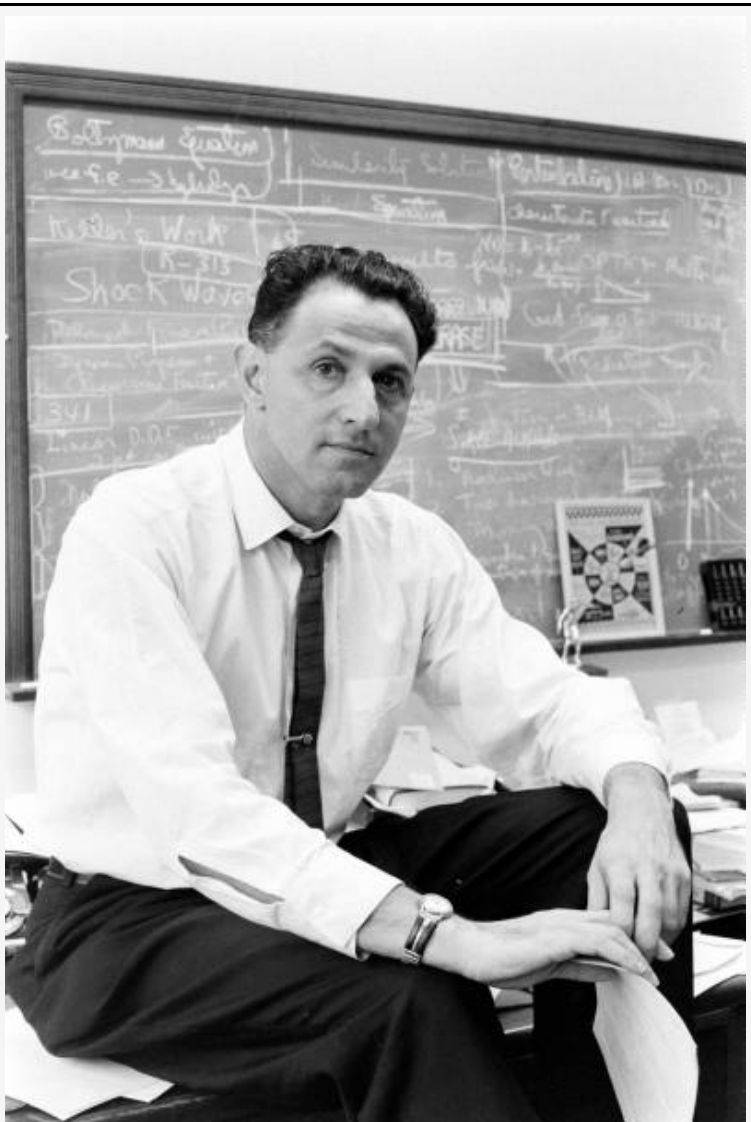

TP d'introduction à la programmation dynamique

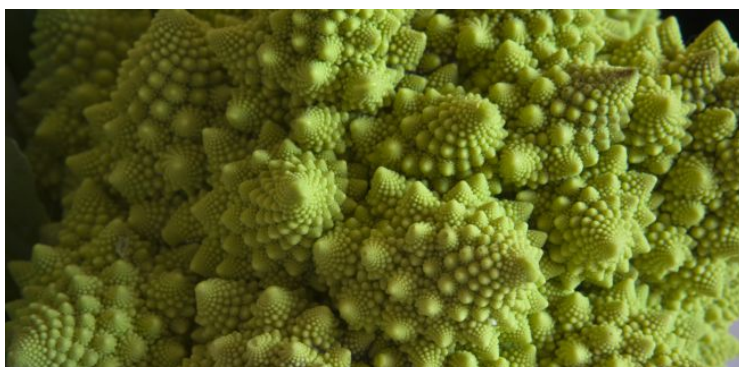
Bellmann l'initiateur de la méthode :



Il y a des blocs de code et des blocs de markdown déjà placés, mais vous pouvez en ajouter afin de mieux organiser votre travail.

Vos codes doivent être testés, et les test visibles

1. Suite de Fibonacci





[\[Micmaths\] Tuto Maths & Cuisine : Fibonacci et le nombre d'or](#)

[\[Micmaths\] Le nombre d'or](#)

Afin de s'échauffer et se familiariser à la résolution de problèmes via la programmation dynamique, cette première partie va la mettre en oeuvre sur un cas très simple, la suite de Fibonacci définie par le schéma de récurrence :

$$\begin{cases} \forall n \in \mathbb{N}, f_{n+2} = f_{n+1} + f_n \\ f_0 = 0 \\ f_1 = 1 \end{cases}$$

1.1. Ecrire un code de f en récursif



Il est important que ce premier code implémente exactement la définition donnée ci dessus, sans aucune optimisation.

In [242]:

```
# code de f sans optimisation
def fibonacci(n):
    if n <= 1 :
        return n
    else :
        return fibonacci(n-1) + fibonacci(n-2)
```

In [243]:

```
fibonacci(5)
fibonacci(10)
fibonacci(20)
fibonacci(30) #35
```

Out[243]:

832040

In []:

1.2. En utilisant le module graphviz, représenter le multi-graphe des appels

[Graphviz](#)

- les noeuds du graphe sont étiquetés par les entiers sur lesquels f est appliquée
- un arc de (p) à (q) indique que la fonction qui a été appelée avec le paramètre 'p' appelle la fonction avec le paramètre (q)

Par exemple, comme $f(12) = f(11) + f(10)$, calculer $f(12)$ générera des arcs (12)-->(11) et (12)-->(10).

C'est bien un multi-graphe, il va y avoir plusieurs arcs entre certains noeud.

In [244]:

```
from graphviz import Graph
# code pour tracer le multi-graphe des appels
def fibonacci_graph(n, dot= Graph()):
    dot.node(str(n))

    if n <= 1 :
        return n, dot
    else :
        dot.node(str(n-1))
        dot.node(str(n-2))
        dot.edge(str(n), str(n-1))
        dot.edge(str(n), str(n-2))

    return fibonacci_graph(n-1, dot) + fibonacci_graph(n-2, dot) , dot
```

In [245]:

```
v1, g1 = fibonacci_graph(7)
display(g1)
```

*** vos commentaires... Le 5 appel 4 et 3 Le 4 appel 3 et 2 Le 3 appel 2 fois 2.... Le nombre d'arc sortants de n à n-1 est égale au nombre d'ac entrant à n.

=> il sort 2 fois plus d'arc que ce qu'il rentre

In [246]:

```
v, g = fibonacci_graph(15)
display(g)
```

1.3. Etude empirique de la complexité

- Réaliser la représentation graphique du nombre d'appels à fibo en fonction de n .
- Déterminer par essai/erreur une équation de courbe approximant le tracé précédant (je vous aide : la

réponse pourrait faire apparaître du $\phi_n = \left(\frac{1+\sqrt{5}}{2}\right)^n$, et pour ceux qui veulent une approximation encore plus

fine, ajouter un terme en $\frac{1}{\phi_n}$)

In [247]:

```
def fibonacci_compte(n):
    if n <= 1 :
        return n, 1
    else :
        v_1, n_1 = fibonacci_compte(n-1)
        v_2, n_2 = fibonacci_compte(n-2)
        return v_1 + v_2, n_1 + n_2 + 1
```

In [248]:

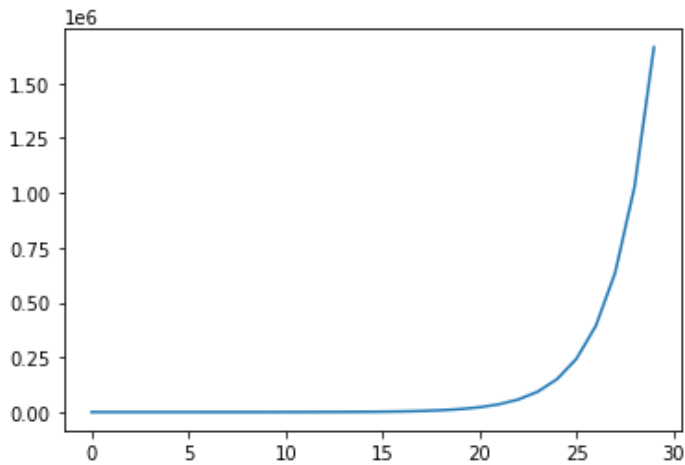
```
# Code de la fonction qui produit le graphique du nombre d'appels à 'f' en fonction de 'n'
tab_n = list(range(30))
tab_nb_op = list(map(lambda n: fibonacci_compte(n)[1], tab_n))
```

In [249]:

```
import seaborn as sns
sns.lineplot(x=tab_n, y=tab_nb_op)
```

Out[249]:

<AxesSubplot:>

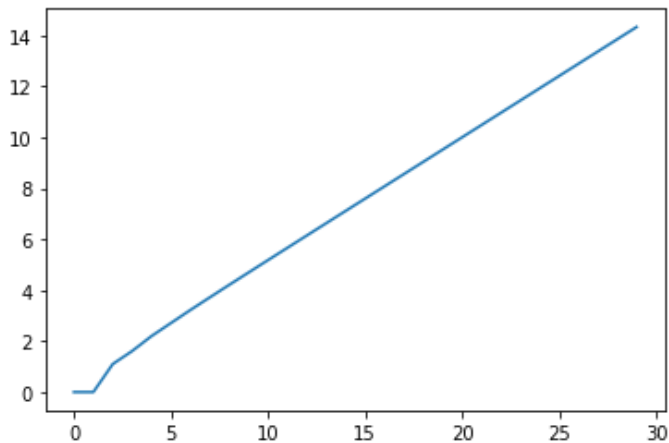


In [250]:

```
import numpy as np
# code de la recherche d'une expression approximant le nombre d'appels
tab_nb_op_log = list(map(np.log, tab_nb_op))
sns.lineplot(x=tab_n, y=tab_nb_op_log)
```

Out[250]:

<AxesSubplot:>



**** votre analyse... Si droite parfaite, $\log(nb_op) = a \cdot n + b$**

$a = 14/30 = 0.5$

$b = 0$

$\log(nb_op) = 0.5 \cdot n + 0$

$nb_op = \exp(0.5n) = (\exp(0.5))^n = 1.6487212707001282^n$

In [251]:

```
np.exp(0.5)
```

Out[251]:

1.6487212707001282

```
In [252]:
```

```
"""
On cherche a, b, c, d, e, f, ...
nb_op = a*phi^n + b*(1/phi)^n + c*n^2 + d*n + e*log(n+1) + f*1 # +g*...
Il faut beaucoup moins de coefficient que de données.
Par rapport à ce qu'on recherche on fait une recherche linéaire car on cherche les coeffi
cient de la forme linéaire
"""
```

```
Out[252]:
```

```
"\nOn cherche a, b, c, d, e, f, ... \nnb_op = a*phi^n + b*(1/phi)^n + c*n^2 + d*n + e*log(n+1) + f*1 # +g*... \nIl faut beaucoup moins de coefficient que de données. \nPar rapport à ce qu'on recherche on fait une recherche linéaire car on cherche les coefficient de la forme linéaire\n"
```

```
In [253]:
```

```
import pandas as pd

phi = (1+np.sqrt(5))/2

data = pd.DataFrame(tab_n, columns=["n"])
data["phi^n"] = data.apply(lambda ligne: phi**ligne[0], axis=1)
data["1/phi^n"] = data.apply(lambda ligne: 1/phi**ligne[0], axis=1)
data["n^2"] = data.apply(lambda ligne: ligne[0]**2, axis=1)
data["log(n+1)"] = data.apply(lambda ligne: np.log(ligne[0]+1), axis=1)
data["1"] = data.apply(lambda ligne: 1, axis=1)

display(data.head())
```

	n	phi^n	1/phi^n	n²	log(n+1)	1
0	0	1.000000	1.000000	0.0	0.000000	1
1	1	1.618034	0.618034	1.0	0.693147	1
2	2	2.618034	0.381966	4.0	1.098612	1
3	3	4.236068	0.236068	9.0	1.386294	1
4	4	6.854102	0.145898	16.0	1.609438	1

```
In [254]:
```

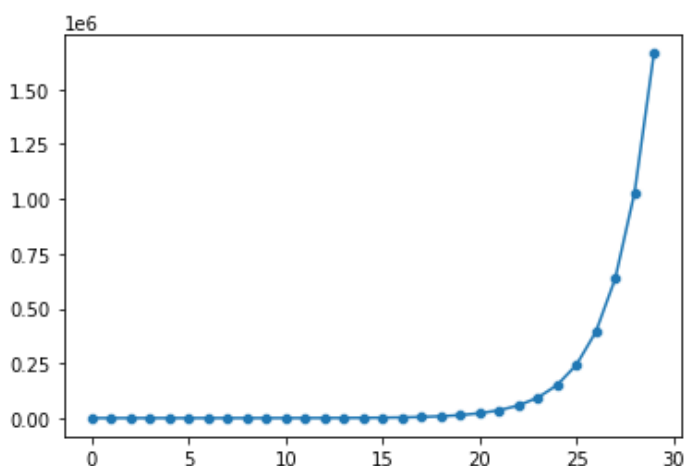
```
from sklearn.linear_model import LinearRegression

model = LinearRegression().fit(data, tab_nb_op)
tab_nb_op_approx = model.predict(data)

sns.lineplot(x=tab_n, y=tab_nb_op)
sns.scatterplot(x=tab_n, y=tab_nb_op_approx)
```

```
Out[254]:
```

```
<AxesSubplot:>
```



In [255]:

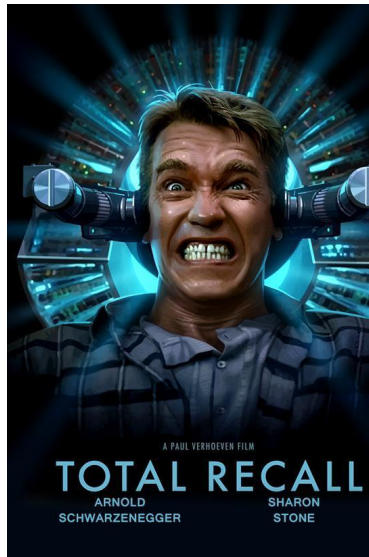
```
display(model.coef_)
```

```
array([ 0.67020627,  1.44721411, -6.58471348, -0.01071309, -5.27788845,
        0.          ])
```

le modele trouvé est $nb_p = 1.44 * phi^n + -6.58 * (1/phi)^n + -0.011 * n^2 + 0.67 * n + -5.27 * \log(n+1) + 0 * 1$
selon l'étude expérimentale

Attention !, phi^n est tellement plus grand que les autres qu'il est le seul significatifs

1.4. Mémoïsation



Rappel : la mémoïsation consiste à stocker les résultats calculés, de façon à ne pas avoir à les recalculer si on en a besoin à nouveau. La structure de données python 'dictionnaire' est bien adaptée pour stocker les résultats déjà calculés.

Ce dictionnaire possède pour clefs les paramètres d'appel et pour valeurs le résultat de la fonction appelée avec ces paramètres.

Selon votre aisance en algorithmique et en python, implémenter la mémoïsation de votre fonction récursive par l'une des méthodes suivantes :

- ré-écrire votre fonction récursive, en implémentant la mémoïsation
 - piste 1 : ajouter le dictionnaire comme paramètre d'entrée supplémentaire
 - piste 2 : utilisation d'une closure (fermeture en français) : créer un dictionnaire ; dans le code de la fonction, avant d'appeler la fonction sur un sous problème, regarder dans le dictionnaire si la valeur a déjà été calculée ; si elle ne correspond à aucune entrée du dictionnaire, la calculer et la stocker dans le dictionnaire ; utiliser la valeur stockée dans le dictionnaire.
- utiliser un décorateur, c'est certainement la solution la plus élégante car elle peut être parfaitement générique (attention à bien comprendre ce que vous faites et à expliquer clairement la magie...)

In [256]:

```
# Vos codes pour la mémoïsation méthode 1
def fibo_memo(n, dico=dict()):
    if n not in dico :
        if n <= 1 :
            dico[n] = n
        else :
            v_1 = fibo_memo(n-1, dico)
            v_2 = fibo_memo(n-2, dico)
            dico[n] = v_1 + v_2
    return dico[n]
```

In [257]:

```
display(fibo_memo(100)) # marche pour 2000
display(1.44*phi**100) #ordre de grandeur de calcul sans memoisation ordre 11^20
```

```
354224848179261915075
```

```
1.1405820093816599e+21
```

In []:

Bien expliquer **votre** solution, avec **vos** mots, et si vous avez choisi la solution avec décorateur, mettez le doigt sur les propriétés de Python qui autorisent ce mécanisme (la même chose ne fonctionnerait pas en c++ par exemple)

*** vos explications ...

1.5. Utiliser graphviz pour représenter le graphe des appels sur la fonction mémorisée

NB : le multi-graphe est maintenant un graphe !

In [258]:

```
# votre code pour produire le multi-graphe des appels
# Vos codes pour la mémorisation méthode 1
def fibo_memo_graph(n, dico=dict(), dot=Graph()):
    dot.node(str(n))
    if n not in dico :
        if n <= 1 :
            dico[n] = n, dot
        else :
            dot.node(str(n-1))
            dot.node(str(n-2))
            dot.edge(str(n), str(n-1))
            dot.edge(str(n), str(n-2))
            v_1 = fibo_memo_graph(n-1, dico, dot)
            v_2 = fibo_memo_graph(n-2, dico, dot)
            dico[n] = v_1 + v_2
    return dico[n], dot
```

In [259]:

```
v, g = fibo_memo_graph(10)
display(g)
```

*** vos commentaires...

1.6. Refaire l'étude de complexité empirique dans le cas de la fonction mémorisée

In [260]:

```
#nbop = a*n +b todo if
```

*** vos commentaires...

1.7. Programmation dynamique

Le passage à la programmation dynamique va consister à ne conserver que les résultats nécessaires

Ecrire le programme mettant en oeuvre cet objectif.

```
# votre code
def fibo_dyna(n):
    if n <= 1 :
        return n
    else :
        prec, prec_prec = 1, 0

        for _ in range(1, n) :
            prec, prec_prec = prec + prec_prec, prec
        return prec
```

```
print(list(map(lambda n: fibonacci(n) != fibo_dyna(n), tab_n)))
print(sum(map(lambda n: fibonacci(n) != fibo_dyna(n), tab_n)))
```

*** vos commentaires ...
avant la boucle $O(1)$
'+ $O(n)$
= $O(n)$

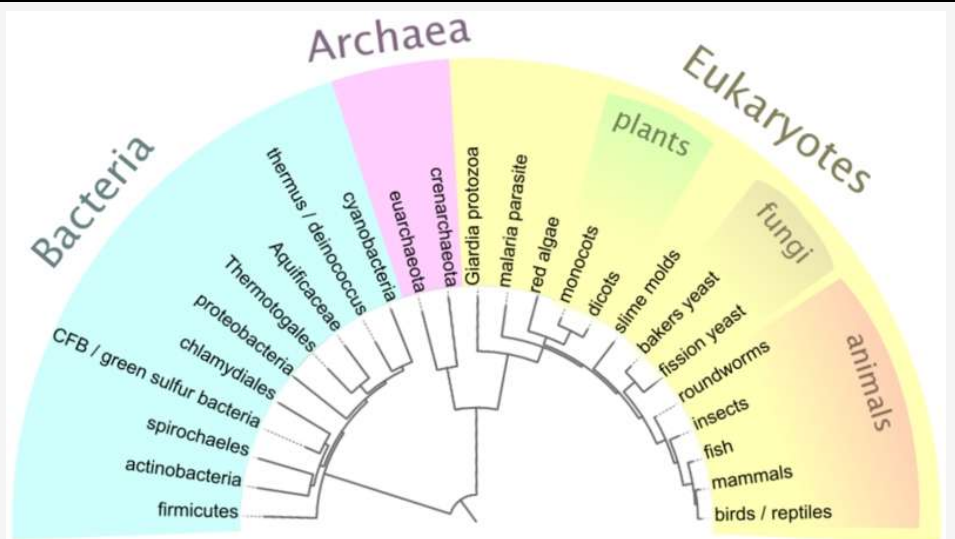
1.8. Refaire l'étude empirique de complexité sur cette nouvelle fonction

```
# votre code
```

***** vos commentaires...**

2. Distance d'édition

Elle peut servir à déterminer la distance entre deux brins d'ADN



Soient M

et P

deux chaînes de caractères. La distance d'édition entre M

et P

est le nombre minimal d'opérations permettant de transformer M

en P

, les opérations étant prises parmi :

- substitution d'un caractère de M par un caractère différent de P (coût $c_s > 0$)
)
- insertion dans M d'un caractère de P (coût $c_i > 0$)
)
- suppression d'un caractère de M (coût $c_d > 0$)
)

Le but de cette partie du TP est de calculer la distance d'édition entre deux mots ainsi que de calculer une suite de transformations de coût minimal permettant de transformer un mot en un autre mot (le coût d'une transformation est la somme des coûts des opérations qui la composent).

Afin de parvenir à ce résultat, plusieurs phases vous sont demandées, présentées dans les questions suivantes.

Pour votre travail, les valeurs des coûts des opérations n'auront pas d'importance (dans des cas applicatifs ils ont évidemment de l'importance).

2.1. Equation de récurrence

L'équation de récurrence était fournie pour la suite de Fibonacci, ici c'est à vous de la trouver (elle est en fait dans la partie cours/TD). En général, c'est cette phase qui est la plus hardue, car 1) il faut effectivement résoudre le problème, 2) toutes les décompositions ne sont pas aisément substituables par de la programmation dynamique

1. Ecrire l'équation de récurrence

2. Justifier votre équation (ébauche de preuve de la correction de votre algorithme)

*** équation de récurrence et vos explications

ci = cout d'insertion

cd = cout de suppression

cs = cout de substitution

$$\left\{ \begin{array}{l} d(E, E) = 0 \\ d(m, E) = |m| * ci \\ d(E, m) = |m| * cd \\ d(m1.x, m2.x) = d(m1, m2) = (m1 == m2) = 0 \\ m1 \neq m2 \quad d(m1, m2) = \min(cs + d(m1[: -1], m2[: -1]); ci + d(m1[: -1], m2); cd + d(m1, m2[: -1])) \end{array} \right.$$

2.3. Ecrire une fonction récursive calculant la distance d'édition

In [264]:

```
# votre code de la version récursive
def distance(m1, m2, cs, ci, cd):
    if len(m2) == 0:
        return len(m1) * ci
    elif len(m1) == 0:
        return len(m2) * cd
    elif m1[-1] == m2[-1]:
        return distance(m1[:-1], m2[:-1], cs, ci, cd)
```

```

else :
    v_s = cs + distance(m1[:-1], m2[:-1], cs, ci, cd)
    v_i = ci + distance(m1[:-1], m2, cs, ci, cd)
    v_d = cd + distance(m1, m2[:-1], cs, ci, cd)
    return min(v_s, v_i, v_d)

```

In [265]:

```
distance("totozefzea", "totifzefzef", cs = 1, ci=1, cd=1)
```

Out[265]:

3

2.4. Calculer une suite de transformations optimale permettant de transformer une chaîne en une autre

Optimale signifie ici " de coût minimal"

Précédemment, on se contentait de calculer la distance, maintenant on désire obtenir la suite des transformations qui permet d'effectivement transformer le premier mot en le second. La fonction doit renvoyer un affichage compact (par exemple [['s', 4, 'b'], ['d', 3], ['i', 5, 'r'], ...], aisé à manipuler informatiquement).

In [266]:

```

# votre code pour le calcul
def distance(m1, m2, cs, ci, cd):
    if len(m2) == 0:
        return len(m1) * ci, ["i 0 " + m1[k] for k in range(len(m1)-1, -1, -1)]

    elif len(m1) == 0:
        return len(m2) * cd, ["d 0 " for _ in m2]

    elif m1[-1] == m2[-1] :
        d, op = distance(m1[:-1], m2[:-1], cs, ci, cd)
        #return d, ["s " + m1[-1] + " " + str(len(m2)-1)] + op
        return d, op

    else :
        v_s, op_s = distance(m1[:-1], m2[:-1], cs, ci, cd)
        v_s += cs
        op_s = ["s " + m1[-1] + " " + str(len(m2)-1)] + op_s

        v_i, op_i = distance(m1[:-1], m2, cs, ci, cd)
        v_i += ci
        op_i = ["i " + m1[-1] + " " + str(len(m2)-1)] + op_i

        v_d, op_d = distance(m1, m2[:-1], cs, ci, cd)
        v_d += cd
        op_d = ["d " + str(len(m2)-1)] + op_d

        tab_v = [v_s, v_i, v_d]
        tab_op = [op_s, op_i, op_d]

        return min(tab_v), tab_op[np.argmin(tab_v)]

```

In [267]:

```

print(distance("abc", "", cs=1, ci=1, cd=1))
print(distance("", "abc", cs=1, ci=1, cd=1))
print(distance("abc", "abc", cs=1, ci=1, cd=1))
print(distance("abc", "def", cs=1, ci=1, cd=1))
print(distance("abcdefg", "abcbfr", cs=1, ci=1, cd=1))

```

```

(3, ['i 0 c', 'i 0 b', 'i 0 a'])
(3, ['d 0 ', 'd 0 ', 'd 0 '])
(0, [])
(3, ['s c 2', 's b 1', 's a 0'])
(3, ['s g 5', 's e 3', 'i d 2'])

```

La sortie précédente n'est pas très conviviale, écrire une fonction qui génère à partir de la représentation compacte un affichage pertinent. Par exemple, avec des coûts de transformation tous à 1, on pourrait avoir comme affichage :

```
Transformation de 'abcdefg' en 'abcbfr'
Départ                                [cout 0]          abcdefg
Substituer à l'indice 4 par un 'b'    [cout 1] --->    abcdbfg
Effacer le caractère d'indice 3       [cout 1] --->    abcbbfg
Insérer à l'indice 5 le caractère 'r' [cout 1] --->    abcbbfr
                                     [cout total : 3]
```

In [268]:

```
# votre code pour le 'pretty print'
```

(bonus) Au lieu d'un affichage en mode texte, on peut également générer à partir de la représebtation compacte un graphe (c'est une chaîne au sens des graphes dans ce cas) où les sommets sont étiquetés par l'état de la chaîne de caractère et les arcs par la transformation effectuée.

In [269]:

```
# code bonus ; représentation graphique de la représentation compacte
```

2.5. Représenter le multi-graphe des appels

In [270]:

```
# votre code
def distance_graph(m1, m2, cs, ci, cd, dot=Graph()):
    dot.node(m1 + " " + m2)

    if len(m2) == 0:
        return len(m1) * ci, ["i 0 " + m1[k] for k in range(len(m1)-1, -1, -1)], dot

    elif len(m1) == 0:
        return len(m2) * cd, ["d 0 " for _ in m2], dot

    elif m1[-1] == m2[-1] :
        dot.node(m1[:-1] + " " + m2[:-1])
        dot.edge(m1 + " " + m2, m1[:-1] + " " + m2[:-1])
        d, op, dot = distance_graph(m1[:-1], m2[:-1], cs, ci, cd, dot)
        return d, op, dot

    else :
        dot.node(m1[:-1] + " " + m2[:-1])
        dot.edge(m1 + " " + m2, m1[:-1] + " " + m2[:-1])
        v_s, op_s, dot = distance_graph(m1[:-1], m2[:-1], cs, ci, cd, dot)
        v_s += cs
        op_s = ["s " + m1[-1] + " " + str(len(m2)-1)] + op_s

        dot.node(m1[:-1] + " " + m2)
        dot.edge(m1 + " " + m2, m1[:-1] + " " + m2)
        v_i, op_i, dot = distance_graph(m1[:-1], m2, cs, ci, cd, dot)
        v_i += ci
        op_i = ["i " + m1[-1] + " " + str(len(m2)-1)] + op_i

        dot.node(m1 + " " + m2[:-1])
        dot.edge(m1 + " " + m2, m1 + " " + m2[:-1])
        v_d, op_d, dot = distance_graph(m1, m2[:-1], cs, ci, cd, dot)
        v_d += cd
        op_d = ["d " + str(len(m2)-1)] + op_d

    tab_v = [v_s, v_i, v_d]
    tab_op = [op_s, op_i, op_d]
```

```
return min(tab_v), tab_op[np.argmin(tab_v)], dot
```

In [271]:

```
v, op, g_2 = distance_graph("cdefg", "cbfr", cs=1, ci=1, cd=1)
display(g_2)
```

2.6. Mémoïser la fonction

In [272]:

```
# votre code les clés du memo m1 + " " + m2
def distance_graph_memo(m1, m2, cs, ci, cd, dico=dict(), dot=Graph()):
    dot.node(m1 + " " + m2)
    if m1 not in dico:
        dico[m1] = dict()
    if m2 not in dico[m1]:
        if len(m2) == 0:
            dico[m1][m2] = len(m1) * ci, ["i 0 " + m1[k] for k in range(len(m1)-1, -1,
-1)], dot
        elif len(m1) == 0:
            dico[m1][m2] = len(m2) * cd, ["d 0 " for _ in m2], dot
        elif m1[-1] == m2[-1] :
            dot.node(m1[:-1] + " " + m2[:-1])
            dot.edge(m1 + " " + m2, m1[:-1] + " " + m2[:-1])
            d, op, dot = distance_graph_memo(m1[:-1], m2[:-1], cs, ci, cd, dico, dot)
            dico[m1][m2] = d, op, dot
        else :
            dot.node(m1[:-1] + " " + m2[:-1])
            dot.edge(m1 + " " + m2, m1[:-1] + " " + m2[:-1])
            v_s, op_s, dot = distance_graph_memo(m1[:-1], m2[:-1], cs, ci, cd, dico, dot)
            v_s += cs
            op_s = ["s " + m1[-1] + " " + str(len(m2)-1)] + op_s
            dot.node(m1[:-1] + " " + m2)
            dot.edge(m1 + " " + m2, m1[:-1] + " " + m2)
            v_i, op_i, dot = distance_graph_memo(m1[:-1], m2, cs, ci, cd, dico, dot)
            v_i += ci
            op_i = ["i " + m1[-1] + " " + str(len(m2)-1)] + op_i
            dot.node(m1 + " " + m2[:-1])
            dot.edge(m1 + " " + m2, m1 + " " + m2[:-1])
            v_d, op_d, dot = distance_graph_memo(m1, m2[:-1], cs, ci, cd, dico, dot)
            v_d += cd
            op_d = ["d " + str(len(m2)-1)] + op_d
            tab_v = [v_s, v_i, v_d]
            tab_op = [op_s, op_i, op_d]
            dico[m1][m2] = min(tab_v), tab_op[np.argmin(tab_v)], dot
    return dico[m1][m2]
```

2.7. Représenter le graphe des appels de la fonction mémoïsée

In [273]:

```
# votre code
v, op, g_2 = distance_graph_memo("cdefg", "cbfr", cs=1, ci=1, cd=1)
display(g_2)
```

2.8. Mettre en oeuvre la programmation dynamique

In [274]:

```
# votre code
def distance_dyna(m1, m2, cs, ci, cd):
    tab = [[0] * (len(m2) + 1)] * 2
    v0 = 0
    v1 = 1

    tab[v0] = [i * cs for i in range(len(m2)+1)]

    for i in range(len(m1)):
        tab[v1][0] = (i + 1) * cs

        for j in range(len(m2)):
            delCost = tab[v0][j + 1] + cd
            insCost = tab[v1][j] + ci
            subCost = tab[v0][j]
            if m1[i] != m2[j]:
                subCost += cs
            tab[v1][j + 1] = min(subCost, delCost, insCost)
        v0, v1 = v1, v0
    return tab[v0][len(m2)]
```

In [275]:

```
v2 = distance_dyna("cdefg", "cbfr", cs=1, ci=1, cd=1)
display(v2)
```

3

***** vos explications** On calcule la ligne des distance courantes et on sauvergade la ligne précédente ainsi de suite jusqu'à avoir calculé la matrice des couts. et obtenir le dsitance totale.

2.9. Trouver une/des mesures vous permettant d'estimer le gain par rapport à la version récursive du début

Cette question n'est pas triviale, les deux chaînes ne sont pas nécessairement de la même longueur, le nombre d'appels dépend des lettres utilisées,... A vous de trouver une mesure satisfaisante qui permettra une représentation graphique mettant en évidence le gain obtenu par la programmation dynamique.

***** vos explications...**

In [276]:

```
# votre code
def distance_dyna_compte(m1, m2):
    return (len(m1) + 1) * len(m2) + 1
```

In [277]:

```
def distance_compte(m1, m2):
    if len(m2) == 0:
        return 1
    elif len(m1) == 0:
        return 1
    elif m1[-1] == m2[-1]:
        return 1 + distance_compte(m1[:-1], m2[:-1])
    else:
        op_s = distance_compte(m1[:-1], m2[:-1])
        op_i = distance_compte(m1[:-1], m2)
        op_d = distance_compte(m1, m2[:-1])
        return 1 + op_s + op_i + op_d
```

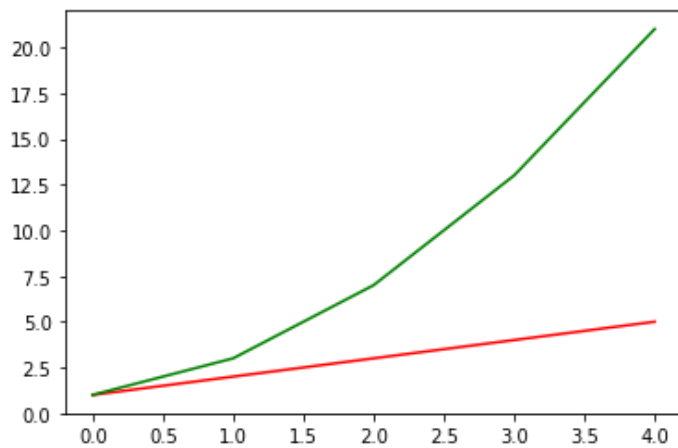
In [278]:

```
tab_n = [i for i in range(5)]
tab_nb_op1 = list(map(lambda n: distance_compte("a" * n, "a" * n), tab_n))
tab_nb_op2 = list(map(lambda n: distance_dyna_compte("a" * n, "a" * n), tab_n))
sns.lineplot(x=tab_n, y=tab_nb_op1, color="red")
```

```
sns.lineplot(x=tab_n, y=tab_nb_op2, color="green")
```

Out[278]:

<AxesSubplot:>

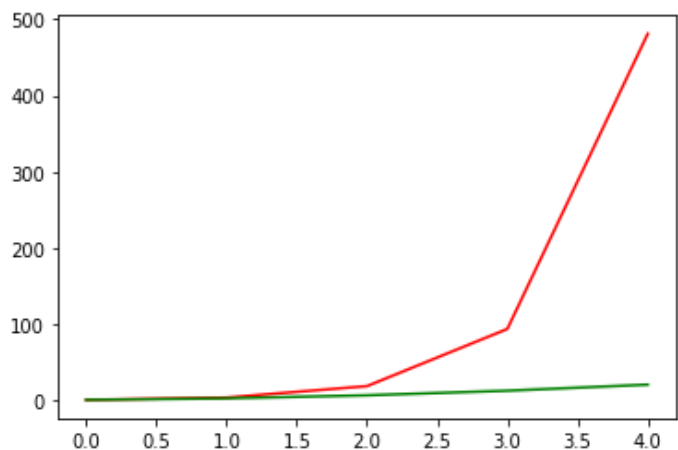


In [279]:

```
tab_n = [i for i in range(5)]
tab_nb_op1 = list(map(lambda n: distance_compte("a" * n, "b" * n), tab_n))
tab_nb_op2 = list(map(lambda n: distance_dyna_compte("a" * n, "b" * n), tab_n))
sns.lineplot(x=tab_n, y=tab_nb_op1, color="red")
sns.lineplot(x=tab_n, y=tab_nb_op2, color="green")
```

Out[279]:

<AxesSubplot:>

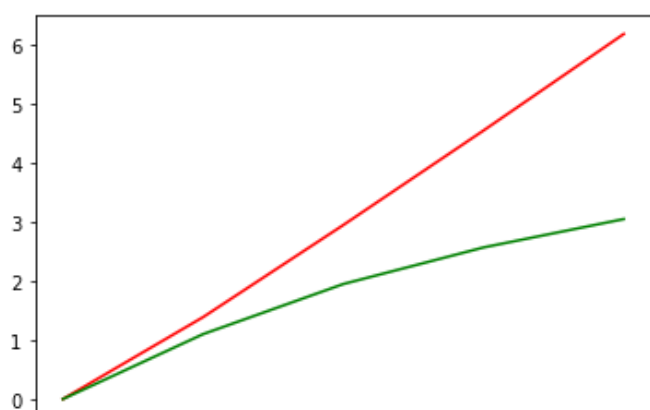


In [280]:

```
tab_nb_op1_log = list(map(np.log, tab_nb_op1))
tab_nb_op2_log = list(map(np.log, tab_nb_op2))
sns.lineplot(x=tab_n, y=tab_nb_op1_log, color="red")
sns.lineplot(x=tab_n, y=tab_nb_op2_log, color="green")
```

Out[280]:

<AxesSubplot:>

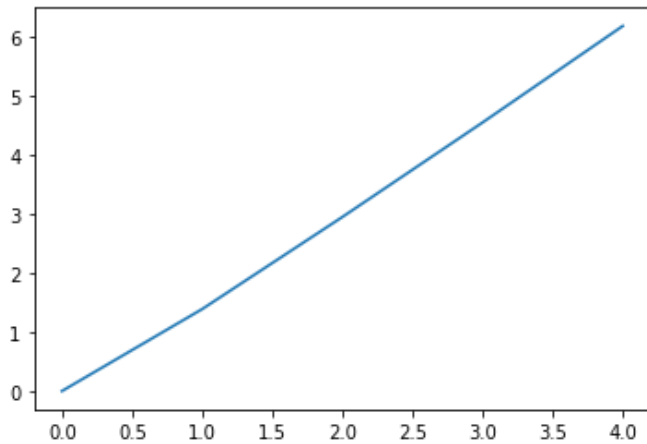


In [281]:

```
tab_nb_op1_log = list(map(np.log, tab_nb_op1))
sns.lineplot(x=tab_n, y=tab_nb_op1_log)
```

Out[281]:

<AxesSubplot:>



In [282]:

```
e = 0
for i in range(1, len(tab_nb_op1_log)):
    e += tab_nb_op1_log[i] / i
e /= len(tab_nb_op1_log) - 1
display(e)
```

1.4792280655798882

In [283]:

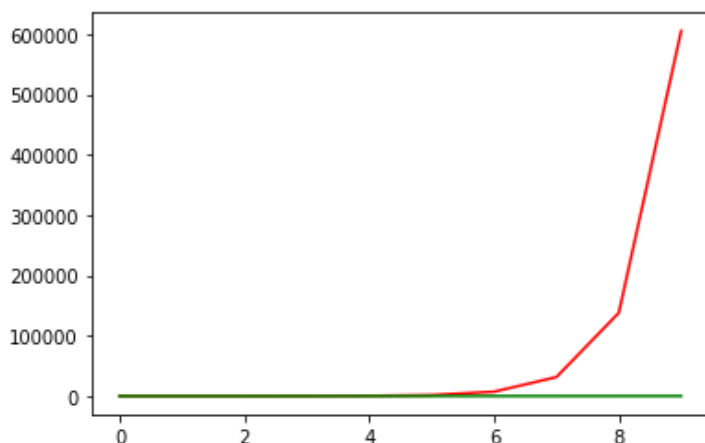
```
def distance_complexite(n):
    return np.exp(1.4792280655798882) ** n
def distance_dyna_complexite(n, m):
    return (n + 1) * m + 1
```

In [284]:

```
tab_n_cmp = [i for i in range(10)]
tab_nb_op1_cmp = list(map(lambda n: distance_complexite(n), tab_n_cmp))
tab_nb_op2_cmp = list(map(lambda n: distance_dyna_complexite(n, n), tab_n_cmp))
sns.lineplot(x=tab_n_cmp, y=tab_nb_op1_cmp, color="red")
sns.lineplot(x=tab_n_cmp, y=tab_nb_op2_cmp, color="green")
```

Out[284]:

<AxesSubplot:>



- Le programme dynamique est plus rapide si les mots sont différents (courbe verte)
- Le programme récursif est plus rapide sur les mots identiques (courbe rouge)

) (en quantité suffisante pour la suite, on en a autant qu'on le désire). Ecrire un programme qui calcule le nombre de façon de payer une somme N passée en argument comme somme d'éléments de E (l'ordre ne compte pas).

Combinaison	Nombre de façons
1 pièce de 5	--> 1 façon
3 pièces de 1, une pièce de 2	--> 1 façon
1 pièce de 1, deux pièce de 2	--> 1 façon
5 pièces de 1	--> 1 façon
TOTAL	--> 4 façons

3.1. Dénombrer 'à la main' le nombre de façons pour différentes valeurs de votre choix

- Pour votre réflexion, il n'est pas nécessaire de faire varier E
- Chercher une présentation systématique de vos calculs (ce n'est pas le cas dans l'exemple ci-dessus, j'ai mélangé les lignes). Tant que vous n'avez pas une façon mécanique d'énumérer sur des exemples, il y a peu de chances que vous trouviez un algorithme
- il y a une et une seule façon de payer 0, c'est de ne rien donner (c'est une des conséquences du fait que E ne contient que des éléments strictement positifs)

N = 10

[illegible]

Nombre de façons	500	200	100	50	20	10	5	2	1
1							1	1	3
1							1		5
1								5	
1								4	2
1								3	4
1								2	6
1								1	8
1									10
11									

N = 7

Nombre de façons	500	200	100	50	20	10	5	2	1
1							1	1	
1							1		2
1								3	1
1								2	3
1								1	5
1									7
6									

3.2. Equation de récurrence

1. Ecrire l'équation de récurrence qui généralise votre travail dans la question précédente sur des exemples particuliers
2. Vérifier sur les cas traités à la main la validité de votre équation de récurrence.
3. Justifier votre équation (ébauche de preuve de la correction de votre algorithme)

`` n = taille de E

x = element de E allant de 0 à n

N = décomposition en facteur de E

= $e_0x_0 + e_2x_2 + \dots + e_nx_n$

f(N) = nombre de décomposition différentes en facteur de E

= ???

``

*** votre équation, justifications, ...

3.3. Ecrire une fonction récursive résolvant le problème posé

In [285]:

```
# Fonction de test retournant l'ensemble des options possibles
def pce_solutions(N, E, offset = 0):
    if N < 0 or offset >= len(E):
        return []
    elif N == 0:
        return [[0] * len(E)]
    elif offset == len(E) - 1:
        times = N // E[offset]
        if E[offset] * times == N:
            tmp = [0] * len(E)
```

```

        tmp[offset] = times
        return [tmp]
    else:
        return []
else:
    times = N // E[offset]
    options = []
    for t in range(times+1):
        reste = N - t * E[offset]
        option = pce_solutions(reste, E, offset + 1)
        for o in option:
            o[offset] += t
            options.append(o)
    return options

```

In [286]:

```

# votre code
# On suppose que l'ensemble E est trié
def pce(N, E, offset = 0):
    if N < 0 or offset >= len(E):
        return 0
    elif N == 0:
        return 1
    elif offset == len(E) - 1:
        times = N // E[offset]
        if E[offset] * times == N:
            return 1
        else:
            return 0
    else:
        times = N // E[offset]
        options = 0
        for t in range(times+1):
            reste = N - t * E[offset]
            options += pce(reste, E, offset + 1)
        return options

```

In [287]:

```

E = [500, 200, 100, 50, 20, 10, 5, 2, 1]
n = 10

```

```

print(pce(n, E))
display(pce_solutions(n, E))

```

11

```

[[0, 0, 0, 0, 0, 0, 0, 0, 10],
 [0, 0, 0, 0, 0, 0, 0, 1, 8],
 [0, 0, 0, 0, 0, 0, 0, 2, 6],
 [0, 0, 0, 0, 0, 0, 0, 3, 4],
 [0, 0, 0, 0, 0, 0, 0, 4, 2],
 [0, 0, 0, 0, 0, 0, 0, 5, 0],
 [0, 0, 0, 0, 0, 0, 1, 0, 5],
 [0, 0, 0, 0, 0, 0, 1, 1, 3],
 [0, 0, 0, 0, 0, 0, 1, 2, 1],
 [0, 0, 0, 0, 0, 0, 2, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 0]]

```

3.4. Représenter le multi-graphe des appels

In [288]:

```

def pce_graph(N, E, offset = 0, dot=Graph()):
    dot.node(str(N) + " " + str(E[offset]))
    if N < 0 or offset >= len(E):
        return 0, dot
    elif N == 0:
        return 1, dot

```

```

elif offset == len(E) - 1:
    times = N // E[offset]
    if E[offset] * times == N:
        return 1, dot
    else:
        return 0, dot
else:
    times = N // E[offset]
    options = 0
    for t in range(times+1):
        reste = N - t * E[offset]
        option, _ = pce_graph(reste, E, offset + 1, dot)
        options += option
        dot.edge(str(N) + " " + str(E[offset]), str(reste) + " " + str(E[offset+1]))
    return options, dot

```

In [289]:

```

o, g = pce_graph(20, [500, 200, 100, 50, 20, 10, 5, 2, 1], 0, Graph())
display(g)
display(o)

```



41

*** vos explications

- Les noeuds sont représentés par le N rechercher ou le reste et la valeur de la pièce.
- Les arcs représentent le lien entre les appels.
- Le nombre de feuilles est égale au nombre de solution de l'arbre

3.5. Etude de complexité empirique de la solution purement réursive

1. Représenter en fonction de N
le nombre d'appels à votre fonction
2. Déterminer par essai/erreur une fonction qui approche au mieux le tracé précédent

In [290]:

```

def pce_compte(N, E, offset = 0):
    if N <= 0 or offset >= len(E) - 1:
        return 1
    else:
        times = N // E[offset]
        nombre = 0
        for t in range(times+1):
            reste = N - t * E[offset]
            nombre += pce(reste, E, offset + 1)
        return nombre

```

In [291]:

```

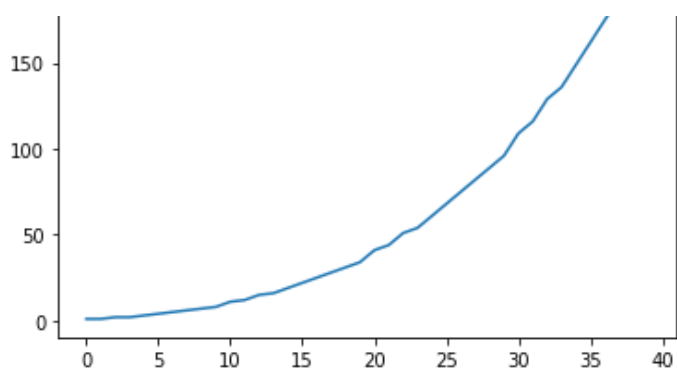
# Votre code pour la représentation graphique
tab_n = [i for i in range(40)]
E = [500, 200, 100, 50, 20, 10, 5, 2, 1]
tab_nb_op = list(map(lambda n: pce_compte(n, E), tab_n))
sns.lineplot(x=tab_n, y=tab_nb_op)

```

Out[291]:

<AxesSubplot:>





In [292]:

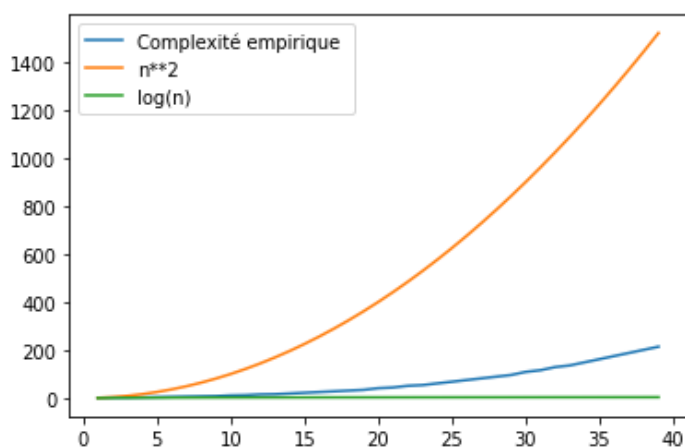
```
# votre code pour la recherche d'une fonction approchante
```

```
tab_n = [i for i in range(1,40)]
E = [500, 200, 100, 50, 20, 10, 5, 2, 1]

# Comparaison avec n**2 et log(n)
tab_nb_op = list(map(lambda n: pce_compte(n, E), tab_n))
sns.lineplot(x=tab_n, y=tab_nb_op, label="Complexité empirique ")
sns.lineplot(x=tab_n, y=list(map(lambda n: n**2, tab_n)), label="n**2")
sns.lineplot(x=tab_n, y=list(map(lambda n: np.log(n), tab_n)), label="log(n) ")
```

Out[292]:

<AxesSubplot:>



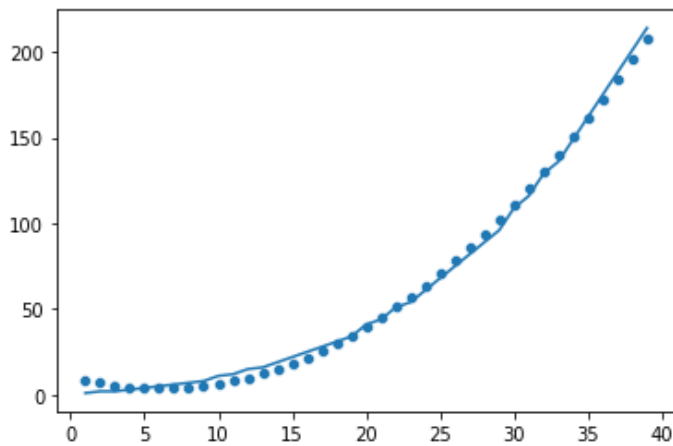
In [293]:

```
# Approximation par la méthode de régression linéaire
phi = (1+np.sqrt(5))/2
data = pd.DataFrame(tab_n, columns=["n"])
data["n²"] = data.apply(lambda ligne: ligne[0]**2, axis=1)
data["1"] = data.apply(lambda ligne: 1, axis=1)
display(data.head())

model = LinearRegression().fit(data, tab_nb_op)
tab_nb_op_approx = model.predict(data)
sns.lineplot(x=tab_n, y=tab_nb_op)
sns.scatterplot(x=tab_n, y=tab_nb_op_approx)
display(model.coef_)
```

	n	n²	1
0	1	1	1
1	2	4	1
2	3	9	1
3	4	16	1
4	5	25	1

```
array([-2.34884761,  0.189789 ,  0.        ])
```



3.6. Mémoïser la fonction récursive

In [294]:

```
# votre code
def pce_memo(N, E, offset = 0, dico = dict()):
    name = str(N) + " " + str(E[offset])
    if name not in dico:
        if N < 0 or offset >= len(E):
            dico[name] = 0
        elif N == 0:
            dico[name] = 1
        elif offset == len(E) - 1:
            times = N // E[offset]
            if E[offset] * times == N:
                dico[name] = 1
            else:
                dico[name] = 0
        else:
            times = N // E[offset]
            options = 0
            for t in range(times+1):
                reste = N - t * E[offset]
                options += pce_memo(reste, E, offset + 1, dico)
            dico[name] = options
    return dico[name]
```

In [295]:

```
d = pce_memo(20, [500, 200, 100, 50, 20, 10, 5, 2, 1])
display(d)
```

41

3.7. Représenter le graphe des appels de la fonction mémoïsée

In [296]:

```
# votre code
def pce_graph_memo(N, E, offset = 0, dot=Graph(), dico = dict()):
    name = str(N) + " " + str(E[offset])
    if name not in dico:
        if dot is not None:
            dot.node(name)
        if N < 0 or offset >= len(E):
            dico[name] = 0
        elif N == 0:
            dico[name] = 1
        elif offset == len(E) - 1:
            times = N // E[offset]
```

```

    if E[offset] * times == N:
        dico[name] = 1
    else:
        dico[name] = 0
else:
    times = N // E[offset]
    options = 0
    for t in range(times+1):
        reste = N - t * E[offset]
        option, _ = pce_graph_memo(reste, E, offset + 1, dot, dico)
        options += option
        if dot is not None:
            dot.edge(name, str(reste) + " " + str(E[offset+1]))
    dico[name] = options
return dico[name], dot

```

In [297]:

```

o, g = pce_graph_memo(20, [500, 200, 100, 50, 20, 10, 5, 2, 1], 0)
display(g)
display(o)

```



41

3.8. Résoudre le problème en appliquant les principes de la programmation dynamique

In [298]:

```

# votre code
def pce_dyna(N, E):
    n_arr = [0] * (N + 1)
    # Cas particuliers
    if N == 0 or N == min(E):
        return 1

    for i in range(min(N, min(E))):
        n_arr[i] = 1
    for coin in E:
        for n in range(len(n_arr)):
            if n >= coin:
                n_arr[n] += n_arr[n - coin]

    return n_arr[N]

```

In [299]:

```
display(pce_dyna(100000, [500, 200, 100, 50, 20, 10, 5, 2, 1]))
```

256945515490162908137251

*** vos explications...

On initialise un tableau contenant chaque valeur de 0 à N.

En parcourant chaque valeur de ce tableau pour chaque pièces on ajoute progressivement le nombre de cas précédents au nombre de cas actuel.

$N(9) : N(4) + 1$ pièce de 5

$N(9) : N(7) + 1$ pièce de 2

$N(9) : N(8) + 1$ pièce de 1

$\Rightarrow N(9) = N(4) + N(7) + N(8)$

Donc $N(i) = \text{Somme}[e \text{ appartient à } E] (N(i-e))$

3.9. Etude de complexité empirique de la solution par programmation

dynamique

1. Représenter en fonction de N
le nombre d'appels à votre fonction
2. Déterminer par essai/erreur une fonction qui approche au mieux le tracé précédent

- On appelle une seule fois notre fonction $O(n*m)$, m a une taille fixe ou négligeable (taille de E), complexité amortie en $O(n)$

In [300]:

```
# votre code
def c_dyna(N, E):
    return N * len(E)

print("Complexité Algo :", c_dyna(50, [5000, 200, 500, 300, 200, 131, 100, 12, ]), "opérations")
```

Complexité Algo : 400 opérations

*** vos conclusions...

Le code avec la programmation dynamique est vraiment plus rapide que le code mémo et le code récursif.

In []: