Exercice 1

On considère la fonction mystère suivante, qui étant donnés deux entiers `x` et `y`, renvoie un autre entier :

```
def mystere(x :int, y :int) -> int :
    a=0
    while y>0 :
        if y%2==1 :
            a=a+x
        x=x+x
        y=y//2
    return(a)
```

1. Recopier et compléter les tableaux suivants donnant l'évolution des variables `x`,`y` et `a` ainsi que de la quantité `a+x*y` lors des appels `f(7,20)` et `f(3,85)` :

Pour **`mystere(7,20)`**	x	y	a	$a+x*y$
Fin du premier tour de boucle	14	10	0	140
Fin du deuxième tour de boucle	28	5	0	140
Fin du troisième tour de boucle	56	2	28	140
Fin du quatrième tour de boucle	112	1	28	140
Fin du cinquième tour de boucle	224	0	140	140

Pour **`mystere(3,85)`**	x	y	a	$a+x*y$
Fin du premier tour de boucle	6	42	3	255
Fin du deuxième tour de boucle	12	21	3	255
Fin du troisième tour de boucle	24	10	15	255
Fin du quatrième tour de boucle	48	5	15	255
Fin du cinquième tour de boucle	96	2	63	255
Fin du sixième tour de boucle	192	1	63	255
Fin du septième tour de boucle	384	0	255	255

2. Montrer que la fonction `mystere` termine.

À chaque tour de boucle, y est divisé par 2 donc la suite des y successifs définit une suite d'entiers strictement décroissante. y est un variant de boucle et le programme termine.

3. Conjecturer la valeur de `mystere(x,y)` et démontrer cette conjecture à l'aide d'un invariant de boucle bien choisi (on pourra appeler x_i, y_i et a_i les valeurs respectives de x, y et a à l'entrée du i^e ($i \geqslant 0$) tour de boucle).

On conjecture que `mystere(x,y)` *renvoie* xy.
Considérons l'invariant : "à l'entrée du i^e*tour de boucle,* $a_i + x_i y_i = x * y$
- Pour $i = 0$*, on a* $a_0 = 0$, $x_0 = x$, $y_0 = y$ *donc* $a_0 + x_0.y_0 = x.y$.
- Soit $i \geqslant 0$*. On suppose qu'au rang* i*, on a* $a_i + x_i y_i = xy$.
. *Soit* y_i *pair alors* $x_{i+1} = x_i + x_i = 2x_i$, $y_{i+1} = \frac{y_i}{2}$ *et* $a_{i+1} = a_i$.
Donc $a_{i+1} + x_{i+1}y_{i+1} = a_i + 2x_i.\frac{y_i}{2} = a_i + x_i y_i = xy$.
. *Soit* y_i *impair alors* $x_{i+1} = xi + x_i = 2x_i$, $y_{i+1} = \frac{y_i - 1}{2}$ *et* $a_{i+1} = a_i + x_i$.
Donc $a_{i+1} + x_{i+1}y_{i+1} = a_i + x_i + 2x_i.\frac{y_i-1}{2} = a_i + x_i + x_i(y_i - 1) = a_i + x_i y_i = xy$.

Cela prouve l'invariant de boucle.
À la dernière étape, on a $a_{i-1} + x_{i-1}y_{i-1} = xy$ *avec* $y_{i-1} = 1$ *donc* $y_i = 0$, $a_i = a_{i-1} + x_{i-1} = a_{i-1} + x_{i-1}y_{i-1} = xy$
donc `mystere(x,y)` *renvoie bien* $x.y$ *ce qui achève la correction de programme.*

4. Quelle est la complexité de cette fonction, en termes de comparaisons, dans le meilleur des cas et dans le pire des cas ? expliquer.

On choisit de compter les comparaisons. Appelons $y_0 = y$ *la valeur initiale de* y *et définissons* $k \in \mathbb{N}$ *tel que* $2^k \leqslant y_0^{k+1}$.
y *est divisé par 2 à chaque tour de boucle. Au bout de* k *boucles,* y *prend la valeur* $\frac{y_0}{2^k} \in [1,2[$ *donc* $\frac{y_0}{2^k} = 1$ *et il y a* $k+1$ *boucles au total.*
- Dans le meilleur des cas, y *est toujours pair, il y a* $k+1$ *comparaisons.*
- Dans le pire des cas, y *est toujours impair, il y a dans chaque tour de boucle, une nouvelle comparaison* `y%2==1`
. Au total, on en a $2 * (k+1) = 2(k+2)$ *comparaisons.*
On obtient en conclusion, une complexité en $O(k) = O(log\, y)$ *soit une complexité logarithmique en* y.

Exercice 2

Recherche du nombre d'éléments distincts dans une liste :

5. Écrire une fonction `Premier(L,i)` qui renvoie `True` si `L[i]` apparaît pour la première fois en i^e position dans la liste `L` et `False` sinon.

```
def Premier(L,i) :
    test=False
    j=0
    while j<=i and not test :
        test=(L[j]==L[i])
        j=j+1
    return(j==i+1)
```

6. Écrire, en utilisant la question précédente, une fonction `NombreDistincts(L)` qui renvoie le nombre d'éléments distincts dans `L`.

```
def NombreDistincts(L) :
    assert type(L)==list, "type d_argument incorrect"
    n=len(L)
    nombre=0
    for i in range(n) :
        if Premier(L,i) :
            nombre+=1
    return(nombre)
```

7. Écrire un code qui permet de tester que la fonction précédente renvoie le bon résultat avec la liste $[6,4,7,4,0,3,6]$

```
assert NombreDistincts([6,4,7,4,0,3,6])==5
```

8. Quelle est la complexité en comparaisons de la fonction précédente en fonction de la longueur n de la liste `L` dans le pire des cas ?

Dans la fonction `Premier(L,i)`, il y a dans le pire des cas i appels à la boucle `while` et à chaque appel, une comparaison pour le test. Donc au total, $2i$ comparaisons donc une complexité linéaire en i.
Dans `NombreDistincts(L)`, il y a n tours de boucles et à chaque tour, un appel à la fonction `Premier(L,i)` de complexité linéaire. On obtient une complexité de l'ordre de $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ soit une complexité quadratique.

Cas d'une liste triée :

On suppose que la liste `L` est triée en ordre croissant.

9. Écrire une fonction `NombreDistinctsTri(L)` qui renvoie le nombre d'éléments distincts dans la liste `L`.

```
def NombreDistinctsTri(L) :
    n=len(L)
    nombre=1
    for i in range(n-1) :
        if L[i]<L[i+1] :
            nombre=nombre+1
    return(nombre)
```

10. Quelle est la complexité de la fonction `NombreDistinctsTri(L)` en fonction de la longueur n de la liste `L`?

Il y a $n-1$ tours de boucle et pour chaque boucle une comparaison, donc une complexité en comparaisons de $n-1$ soit une complexité linéaire.

11. Définir un invariant de la boucle constituant la fonction `NombreDistinctsTri(L)` et justifier la correction de cette fonction.

Un invariant de boucle est : "à l'entrée du i^e tour, nombre contient le nombre d'éléments distincts dans $L[:i+1]$".
- Pour $i=0$, nombre =1 = nombre d'éléments distincts dans $L[:1]$ qui est un singleton.
- Supposons cet invariant à l'entrée du i^e tour. En sortie de ce tour :
. soit L[i]=L[i+1] alors nombre ne change pas et contient le nombre d'éléments distincts de L[:i+2].
. soit L[i]=L[i+1] alors nombre est incrémenté de 1 et contient le nombre d'éléments distincts de L[:i+2].
Donc en sortie du tout i, et donc à l'entrée du tour $i+1$, nombre contient le nombre d'éléments distincts de L[:i+2].
- Au dernier tour, $i=n-2$. En entrée de boucle, nombre contient le nombre d'éléments distincts de L[:n-1] et nombre contient le nombre d'éléments distincts de L[:n]=L ie le résultat cherché.

Vérification qu'une liste est bien triée :

12. Écrire une fonction `TestTri(L)` qui renvoie `True` ou `False` selon que la liste `L` est triée ou pas.

```
def TestTri(L) :
    test=True
    i=0
    n=len(L)
    while i<=n-2 and test :
        test=(L[i]<=L[i+1])
        i=i+1
    return(test)
```

13. Quelle est la complexité C_n, en nombre de comparaisons, de la fonction `TestTri(L)` en fonction de la longueur n de la liste `L`?

Il y a $n-1$ appels à la boucle `while` et pour chaque tour, une comparaison pour `test`. Au total, cela donne une complexité $C_n = 2(n-1) = O(n)$ linéaire.

14. Etant donnée une liste `X` de longueur n, que représentent les commandes X[:n//2] et X[n//2 :] ?

X[:n//2] renvoie la sous-liste de X constituée des éléments d'indice inférieur strict à partie entière de n/2.
X[n//2 :] renvoie la sous-liste de X constituée des éléments d'indice supérieur large à partie entière de n/2.

15. En déduire un algorithme récursif `TestTriRec(L)` qui, étant donnée une liste `L` de longueur $n \geqslant 2$, renvoie `True` ou `False` selon que la liste `L` est triée ou pas.

```
def TestTriRec(L) :
    n=len(L)
    if n==2 :
        return(L[0]<=L[1])
    else :
        return(TestTriRec(L[ :n//2]) and TestTriRec(L[n//2 :]))
```

16. Déterminer la complexité C'_n, en nombre de comparaisons, pour une liste de longueur $n = 2^p$ avec $p \in \mathbb{N}^*$.

Si $n = 2$, il y a une comparaison. Si $n = 2^p$ avec $p \geqslant 2$, on fait appel à TestTriRec sur deux listes de longueur $n/2 = 2^{p-1}$. Donc $C'_n = 2C'_{n/2}$ ie $C'_{2^p} = 2C'_{2^{p-1}}$. Par itération, on obtient $C'_{2^p} = 2^{p-1}C'_2 = 2^p = n$.
On obtient une complexité linéaire en n.

17. Conclusion ?

Les fonctions TestTri et TestTriRec ont la même complexité temporelle. Néanmoins, la forme récursive utilise des copies de listes lors des slicing ce qui donne une complexité spatiale moins intéressante que pour la fonction TestTri. La forme itérative est donc à privilégier.

Optimisation d'un algorithme de tri

On peut optimiser l'algorithme du tri par insertion, en recherchant par dichotomie la position d'insertion.

18. Écrire une fonction `ajouter(L,x)` qui étant donnés une liste triée `L` et un élément `x`, modifie `L` en insérant `x` dans la liste `L` à la bonne place (de sorte d'obtenir une liste triée dans l'ordre croissant). On recherchera la position d'insertion par dichotomie.

```
def ajouter(L,x) :
    g,d=0,len(L)-1
    while g<=d :
        m=(g+d)//2
        if x==L[m] :
            return(L.insert(m,x))
        elif x<L[m] :
            d=m-1
        else :
            g=m+1
    return(L.insert(g,x))
```

19. Estimer, en justifiant brièvement, la complexité de la fonction `ajouter()` si on suppose que la fonction `insert()` a une complexité en O(1).

La complexité est équivalente à celle d'une dichotomie, soit une complexité logarithmique en $O(n\log n)$.

20. On donne la fonction `TriInsertion(L)` qui permet de trier une liste `L` par l'algorithme du tri par insertion et qui utilise la fonction `ajouter(L,x)` définie précédemment.

```
def TriInsertion(L) :
    for i in range(1,len(L)) :
        SL = L[0 :i]  sous-liste triee
        x=L.pop(i)
        ajouter(SL,x)
        L[0 :i+1] = SL
    return L
```

Expliquer simplement pourquoi, si n est la longueur de la liste, alors la complexité de cet algorithme est en $O(n\log(n))$. On précise que les méthodes "classiques" des listes comme pop, append... fonctionnent en temps constant.

Il y a n tours de boucle et à chaque tour i, on appelle ajouter(SL,x) qui a une complexité de logi. Au total, cela donne une complexité de l'ordre de $\sum_{i=1}^{n-1} log(i) \leqslant n.log(n)$ d'où une complexité en $O(n\log n)$.

En déduire l'avantage et l'inconvénient de cette version du tri par insertion par rapport à l'algorithme « classique » vu en cours.

L'algorithme classique est en complexité quadratique donc :
- Avantage : lorsque la liste est en sens décroissant, on gagne en complexité.
- Inconvénient : lorsque la liste est triée, on perd en complexité.

21. Donner une version récursive, notée `TriInsertionRec(L)` de la fonction `TriInsertion(L)`.

```
def TriInsertionRec(L) :
    if len(L)==1 :
        return(L)
    else :
        x=L.pop()
        TriInsertionRec(L)
        ajouter(L,x)
    return(L)
```