b_k$ $t_{k+1} = t_k$

D'où $I_k = 2a_{k+1} \times \frac{b_{k+1}}{2} + t_{k+1} = a_{k+1} \times b_{k+1} + t_{k+1} = I_{k+1}$

Vérification : à la fin de la boucle $a_n = 0$ donc $I_n = t_n = a_0 \times b_0$.

L'algorithme est donc correct.

L'algorithme termine et est correct, nous avons donc montré la correction totale de cet algorithme.

PARTIE 2 : MESURE DE LA COMPLEXITE EN TEMPS D'UN ALGORITHME

La complexité en temps $\mathcal{C}$(n) d'un algorithme permet de déterminer la durée d'exécution du programme constituant une implémentation de cet algorithme en fonction de la taille d'instance n associée. On peut, de manière parfaitement équivalente, considérer la complexité en termes de nombre d'opérations élémentaires d'un algorithme, ces deux grandeurs étant équivalentes en notation de Landau.

Q1. Justifier qu'il soit équivalent, en notation de Landau, de considérer les complexités en temps et en nombre d'opérations élémentaires.

Un processeur réalise une opération élémentaire pendant une durée fixée par sa fréquence d'horloge (nombre d'opérations élémentaires pouvant être effectuées par seconde). Le temps d'exécution d'un programme est donc, grossièrement, égal au nombre d'opérations élémentaires de ce programme multiplié par le temps de calcul pour une opération élémentaire. En notation de Landau, les complexités en temps et en nombre d'opérations élémentaires sont donc équivalentes.

Afin d'étudier la complexité en temps $\mathcal{C}$ d'un algorithme, on mesure la durée d'exécution Δt d'une fonction donnée en faisant varier la taille d'instance n.

Q2. Expliquer pourquoi, pour un algorithme de complexité en temps $\mathcal{C}$ polynomiale, il peut être particulièrement intéressant de représenter la fonction log(Δt) = f (log(n)), avec Δt le temps d'exécution de l'algorithme et n la taille d'instance. On supposera que n est « suffisamment grand » pour procéder à quelques simplifications que l'on explicitera.

Si un algorithme a une complexité polynomiale alors elle s'écrit :

$$C(n) = \sum_{i=0}^{D} c_i \times n^i$$

Lorsque $n \to +\infty,\ C(n) \approx c_D \times n^D$ avec D le degré du polynôme.

En linéarisant avec le logarithme, on obtient :

$$\log\big(C(n)\big) = \log(c_D) + D \times \log(n)$$

Sachant que la durée d'exécution de l'algorithme Δt est proportionnelle à la complexité de l'algorithme :

$$\log(\Delta t) = Cste + D \times \log(n)$$

$\log(\Delta t)$ est donc une fonction affine de $\log(n)$ avec un coefficient directeur égal au degré D du polynôme correspondant à la complexité de l'algorithme.

Afin de s'appuyer sur des exemples, on travaille avec les fonctions `foo` et `bar` suivantes :

```
def foo(n: int) -> int:
    s = 0
    for k in range(n):
        s = s + k**2
    return s
```

```
def bar(n:int)-> int :
    a=0
    for i in range (n):
        for j in range (n):
            a=a+i*j
    return(a)
```

Q3. Que fait la fonction `foo` ? Déterminer sa complexité en temps C_{foo} en notation de Landau.

La fonction foo renvoie la somme des carrés des entiers de 1 à n-1.
Elle comporte une seule boucle for, sa complexité est donc en linéaire. $C_{foo}(n) = \mathcal{O}(n)$

Q4. Que fait la fonction `bar` ? Déterminer sa complexité en temps C_{bar} en notation de Landau.

La fonction bar permet de calculer la somme des produits $i \times j$ pour toutes les paires $(i,j) \in [\![0, n-1]\!]^2$:

$$\begin{aligned} bar(n) &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} i \times j \\ &= \left(\sum_{i=0}^{n} i\right) \times \left(\sum_{i=0}^{n} j\right) \\ &= \left(\frac{n \times (n-1)}{2}\right)^2 \end{aligned}$$

La fonction bar contient 2 boucles for imbriquées dont le nombre d'itérations est égal à la taille d'instance n pour chacune d'entre elles soit :

$$C_{bar}(n) = \mathcal{O}(n^2)$$

Q5. Écrire une fonction `chrono` prenant comme arguments une fonction `g` et un entier n (représentant la taille d'instance du problème étudié), et retournant la durée d'exécution de la fonction g pour la taille d'instance donnée. On utilisera la méthode `time` de la bibliothèque `time`.

```
def chrono (g:callable, n:int)-> float :
    """
    Renvoie la durée d'exécution de la fonction g pour une instance n donnée.
    cette fonction utilise la méthode time de la bibliothèque time en secondes

    =======

    jeu de test :
        >>> chrono(bar,10000)
        6.20145583152771

        >>> chrono(foo,10000)
```

```
        0.0014514923095703125
    """
    heure=time.time()
    g(n)
    return(time.time()-heure)
```

Q6. Saisir les commandes suivantes dans le shell :

```
>>> chrono(foo, 100000)
>>> chrono(foo, 1000000)
>>> chrono(foo, 10000000)
```

Ces résultats sont-ils en accord avec la complexité en temps de la fonction `foo` déterminée précédemment ?

```
In [7]: chrono(foo,100000)
Out[7]: 0.015599966049194336

In [8]: chrono(foo,1000000)
Out[8]: 0.1718142032623291

In [9]: chrono(foo,10000000)
Out[9]: 1.8594765663146973
```

Remarque : les valeurs peuvent différer d'un ordinateur à un autre.

On constate que les durées d'exécution sont multipliées par 10 (environ) lorsque la taille d'instance est multipliée par 10. Ce résultat est compatible avec la complexité en temps linéaire déterminée précédemment pour la fonction foo.

Q7. Saisir les commandes suivantes dans le shell :

```
>>> chrono(bar, 100)
>>> chrono(bar, 1000)
>>> chrono(bar, 10000)
```

Ces résultats sont-ils en accord avec la complexité en temps de la fonction bar déterminée précédemment ?

```
In [12]: chrono(bar,100)
Out[12]: 0.000852518613515159

In [13]: chrono(bar,1000)
Out[13]: 0.09363818168640137

In [14]: chrono(bar,10000)
Out[14]: 10.312515497207642
```

On constate que les durées d'exécution sont multipliées par 100 (environ) lorsque la taille d'instance est multipliée par 10. Ce résultat est compatible avec la complexité en temps quadratique déterminée précédemment pour la fonction bar.

Q8. Écrire une fonction `complexite` prenant comme arguments une fonction `g` et une liste d'entiers L (représentant plusieurs tailles d'instance n_i), et retournant les deux listes suivantes :

- une liste constituée des valeurs $log(n_i)$;
- une liste constituée des valeurs $log\,(\Delta t(g(n_i)))$ avec Δ t(g(n_i)) le temps d'exécution de la fonction g pour laquelle la taille d'instance choisie est n_i.

On utilisera la fonction `chrono`.

```
def complexite (g:callable,L:list)-> tuple :
"""
Cette fonction prend en argument une fonction g dont on souhaite connaitre la complexité
et une liste d'entiers L correspondant à une liste de taille d'instance (n)

renvoie une liste comportant ln(L[ni]) et une autre comportant ln(temps d'excécution) avec le temps d'exécution étant celui de la fonction g pour une instantce ni

cette fonction utilise la fonction chrono définie ci-dessus
==============
    Test
==============
>>> complexite(foo,[100000,1000000,10000000])
([11.512925464970229,      13.815510557964274,      16.11809565095832],      [-4.041846311331792, -1.832126545381148, 0.5063514802211864])

"""
temps_execution=[]
ln_n=[]
for n in L:
     ln_n = ln_n + [math.log10(n)]
     temps_execution = temps_execution + [math.log10(chrono(g,n))]
return(ln_n,temps_execution)
```

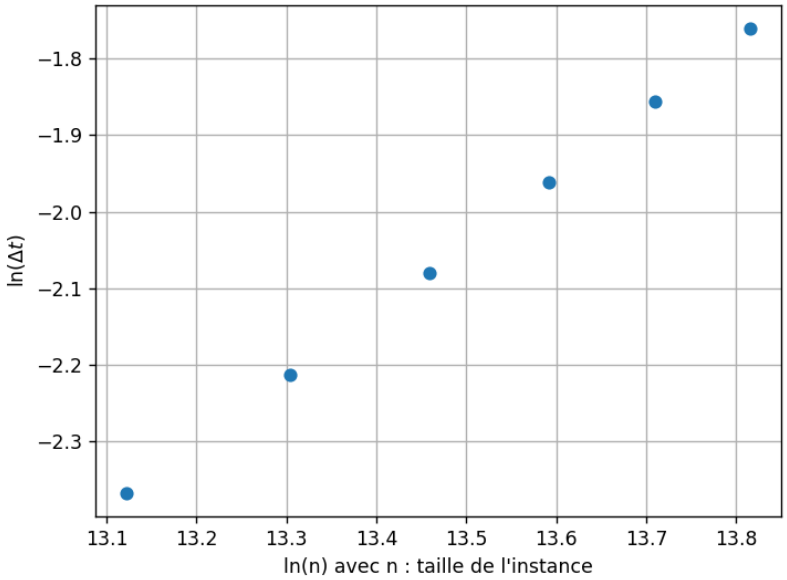
Q9. Proposer une suite d'instructions permettant de tracer la courbe $ln\,(\Delta t(g(n_i))) = f(log(n_i))$. On utilisera la bibliothèque `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt

ln_n,temps_exec=complexite(g,L)
plt.plot(ln_n,temps_exec,'o:')
plt.grid()
plt.xlabel("log(n) avec n : taille de l'instance")
plt.ylabel(r"$\log(\Delta t)$")
plt.show()
```

Q10. Appliquer ce protocole aux cas suivants :

a. étude de la complexité de la fonction `foo` – on prendra les valeurs suivantes pour n : $n \in \{5\cdot10^5, 6\cdot10^5, 7\cdot10^5, 8\cdot10^5, 1\cdot10^6\}$

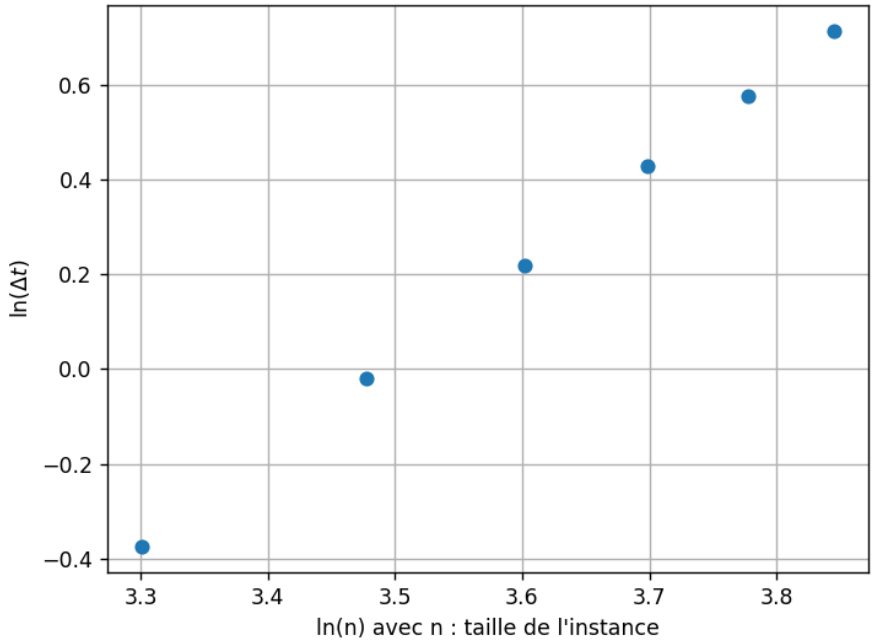
```
L=[int(5e5),int(6e5),int(7e5),int(8e5),int(9e5),int(1e6)]
g=foo
```



On obtient une droite, ce qui confirme l'hypothèse de complexité polynomiale. La pente de la droite veut environ 1 ce qui confirme l'hypothèse d'une complexité en temps linéaire.

b. étude de la complexité de la fonction `bar` – on prendra les valeurs suivantes pour n : $n \in \{500, 600, 700, 800, 900, 1000\}$

```
L=[2000, 3000, 4000, 5000, 6000,7000]
g= bar
```



On obtient une droite, ce qui confirme l'hypothèse de complexité polynomiale. La pente de la droite veut environ 2 ce qui confirme l'hypothèse d'une complexité en temps quadratique.

Q11. Déterminer la pente des droites obtenues pour les fonctions foo et bar. On pourra utiliser la méthode `polyfit` de la bibliothèque `numpy` ou la méthode `curve_fit` de la bibliothèque `scipy.optimize`. Conclusion ?

```
#regression linéaire
```

```
A,B=np.polyfit(log_n,temps_exec,1)
print(A,B)
```

On obtient pour foo :
A = 0.962, soit environ 1, ce qui confirme la complexité linéaire de la fonction foo.

Ce qui confirme l
Et pour bar : A = 1.962 soit environ 2 ce qui confirme la complexité quadratique de la fonction.

PARTIE 3 : APPLICATION AU PROBLEME DU VOYAGEUR DE COMMERCE

Le problème du voyageur de commerce peut s'énoncer de la manière suivante :

Un voyageur de commerce doit – au cours de sa tournée – visiter les n villes de la zone géographique qu'il souhaite couvrir. Connaissant les coordonnées $(x_i, y_i)_{i\in[\![1,n]\!]}$ des n villes visitées par le voyageur de commerce, quel est le trajet le plus court permettant de visiter une et une seule fois chaque ville ?

Ce problème « anodin en apparence » est en réalité un problème majeur en Informatique, et permet d'illustrer l'une des questions centrales des Mathématiques : « tout problème possédant une solution peut-il être résolu par un algorithme de complexité en temps polynomiale ? ».
Cette question est, à ce jour, sans réponse – elle constitue l'un des « problèmes du prix du millénaire » (chacun dotés d'un prix d'un million de dollars US). La résolution de ce problème aurait des répercussions majeures, notamment en cryptographie.
Afin d'étudier ce problème, on suppose que chaque ville i se situe au point de coordonnées (x_i, y_i) avec $x_i \in [0,1000]$ u.a. et $y_i \in [0,1000]$ u.a. Chaque ville est donc positionnée au hasard dans un carré de côté 1000 u.a. Dans la suite de l'exercice, chaque ville est représentée par un tuple comprenant :

- un entier représentant le numéro i de la ville ;
- un flottant représentant l'abscisse x_i de la ville ;
- un flottant représentant l'ordonnée y_i de la ville.

Q12. Écrire une fonction `posville` ne prenant pas d'argument, et retournant un tuple de deux flottants représentant les coordonnées (x_i, y_i) de la ville i. Les coordonnées x_i et y_i sont chacune tirées aléatoirement de manière uniforme sur l'intervalle [0, 1000]. On utilisera la bibliothèque `random`.

```
def posville()-> tuple :
"""
entrée : None
sortie : un tuple (x,y) correspondant aux coordonnées de la ville i

cette fonction utilise la bibliothèque random

=====
jeux de tests
=====
>>> posville()
(245.74907427849269, 17.48935628127779)
"""
return(1000*rd.random(),1000*rd.random())
```

Remarque : autre possibilité : utilisation de `rd.uniform(1,1000)`

Q13. Écrire une fonction `carte` prenant comme argument un entier n, et retournant une liste de n villes numérotées de 1 à n selon le format décrit précédemment (chaque ville est représentée par un tuple comportant 3 éléments). On utilisera la fonction `posville`.

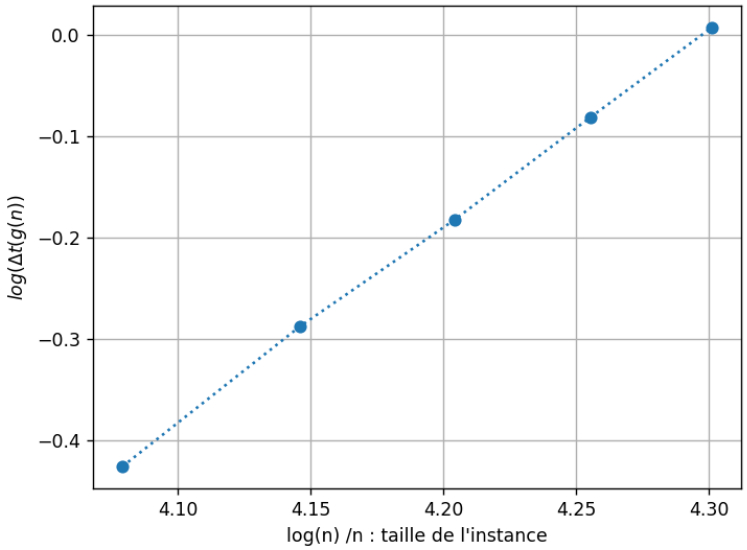
```
def carte (n):
"""
entrée : un entier n correspondant aux nombres de villes sur la carte, elles sont numérotées de 1 à n
sortie : une liste de tuple de format (i,x,y) avec i le numéro de la ville, x,y ses coordonnées.
======
jeux de tests :

>>> carte(4)
[(1,    680.1432290203232,    705.8501385044273),    (2,    926.9498111489498,
848.7696405804669),    (3,    545.0669280861879,    513.1954029171278),    (4,
397.94100567522315, 734.6903360624124)]
"""
carto=[]
for k in range (1,n+1): # n-1 itérations
     coord=posville()
     carto.append((k,coord[0],coord[1]))
return(carto)
```

Q14. Quelle est la complexité en temps de la fonction carte ? Vérifier ce résultat à l'aide des outils développés dans la première partie. Commenter le résultat obtenu et proposer une interprétation.

La fonction carte contient une boucle for avec n-1 itérations, la complexité est donc linéaire. $C_{carte}(n) = \mathcal{O}(n)$.

En vérifiant avec la fonction complexité, on obtient la droite ci-contre.

```
L=[1000000,1500000,2000000,2500000,3000000]
ln_n,temps_exec=complexite(carte,L)
plt.plot(ln_n,temps_exec,'o:')
plt.grid()
plt.xlabel("log(n) /n : taille de l'instance")
plt.ylabel(r"$log(\Delta t(g(n))$")
plt.show()
```



Avec une pente d'environ 1 ce qui correspond à la complexité linéaire comme déterminé précédemment.

Remarque : si dans la fonction vous avez utilisé `carto = carto + [(k,coord[0],coord[1])]` *à la place de la méthode append, vous devez obtenir une pente de l'ordre de 1,8-2,0. Ceci est du au fait qu'en python, la concaténation de liste n'a pas une complexité en* $\mathcal{O}(1)$ *en python.*

Q15. Écrire une fonction `trajets` prenant comme argument une liste C (représentant la liste des villes visitées par le voyageur de commerce), et retournant tous les trajets possibles passant une seule fois par chaque ville sous la forme d'une liste de tuples (chacun de ces tuples représentant une ville visitée). On pourra utiliser la méthode `permutations` de la bibliothèque `itertools` afin de générer toutes les permutations des éléments d'une liste :

```
import itertools

def trajet (C:list):
    """
    entrée : une liste de n tuples (i,x,y) avec i entier le numéro d'une ville,
    et x et y ses coordonnées, des flottants
    sortie : une liste de n tuples représentant la liste des trajets possibles
    entre les villes
    en ayant été une seule fois maximum par ville

    ====
    jeux de test
    ====
    >>> trajet([(1,10,10),(2,20,20),(3,30,30)])

    [[(1, 10, 10), (2, 20, 20), (3, 30, 30)], [(1, 10, 10), (3, 30, 30), (2,
    20, 20)], [(2, 20, 20), (1, 10, 10), (3, 30, 30)], [(2, 20, 20), (3, 30,
    30), (1, 10, 10)], [(3, 30, 30), (1, 10, 10), (2, 20, 20)], [(3, 30, 30),
    (2, 20, 20), (1, 10, 10)]]
    """

    n = len(C)
    #génération de la liste des numéros des villes (on numérote entre 0 et n-
    1, pas entre 1 et n (numérotation réelle des villes avec 'carte') car
    utilisation dans range
    c=[]
    for i in range (n):
        c=c+[i]

    #génération des permutations
    P=itertools.permutations(c)

    #construction de la liste (il faut convertir la liste de tuples en liste
    de listes)

    T = []
    for t in P:
        trajet = []
        for k in range(n) :
            trajet = trajet +[C[t[k]]]

        T = T + [trajet]

    return(T)
```

Pour les plus rapides : on peut également essayer de coder soi-même un algorithme permettant de générer toutes les permutations des éléments d'une liste (une fois le TP terminé)

Q16. Écrire une fonction `distance` prenant comme arguments deux tuples (représentant les villes i et j), et retournant la distance (à vol d'oiseau) entre les villes i et j.

```
def distance (V:tuple,W:tuple)-> float:
    """
    renvoie la distance entre deux villes

    entrée : deux tuples V et W de forme (i,xi,yi) avec i le numéro de la ville,
xi et yi ses coordonnées sur la carte

    sortie : distance -> float : distance entre les villes V et W

    ===
    jeux de tests
    ===
    >>> distance((1,5,5),(2,6,6))
    1.4142135623730951

    """
    return(math.sqrt((V[1]-W[1])**2+(V[2]-W[2])**2))
```

Q17. Écrire une fonction `voycom` prenant comme argument une liste C (représentant la liste des villes visitées par le voyageur de commerce), et retournant une liste représentant le trajet effectué par le voyageur de commerce entre ces villes permettant de minimiser la distance totale parcourue. On utilisera les fonctions `trajets` et `distance`.

```
def voycom(C:[[int,float,float]])->[[int,float,float]]:
     """
     Résolution du problème du voyageur de commerce pour une liste de villes
données

     entrée :
     liste de ville de la forme [(i,xi,yi)] avec i le numéro de la ville et xi
et yi les coordonnées [(int,float,float)]

     sortie :
     liste de type [[int,float,float]] correspondant au trajet optimal pour le
commerçant

     ===
     jeux de tests
     ===
     >>> voycom([[(1,10,10),(2,20,20),(3,30,30)]])
     [(1, 10, 10), (2, 20, 20), (3, 30, 30)]

     >>> voycom([[1, 0, 0], [2, 1, 0], [3, 0, 2], [4, 1, 3]])
     [[2, 1, 0], [1, 0, 0], [3, 0, 2], [4, 1, 3]]

     """
     D_min=float('inf')
     trajet_opt=[]
```

```
    T = trajet(C)
    for t in T :
        D=0
        for k in range (1,len(C)):
            D = D + distance(t[k],t[k-1])
        if D<D_min :
            D_min=D
            trajet_opt = t
    return(trajet_opt)
```

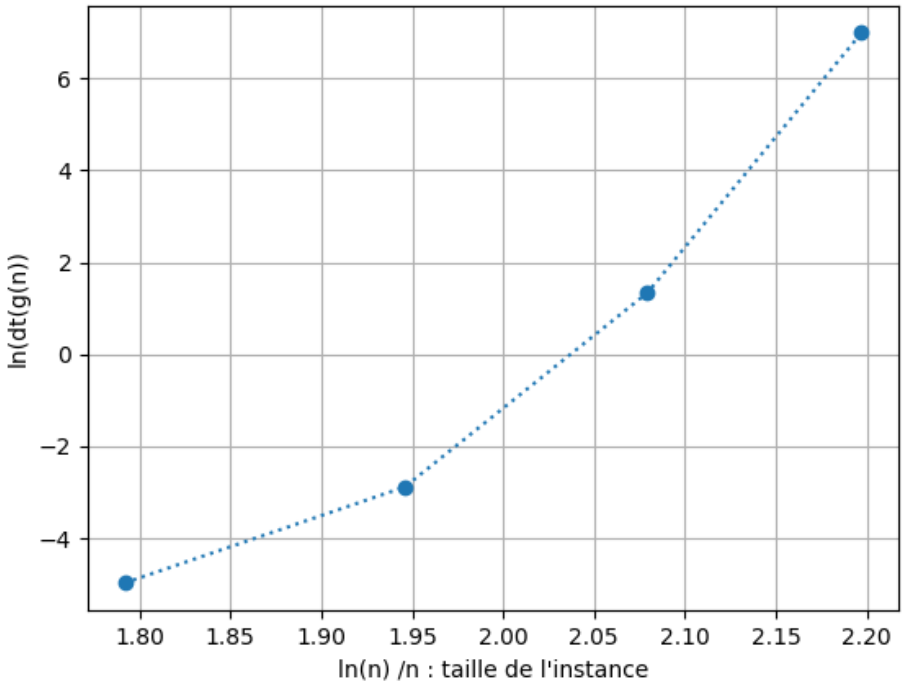
Q18. Ecrire une fonction `voycomalea` prenant comme argument un entier n, et retournant une liste représentant le trajet effectué par le voyageur de commerce entre n villes positionnées aléatoirement dans un carré de côté 1000 u.a. et permettant de minimiser la distance totale parcourue. On utilisera les fonctions `carte` et `voycom`.

```
def voycomalea(n:int) ->[[int,float,float]]:
    """
    Résolution du problème du voyageur de commerce pour une liste de n villes
comprises dans un rectangle (0<x<1000)*(0<y<1000)

    entrée :
    n : nombre de villes à positionner (int)

    sortie :
    liste de type [[int,float,float]] correspondant au trajet optimal pour le
commerçant

    ===
    jeux de tests
    ===
    >>> voycomalea(4)
    [(3,    81.8703674864415,    962.3719803911147),    (1,    240.6057248351131,
570.7578754059323),    (2,    525.1798464493764,    754.113281437924),    (4,
592.349428954073, 117.90929006641915)]

    """
    A=carte(n)
    return(voycom(A))
```

Q19. Quelle est la complexité en temps de la fonction `voycomalea` ? Vérifier ce résultat à l'aide des outils développés dans la première partie. On rappelle la formule de Stirling :

$$\text{Lorsque n} \to \infty, \quad \ln(n!) \sim n \times \ln(n) - n$$

Attention, ne testez pas avec plus de 9 villes, le temps de calcul devient vite conséquent.



On observe que dans ce cas-là, on n'obtient pas une droite, la complexité n'est pas polynomiale.

Il est possible d'expliquer ceci en analysant la fonction trajet :
Il y a n choix pour la ville de départ, puis en partant de la première ville, il y a n-1 possibilités pour la suivante, puis n-2 etc … la complexité est donc factorielle.

En réalisant le changement de variable $y = ln(\Delta t)\ et\ x = ln(n)$, on constate que la formule de Stirling conduit à une courbe du type :

$$y = e^x \times (x-1) \sim e^x \times x$$

Ce qui correspond approximativement à une courbe représentative de la fonction exponentielle. Le graphique obtenu semble compatible avec ce résultat théorique. (on pourrait modéliser la courbe pour vérifier).