Chaines_Addition

April 3, 2023

1 Chaînes d'addition

Marc Lorenzi

3 avril 2023

[1]:
```
import matplotlib.pyplot as plt
from math import log, floor
```

[2]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

1.1 1. Introduction

1.1.1 1.1 Calculs de puissances ...

Soit x un élément d'un monoïde $(\mathcal{M}, \times)$. Supposons que je veuille calculer x^{23}.

- Évidemment, je peux écrire $x^{23} = x \times ... \times x$, et faire 22 multiplications.
- Un peu de réflexion me permet d'écrire $x^{23} = x \times x^2 \times x^4 \times x^{16}$. Mais $x^4 = (x^2)^2$ et $x^{16} = ((x^4)^2)^2$. Cette fois-ci, 7 mutiplications sont suffisantes (une élévation au carré compte pour une multiplication).
- Puis-je faire mieux ? Oui, car $x^{23} = x^{10} \times x^{13}$, $x^{13} = x^3 \times x^{10}$, $x^{10} = x^5 \times x^5$, $x^5 = x^2 \times x^3$, $x^3 = x \times x^2$, $x^2 = x \times x$. Ainsi, 6 multiplications suffisent.
- Puis-je faire mieux ? La réponse est non mais il faudra attendre la fin de ce notebook pour en être sûrs.

Reprenons le troisième point ci-dessus. Si on peut calculer x^{23} en faisant 6 multiplications, c'est parce que $1 + 1 = 2$, $1 + 2 = 3$, $2 + 3 = 5$, $5 + 5 = 10$, $3 + 10 = 13$ et $10 + 13 = 23$. Dit autrement, la suite $(1, 2, 3, 5, 10, 13, 23)$ possède une propriété remarquable. Chaque nombre de la suite (sauf 1, bien entendu) est la somme de deux des nombres qui le précèdent. Minimiser le nombre de multiplications dans un calcul de puissance c'est tout simplement trouver une telle suite aussi courte que possible.

Voilà notre objectif fixé : Trouver des suites d'entiers ayant cette propriété et qui soient aussi courtes que possible.

1.1.2 1.2 C'est quoi une chaîne d'addition ?

Définition : Soit $n \in \mathbb{N}^*$. Une chaîne d'addition (en abrégé : chaîne) pour l'entier n est une suite finie d'entiers $a = (a_0, a_1, ..., a_r)$ où $r \in \mathbb{N}$, telle que

- $a_0 = 1$
- $a_r = n$
- $a_0 \leq a_1 \leq ... \leq a_r$
- Pour tout i tel que $0 < i \leq r$ il existe j et k tels que $0 \leq j \leq k < i$ et $a_i = a_j + a_k$

L'entier r est la **longueur** de la chaîne a. On le notera au besoin $L(a)$. Remarquer que $L(a)$ vaut 1 de moins que ce que l'on appelle usuellement la longueur d'une suite.

Le plus petit entier r tel qu'il existe une chaîne d'addition pour n de longueur r est noté $\ell(n)$. Une chaîne de longueur $\ell(n)$ est appelée une **chaîne d'addition optimale**.

Dans tout le notebook, n désigne un entier naturel non nul.

[3]:
```
def L(a): return len(a) - 1
```

[4]:
```
L([1, 2, 3, 4, 5])
```

[4]: 4

1.1.3 1.3 Une (très mauvaise) majoration de $\ell(n)$

Il faut bien commencer par quelque part ! Posons, pour $i \in [|0, n-1|]$, $a_i = i+1$. Nous avons là une chaîne d'addition pour n de longueur $n-1$. En effet, pour tout $1 \leq i \leq n-1$, on a $i = (i-1)+1$, c'est à dire $a_i = a_{i-1} + a_0$. En fait cette chaîne est celle qui est associée au calcul naïf d'une puissance : $x^n = x \times x \times ... \times x$.

On en déduit la majoration : $\ell(n) \leq n - 1$. Nous allons bientôt améliorer considérablement ce majorant.

1.1.4 1.4 Une (pas mauvaise) minoration de $\ell(n)$

Nous allons donner un minorant de $\ell(n)$ (meilleur que 0 :-)). Avant, définissons la fonction

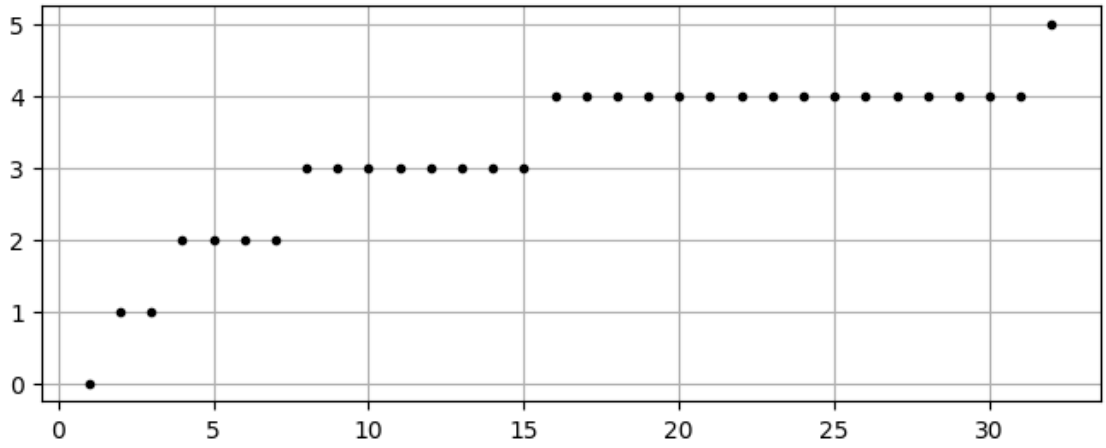
$$\lambda : n \longmapsto \lfloor \lg n \rfloor$$

où lg désigne le logarithme en base 2. La fonction Python `lg` fait le travail, sans utiliser ni flottants ni partie entière ni logarithme.

[5]:
```
def lg(n):
    p = 1
    c = 0
    while p <= n:
        p = 2 * p
```

```
        c = c + 1
    return c - 1
```

[6]:
```
rg = range(1, 33)
plt.plot(rg, [lg(k) for k in rg], 'ok', ms=3)
plt.grid()
```



1.1.5 1.5 Quelques propriétés

Proposition : On a, pour $1 \leq m \leq n$, $\lambda(m+n) \leq \lambda(2n) = \lambda(n) + 1$.

Démonstration : exercice facile.

Proposition : Soit $(a_0 = 1, \ldots, a_r = n)$ une chaîne d'addition pour n. Pour $1 \leq i < r$, on a $\lambda(a_{i+1}) = \lambda(a_i)$ ou $\lambda(a_{i+1}) = \lambda(a_i) + 1$.

Démonstration : $a_i \leq a_{i+1}$ donc $\lambda(a_i) \leq \lambda(a_{i+1})$, car λ est croissante. Par ailleurs, $a_{i+1} = a_j + a_k$ où $1 \leq j \leq k \leq i$. De là,

$$\lambda(a_{i+1}) = \lambda(a_j + a_k) \leq \lambda(a_k) + 1 \leq \lambda(a_i) + 1$$

Proposition : Soit $(a_0 = 1, \ldots, a_r = n)$ une chaîne d'addition pour n. Pour $1 \leq i \leq r$, on a $\lambda(a_i) \leq i$.

Démonstration : Conséquence facile de la proposition précédente. En passant d'une terme de la chaîne au terme suivant, λ augmente de 0 ou de 1.

Proposition : Pour tout entier $n \geq 1$, $\ell(n) \geq \lambda(n)$.

Démonstration : Soit $(a_0 = 1, \ldots, a_r = n)$ une chaîne d'addition pour n. On a d'après la proposition précédente $\lambda(a_r) \leq r$. Mais $a_r = n$, donc $\lambda(n) \leq r$. Il suffit maintenant de choisir une chaîne optimale. Pour cette chaîne, $r = \ell(n)$.

1.1.6 1.6 Petits sauts, grands sauts

La fonction λ ne peut augmenter que de 1 à la fois à l'intérieur d'une chaîne d'addition. Comme on démarre de $0 = \lambda(1)$ et que la fin de la chaîne d'addition est signalée par $\lambda(n)$, on a intérêt à ce que λ augmente aussi vite que possible lorsqu'on parcourt les termes de la suite. Lorsque λ reste constante en passant d'un certain terme de la suite au suivant, on dit que l'on a affaire à un **petit saut**. Sinon, on a un **grand saut**. Le nombre de grands sauts est exactement égal à $\lambda(n)$. En revanche, le nombre de petits sauts peut varier d'une chaîne d'addition à une autre. Bref :

Une chaîne d'addition optimale, c'est une chaîne d'addition ayant le moins possible de petits sauts.

Cas très particulier : Si $n = 2^p$ est une puissance de 2, alors $(1, 2, 2^2, \dots, 2^p)$ est une chaîne d'addition pour n de longueur $p = \lambda(n)$. Elle ne comporte aucun petit saut, sa longueur est $\lambda(n)$, elle est donc optimale. Ainsi, dans ce cas, $\ell(n) = p = \lambda(n)$.

1.2 2. La stratégie binaire

Soit $(B_n)_{n\geq 1}$ la suite de suites d'entiers (non, je ne bégaie pas) définie par :

- $B_0 = (1)$, la suite ayant un seul terme, égal à 1.
- $\forall p > 0, B_{2p} = B_p \oplus (2p)$
- $\forall p \geq 0, B_{2p+1} = B_p \oplus (2p, 2p+1)$

Où $\oplus$ désigne l'opération de concaténation des suites : $(a_0, \dots, a_r) \oplus (b_0, \dots, b_s) = (a_0, \dots, a_r, b_0, \dots, b_s)$.

Proposition : B_n est une chaîne d'addition pour l'entier n.

Démonstration : Faire une récurrence forte sur n. Il suffit de remarquer que $2p = p + p$.

Remarque : B_n n'est autre que la suite des puissances calculées lors de l'algorithme d'exponentiation rapide (ou plutôt de l'une des implémentations de cet algorithme, car il y a plusieurs façons de l'écrire).

[7]:
```
def chaine_binaire(n):
    if n == 1: return [1]
    elif n % 2 == 0: return chaine_binaire(n // 2) + [n]
    else: return chaine_binaire(n - 1) + [n]
```

[8]:
```
n = 23
s = chaine_binaire(n)
print(s)
print(L(s) - 1, lg(n))
```

```
[1, 2, 4, 5, 10, 11, 22, 23]
6 4
```

Essayez d'autres valeurs de n. La stratégie binaire renvoie bien une chaîne d'addition, mais nous allons voir plus loin que cette chaîne n'est pas toujours optimale. Voilà pourquoi nous parlons de stratégie ! Une stratégie peut échouer :-) …

[9]:
```
def longueur_binaire(n):
    return len(chaine_binaire(n)) - 1
```

[10]:
```
print([longueur_binaire(n) for n in range(1, 100)])
```

```
[0, 1, 2, 2, 3, 3, 4, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
7, 6, 7, 7, 8, 5, 6, 6, 7, 6, 7, 7, 8, 6, 7, 7, 8, 7, 8, 8, 9, 6, 7, 7, 8, 7, 8,
8, 9, 7, 8, 8, 9, 8, 9, 9, 10, 6, 7, 7, 8, 7, 8, 8, 9, 7, 8, 8, 9, 8, 9, 9, 10,
7, 8, 8, 9, 8, 9, 9, 10, 8, 9, 9, 10, 9, 10, 10, 11, 7, 8, 8, 9]
```

Proposition : $L(B_n) \leq 2\lambda(n)$.

Démonstration : On fait une récurrence forte sur n.

- On a $L(B_1) = 0 = 2\lambda(1)$.
- Soit maintenant $n > 0$. Supposons que pour tout entier $1 \leq p < n$, $L(B_p) \leq \lambda(p)$. On a alors $L(B_n) \leq L(B_p) + 2 \leq 2\lambda(p) + 2 \leq 2\lambda(2p) \leq 2\lambda(n)$.

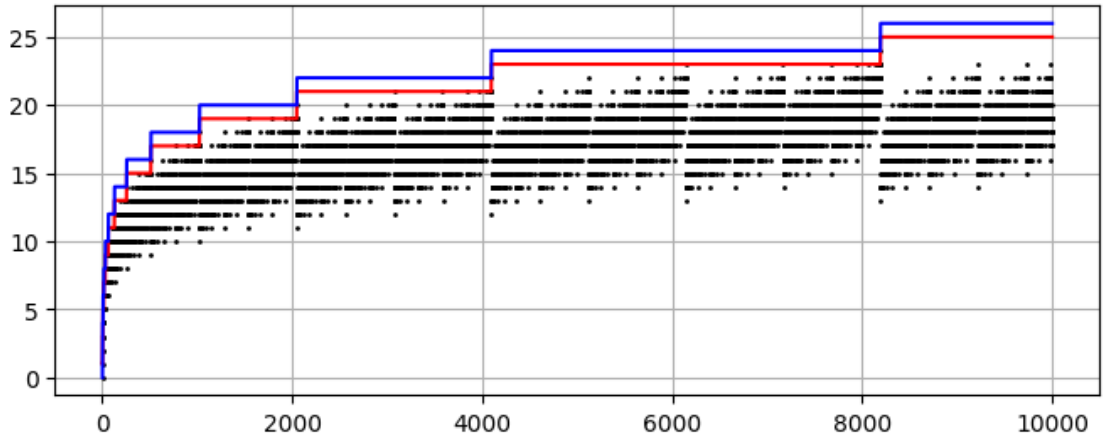
Corollaire : $\lambda(n) \leq \ell(n) \leq 2\lambda(n)$.

Démonstration : La première inégalité a déjà été prouvée. La seconde provient de ce que nous venons d'exhiber une chaîne d'addition pour n de longueur inférieure ou égale à $2\lambda(n)$.

Exercice : Montrer l'inégalité légèrement meilleure : $L(B_n) \leq 2\lambda(n+1) - 1$. On a donc $\ell(n) \leq 2\lambda(n+1) - 1$. Ceci nous sera utile à la toute fin de ce notebook. Pour les paresseux, une preuve analogue est donnée dans le notebook sur l'exponentiation rapide.

Ci-dessous, en rouge le majorant de l'exercice et en bleu le majorant de la proposition. Les points noirs sont les valeurs prises par `longueur_binaire`.

[11]:
```
rg = range(1, 10001)
s1 = [longueur_binaire(n) for n in rg]
s2 = [2 * lg(n + 1) - 1 for n in rg]
s3 = [2 * lg(n) for n in rg]
plt.plot(rg, s1, 'ok', markersize=1)
plt.plot(rg, s2, 'r')
plt.plot(rg, s3, 'b')
plt.grid()
```



1.3 3. La stratégie des facteurs

Tentons une toute autre stratégie : si on sait factoriser n sous la forme $n = pq$, alors on peut déduire une chaîne d'addition pour n si on connaît une chaîne pour p et une chaîne pour q.

Proposition : Soient $p, q \geq 1$. Soient $a = (a_0, \dots, a_r)$ une chaîne pour p et $b = (b_0, \dots, b_s)$ une chaîne pour q. Alors $c = (a_0, \dots, a_r, pb_1, \dots, pb_s)$ est une chaîne pour pq.

Démonstration : La longueur de c est $L(c) = r + s$. Soit $1 \leq i \leq r + s$.

- Si $i \leq r$ il existe $0 \leq j \leq k < i$ tels que $c_i = a_i = a_j + a_k = c_j + c_k$.
- Si $i > r$, $c_i = pb_{i-r}$. Il existe donc $0 \leq j \leq k < i$ tels que $c_i = pb_{i-r} = p(b_j + b_k) = c_{j+r} + c_{k+r}$. Il reste à constater que $0 \leq j + r \leq k + r \leq i$.

Ceci suggère une stratégie pour déterminer une chaîne F_n (F pour « facteurs ») d'additions pour n :

- $F_1 = (1)$.
- Si n est premier, $F_n = F_{n-1} \oplus (n)$.
- Sinon, soit p le plus petit diviseur de n. Soit $q = \frac{n}{p}$. Soient $(a_0, \dots, a_r) = F_p$ et $(b_0, \dots, b_s) = F_q$. Alors, $F_n = (a_0, \dots, a_r, pb_1, \dots, pb_s)$.

La fonction `plus_petit_facteur` calcule le plus petit facteur premier de n.

[12]:
```
def plus_petit_facteur(n):
    p = 2
    while p * p <= n and n % p != 0: p += 1
    if p * p > n: return n
    else: return p
```

[13]:
```
plus_petit_facteur(91)
```

[13]: 7

Voici la fonction `chaine_facteurs` qui calcule une chaîne d'addition pour n par la stratégie des facteurs.

[14]:
```
def chaine_facteurs(n):
    if n == 1: return [1]
    else:
        p = plus_petit_facteur(n)
        if p != n:
            s1 = chaine_facteurs(p)
            s2 = chaine_facteurs(n // p)
            return s1 + [p * x for x in s2[1:]]

        else:
            return chaine_facteurs(n - 1) + [n]
```

[15]:
```
n = 30
print(chaine_facteurs(n))
print(chaine_binaire(n))
```

```
[1, 2, 4, 6, 12, 24, 30]
[1, 2, 3, 6, 7, 14, 15, 30]
```

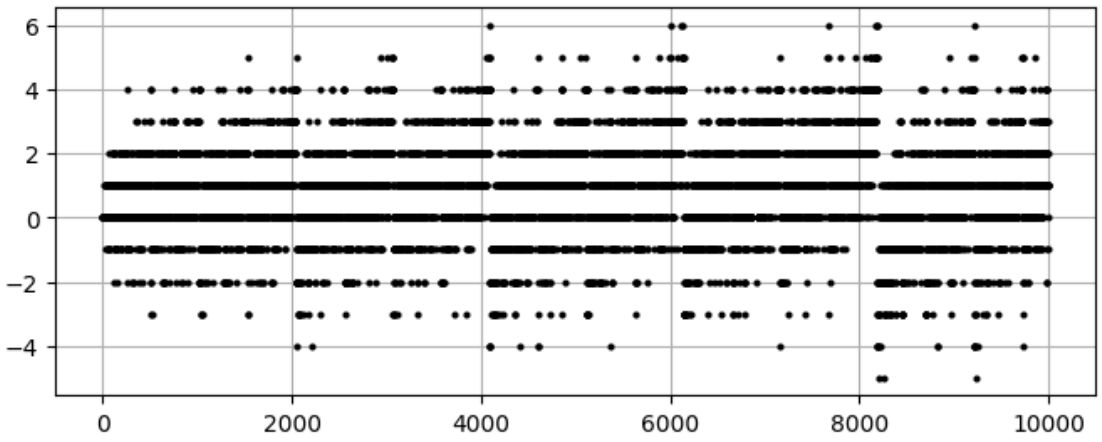
[16]:
```
def longueur_facteurs(n):
    return len(chaine_facteurs(n))- 1
```

[17]:
```
print([longueur_facteurs(n) for n in range(1, 100)])
```

```
[0, 1, 2, 2, 3, 3, 4, 3, 4, 4, 5, 4, 5, 5, 5, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
6, 6, 7, 6, 7, 5, 7, 6, 7, 6, 7, 7, 7, 6, 7, 7, 8, 7, 7, 8, 9, 6, 8, 7, 7, 7, 8,
7, 8, 7, 8, 8, 9, 7, 8, 8, 8, 6, 8, 8, 9, 7, 9, 8, 9, 7, 8, 8, 8, 8, 9, 8, 9, 7,
8, 8, 9, 8, 8, 9, 9, 8, 9, 8, 9, 9, 9, 10, 9, 7, 8, 9, 9]
```

La stratégie des facteurs est elle meilleure que la stratégie binaire ? Comparons les longueurs de chaînes obtenues pour les entiers inférieurs à 10000.

[18]:
```
s = [longueur_binaire(n)-longueur_facteurs(n) for n in range(1, 10000)]
plt.plot(s, 'ok', markersize=2)
plt.grid()
```



Bilan : Aucune des deux stratégies n'est tout le temps meilleure que l'autre. Alors ? La meilleure solution est d'appliquer les deux stratégies et de garder le meilleur résultat ! Voici le score :

[19]:
```
len([n for n in range(1, 10000) if longueur_binaire(n) > longueur_facteurs(n)])
```

[19]: 5750

1.4 4. La stratégie de l'arbre des puissances

1.4.1 4.1 L'arbre des puissances

Parmi les stratégies permettant le calcul de chaînes d'addition, la méthode de l'arbre des puissances donne de bons résultats. Cette méthode est décrite par Donald Knuth dans *The Art of Computer Programming, Volume 2, pp 464, 481.* On construit un arbre de la façon suivante :

- À la racine (niveau 0) on place le nombre 1.
- Ayant construit tous les niveaux de l'arbre jusqu'au niveau k, on construit de gauche à droite les fils des noeuds de ce niveau. Si x est un noeud du niveau k et $a_0 = 1, a_1, \dots, a_r = x$ sont les noeuds situés sur le chemin qui va de la racine de l'arbre à x, les fils de x, de gauche à droite, sont $x + a_0, x + a_1, \dots, x + a_r$, ou plus exactement ceux parmi ces entiers qui ne sont pas déjà dans l'arbre.

Voici par exemple les niveaux 0, 1, 2 et 3 :

- 1 n'a qu'un fils : $1 + 1 = 2$.
- 2 a deux fils : $2 + 1 = 3$ et $2 + 2 = 4$.
- 3 a deux fils : $3 + 2 = 5, 3 + 3 = 6$. L'entier $3 + 1 = 4$ est déjà dans l'arbre !
- 4 n'a qu'un fils, $4 + 4 = 8$, car les entiers $4 + 1$ et $4 + 2$ sont déjà dans l'arbre.
- etc.

Et la chaîne d'addition de n, dans tout ça ? Eh bien c'est précisément la liste des noeuds qui sont sur le chemin de la racine de l'arbre au noeud n. On laisse le soin au lecteur de faire une récurrence sur k pour montrer qu'il s'agit bien d'une chaîne d'addition.

1.4.2 4.2 La fonction Python

On représente l'arbre en Python par un dictionnaire `p`. Si x est un noeud de l'arbre, $p[x]$ est son père. On pose arbitrairement $p[1] = 1$. Avec les valeurs données ci-dssus en exemple, on a :

[20]:
```
peres = {1:1, 2:1, 3:2, 4:2, 5:3, 6:3, 8:4}
```

La fonction `branche` prend en paramètres un noeux x de l'arbre modélisé par le dictionnaire p. Elle renvoie la liste $[a_0 = 1, \dots, a_r = x]$ des ancêtres de x.

[21]:
```
def branche(x, p):
    s = []
    while p[x] != x:
        s.append(x)
        x = p[x]
    s.append(1)
    s.reverse()
    return s
```

[22]:
```
branche(6, peres)
```

[22]:
```
[1, 2, 3, 6]
```

La fonction `arbre_puissances` construit l'arbre des puissances jusqu'au niveau k. Elle maintient un ensemble `visites` qui contient à chaque instant les entiers déjà présents dans l'arbre, et elle construit l'arbre niveau par niveau.

[23]:
```
def arbre_puissances(k):
    peres = {1:1}
    level = [1]
    visites = set([1])
    for j in range(k):
        level1 = []
        for x in level:
            for y in branche(x, peres):
                z = x + y
                if not z in visites:
                    visites.add(z)
                    peres[z] = x
                    level1.append(z)
        level = level1
    return peres
```

[24]:
```
arbre_puissances(3)
```

[24]:
```
{1: 1, 2: 1, 3: 2, 4: 2, 5: 3, 6: 3, 8: 4}
```

[25]:
```
print(arbre_puissances(5))
```

```
{1: 1, 2: 1, 3: 2, 4: 2, 5: 3, 6: 3, 8: 4, 7: 5, 10: 5, 9: 6, 12: 6, 16: 8, 14:
7, 11: 10, 13: 10, 15: 10, 20: 10, 18: 9, 24: 12, 17: 16, 32: 16}
```

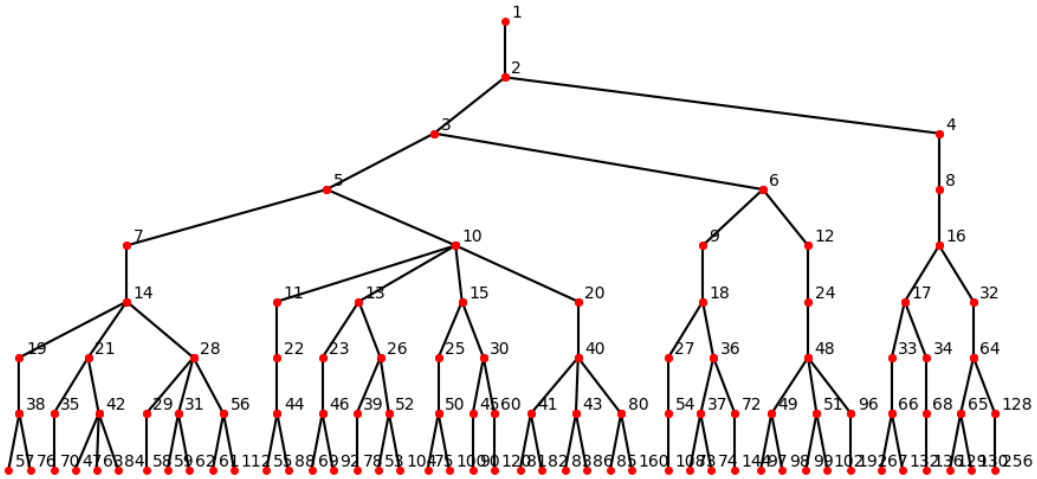
1.4.3 4.3 Dessiner l'arbre des puissances

Une représentation graphique de l'arbre des puissances peut être intéressante. La fonction `dessin_puissances` se trouve dans le module `dessin_arbre`. Elle prend en paramètre un arbre des puissances et affiche joliment cet arbre. Soyons raisonnables sur le nombre de niveaux de l'arbre !

[26]:
```
from dessin_arbre import dessin_puissances
```

[27]:
```
plt.rcParams['figure.figsize'] = (12, 6)
```

[28]:
```
dessin_puissances(arbre_puissances(8))
```



[29]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

1.4.4 4.4 Quelques compléments

Construisons une fois pour toutes l'arbre des puissances jusqu'au niveau 22.

[30]:
```
ap = arbre_puissances(22)
```

Notre arbre contient tous les entiers jusqu'à 150000, sauf 5 d'entre-eux.

[31]:
```
[x for x in range(1, 150000) if x not in ap]
```

[31]:
```
[138959, 140159, 143231, 148223, 148989]
```

Si l'on s'intéresse uniquement à une chaîne d'addition pour un certain entier n, il suffit d'exécuter l'algorithme de construction de l'arbre des puissances jusqu'au moment où n est placé dans l'arbre, puis de renvoyer la liste des ancêtres de n. La fonction `chaine_puissances` fait le travail.

[32]:
```
def chaine_puissances(n):
    peres = {1:1}
    level = [1]
    visites = set([1])
    while True:
        if n in peres:
            return branche(n, peres)
        level1 = []
        for x in level:
            for y in branche(x, peres):
                z = x + y
                if not z in visites:
                    visites.add(z)
                    peres[z] = x
                    level1.append(z)
        level = level1
```

[33]:
```
print(chaine_puissances(123456))
```

```
[1, 2, 3, 5, 10, 20, 40, 80, 160, 320, 640, 643, 1286, 1929, 3858, 7716, 15432,
30864, 61728, 123456]
```

Enfin, la longueur de la chaine d'addition pour l'entier n est calculée par la fonction `longueur_puissances`, qui prend en paramètre, outre l'entier n, un arbre des puissances déjà calculé et supposé être assez gros pour contenir n.

[34]:
```
def longueur_puissances(n, c):
    return len(branche(n, c)) - 1
```

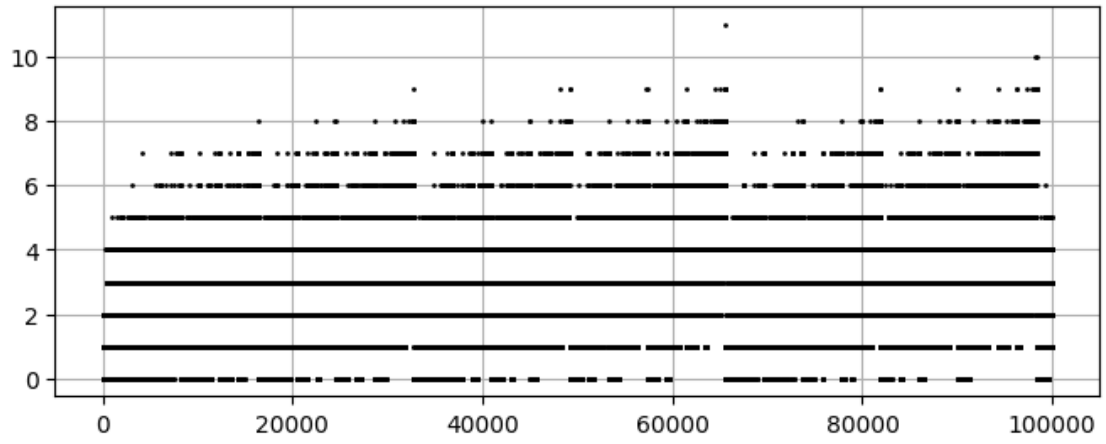
[35]:
```
print([longueur_puissances(n, ap) for n in range(1, 100)])
```

```
[0, 1, 2, 2, 3, 3, 4, 3, 4, 4, 5, 4, 5, 5, 5, 4, 5, 5, 6, 5, 6, 6, 6, 5, 6, 6,
6, 6, 7, 6, 7, 5, 6, 6, 7, 6, 7, 7, 7, 6, 7, 7, 7, 7, 7, 7, 8, 6, 7, 7, 7, 7, 8,
7, 8, 7, 8, 8, 8, 7, 8, 8, 8, 6, 7, 7, 8, 7, 8, 8, 9, 7, 8, 8, 8, 8, 9, 8, 9, 7,
8, 8, 8, 8, 8, 8, 9, 8, 9, 8, 9, 8, 9, 9, 9, 7, 8, 8, 8]
```

1.4.5 4.5 Comparaison avec les deux autres stratégies

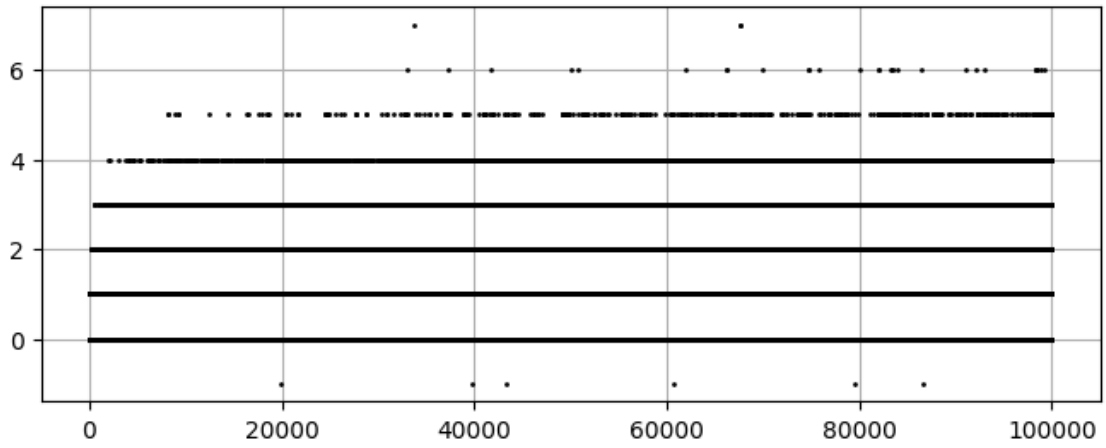
4.3.1 Stratégie binaire La stratégie de l'arbre des puissances est **toujours** meilleure que la stratégie binaire. Nous ne le montrerons pas ici mais en voici une illustration pour $1 \leq n \leq 10^5$.

[36]:
```
s = [longueur_binaire(n) - longueur_puissances(n, ap) for n in range(1, 100000)]
plt.plot(s, 'ok', markersize=1)
plt.grid()
```



4.3.2 Stratégie des facteurs La stratégie de l'arbre des puissances est **très souvent** meilleure que la stratégie des facteurs. En voici une illustration pour $1 \leq n \leq 10^5$.

[37]:
```
s = [longueur_facteurs(n) - longueur_puissances(n, ap) for n in range(1, 100001)]
plt.plot(s, 'ok', markersize=1)
plt.grid()
```



On voit qu'il n'y a que 6 valeurs de n pour lesquelles la stratégie des facteurs donne un meilleur résultat.

[38]:
```
[n for n in range(1, 100001) if longueur_facteurs(n) < longueur_puissances(n,␣
↪ap)]
```

[38]:
```
[19879, 39758, 43277, 60749, 79516, 86554]
```

Les deux stratégies donnent le même résultat dans 11% des cas.

[39]:
```
len([n for n in range(1, 100001) if longueur_facteurs(n) ==␣
↪longueur_puissances(n, ap)])
```

[39]:
```
11191
```

Aparté : Donald Knuth trouve comme moi, page 464 de The Art Of Computer Programming. Enfin, c'est plutôt moi qui trouve comme lui :-).

Et donc, la stratégie de l'arbre des puissances est meilleure dans environ 89% des cas.

[40]:
```
100000 - 6 - 11191
```

[40]:
```
88803
```

1.5 5. Trouver une chaîne d'addition optimale

Nous y voilà. Comment obtenir une chaîne d'addition optimale pour l'entier n ? On ne connaît aucun algorithme efficace pour résoudre ce problème. Des algorithmes très sophistiqués permettent cependant de résoudre le problème pour des valeurs de n allant jusqu'à quelques milliards. Nous alons être beaucoup plus modestes …

Ce notebook est déjà bien long. Nous allons nous contenter de résoudre ce problème pour des petites valeurs de n, disons inférieures à 50. La fonction `chaines` prend un entier n en paramètre et renvoie un dictionnaire contenant, pour $1 \leq m \leq n$, "toutes" les chaînes d'addition pour l'entier m. Notez les guillemets autour du mot "toutes", en fait toutes les chaînes

- suffisamment longues pour ne pas rater une chaîne optimale pour l'entier m.
- suffisamment courtes pour que le calcul se fasse en un temps humainement acceptable.

Tout est dans le "suffisamment". Une amélioration ne serait-ce que de 1 dans la valeur du majorant choisi pour les longueurs peut diviser le temps de calcul par 2 … bienvenue dans le monde des fonctions de complexité exponentielle !

Voici tout d'abord une fonction `acceptable`. Elle prend en paramètres un entier n et une chaîne d'addition s pour un entier $m < n$. La fonction teste si n est la somme de deux éléments de s. Dans ce cas, $s \oplus (n)$ est une chaîne d'addition pour n.

[41]:
```
def acceptable(s, n):
    r = len(s)
    for i in range(r):
        if n - s[i] in s: return True
    return False
```

Voici le fameux majorant de $\ell(n)$ démontré en exercice.

[42]:
```
def majo(n): return 2 * lg(n + 1) - 1
```

Voici la fonction `chaînes`.

La fonction crée un dictionnaire `cs` et initialise `cs[1]`. Puis, pour tous les entiers m, $2 \leq m \leq n$, elle calcule les chaînes d'addition pour m : ces chaînes sont de la forme $s \oplus (m)$ où s est une chaîne d'addition pour un entier k, $\frac{m}{2} \leq k < m$, s est acceptable pour l'entier m, et la longueur de s n'est pas trop grande.

Cette fonction est terriblement synthétique, mais aussi terriblement inefficace. On peut faire infiniment mieux.

[43]:
```
def chaines(n):
    cs = {1:[[1]]}
    for m in range(2, n + 1):
        lmax = majo(min(2 * m, n))
        lmax1 = max(majo(m), lmax - 1)
        cs[m] = [s + [m] for k in range(m // 2, m)  for s in cs[k] if␣
 ↪acceptable(s, m) and L(s) < lmax1]
    return cs
```

[44]:
```
chaines(7)
```

[44]:
```
{1: [[1]],
 2: [[1, 2]],
 3: [[1, 2, 3]],
 4: [[1, 2, 4], [1, 2, 3, 4]],
 5: [[1, 2, 3, 5], [1, 2, 4, 5], [1, 2, 3, 4, 5]],
 6: [[1, 2, 3, 6],
  [1, 2, 4, 6],
  [1, 2, 3, 4, 6],
  [1, 2, 3, 5, 6],
  [1, 2, 4, 5, 6]],
 7: [[1, 2, 3, 4, 7],
  [1, 2, 3, 5, 7],
  [1, 2, 4, 5, 7],
  [1, 2, 3, 4, 5, 7],
  [1, 2, 3, 6, 7],
  [1, 2, 4, 6, 7],
  [1, 2, 3, 4, 6, 7],
  [1, 2, 3, 5, 6, 7],
  [1, 2, 4, 5, 6, 7]]}
```

La fonction `chaines_optimales` appelle la fonction `chaines` et ne conserve pour chaque entier $1 \leq k \leq n$ qu'une chaîne d'addition de longueur minimale. Cette chaîne est exactement de longueur $\ell(k)$.

[45]:
```
def chaines_optimales(n):
    memo = chaines(n)
    opt = {}
    for k in range(1, n + 1):
        s0 = memo[k][0]
        for s in memo[k]:
            if len(s) < len(s0): s0 = s
            opt[k] = s0
    return opt
```

[46]:
```
chaines_optimales(20)
```

[46]:
```
{1: [1],
 2: [1, 2],
 3: [1, 2, 3],
 4: [1, 2, 4],
 5: [1, 2, 3, 5],
 6: [1, 2, 3, 6],
 7: [1, 2, 3, 4, 7],
 8: [1, 2, 4, 8],
 9: [1, 2, 4, 5, 9],
 10: [1, 2, 3, 5, 10],
 11: [1, 2, 3, 5, 6, 11],
 12: [1, 2, 3, 6, 12],
 13: [1, 2, 3, 6, 7, 13],
 14: [1, 2, 3, 4, 7, 14],
 15: [1, 2, 3, 6, 9, 15],
 16: [1, 2, 4, 8, 16],
 17: [1, 2, 4, 8, 9, 17],
 18: [1, 2, 4, 5, 9, 18],
 19: [1, 2, 4, 5, 9, 10, 19],
 20: [1, 2, 3, 5, 10, 20]}
```

Et enfin, LA fonction attendue avec impatience ! `longueurs_opts` renvoie la liste des $\ell(k)$ pour $1 \leq k \leq n$. Soyons réaliste, ne dépassons pas quelques dizaines pour la valeur de n. Même $n = 100$ est inaccessible avec cette fonction.

[47]:
```
def longueurs_opts(n):
    ls = {}
    opt = chaines_optimales(n)
    for k in opt:
        ls[k] = len(opt[k]) - 1
    return ls
```

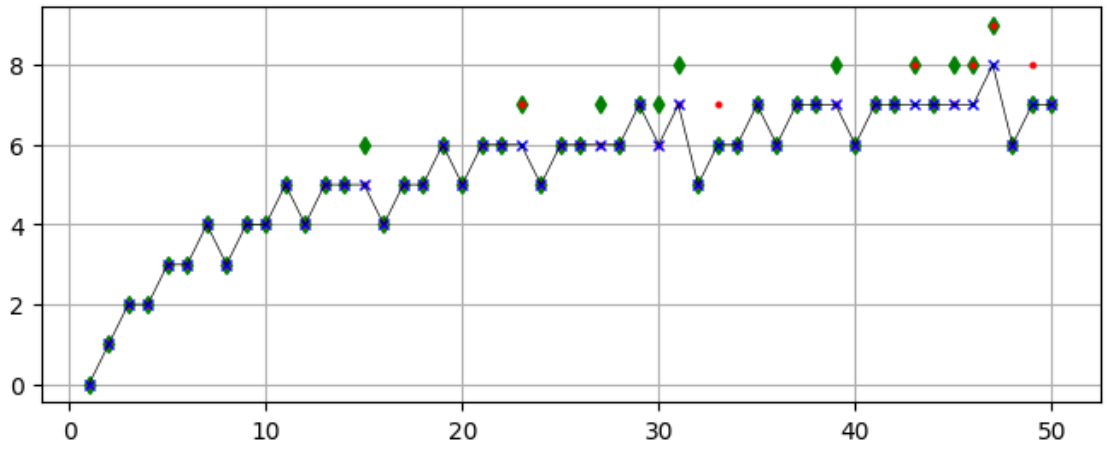
[48]:
```
print(longueurs_opts(36))
```

```
{1: 0, 2: 1, 3: 2, 4: 2, 5: 3, 6: 3, 7: 4, 8: 3, 9: 4, 10: 4, 11: 5, 12: 4, 13:
5, 14: 5, 15: 5, 16: 4, 17: 5, 18: 5, 19: 6, 20: 5, 21: 6, 22: 6, 23: 6, 24: 5,
```

```
25: 6, 26: 6, 27: 6, 28: 6, 29: 7, 30: 6, 31: 7, 32: 5, 33: 6, 34: 6, 35: 7, 36:
6}
```

Traçons ! Soyez patients, je prend $n = 50$.

[49]:
```
n = 50
rg = range(1, n + 1)
s1 = [longueur_binaire(n) for n in rg]
s2 = [longueur_facteurs(n) for n in rg]
s3 = [longueur_puissances(n, ap) for n in rg]
opt = longueurs_opts(n)
s4 = [opt[k] for k in rg]
plt.plot(rg, s1, 'dg', markersize=5)
plt.plot(rg, s2, 'or', markersize=2)
plt.plot(rg, s3, 'xb', markersize=5)
plt.plot(rg, s4, 'k', lw=0.5)
plt.grid()
```



- Les diamants verts représentent les longueurs obtenues par la stratégie binaire.
- Les points rouges représentent les longueurs obtenues par la stratégie des facteurs.
- Les croix bleues représentent les longueurs obtenues par la stratégie de l'arbre des puissances.
- La ligne brisée noire est la "ligne optimale".

On constate que tous les points bleus sont sont sur la ligne noire. La stratégie de l'arbre des puissances serait-elle optimale ? On peut en rêver, mais cela restera un rêve. En effet, pour $n = 77$, la stratégie de l'arbre ne donne pas la longueur optimale :-(. Hélas, notre fonction `longueurs_opts` ne nous permet pas d'aller si loin dans les valeurs de n en un temps raisonnable.

Conclusion : Trouver une chaîne additive optimale est un problème **vraiment difficile**.

1.6 6 Quelques compléments

1.6.1 6.1 Un équivalent de $\ell(n)$

Le temps d'exécution de notre fonction calculant $\ell(n)$ dépend crucialempent d'une majoration de $\ell(n)$. Peut-on faire mieux que $2\lambda(n+1)-1$? Le problème est que les majorations ne sont en général vraies que pour tout n assez grand. Citons la majoration de Brauer : pour tout n assez grand,

$$\ell(n) < \lg n(1 + \frac{1}{\ln\ln n} + \frac{2\ln 2}{(\ln n)^{1-\ln 2}})$$

La parenthèse tend vers 1 lorsque n tend vers l'infini ! On en déduit que

$$\ell(n) \sim \lg n$$

Dit autrement, lorsque n tend vers l'infini, $\ell(n)$ est équivalent à son minorant $\lambda(n)$.

1.6.2 6.2 La conjecture de Scholz-Brauer

Quels sont les exposants les plus « défavorables » pour l'algorithme d'exponentiation rapide ? Ce sont les entiers de la forme $n = 2^p - 1$, pour lesquels le nombre de multiplications à effectuer est $2p - 2$.

[50]:
```
[longueur_binaire(2 ** p - 1) for p in range(1, 11)]
```

[50]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

La conjecture de Scholz-Brauer, non démontrée à ce jour, majore $\ell(2^p - 1)$ en fonction de $\ell(p)$. Elle a été vérifiée jusqu'à $n = 2^{36}$ (à ma connaissance).

Conjecture : $\ell(2^p - 1) \leq \ell(p) + p - 1$

Par exemple, $\ell(32) = 5$, comme calculé un peu plus haut. Donc si la conjecture est vérifiée, $\ell(2^{32}-1) = \ell(4294967295) \leq 5+32-1 = 36$. Notre majorant fétiche, quant à lui, vaut $2\lambda(2^{32})-1 = 63$. Le majorant de la conjecture est bien meilleur.

1.6.3 6.3 Petite bibliographie

- Donald E. Knuth. The Art of Computer Programmming Volume 2, Seminumerical Algorithms (1998). pages 461-485

De nombreux résultats sur les chaînes d'addition, et aussi des tas d'exercices. L'exercice 42 demande de démontrer la conjecture de Scholz-Brauer :-).

- Neill M. Clift. Calculating Opimal Addition Chains, Computing, 91 (2011). pages 265-284

Description d'un algorithme de calcul de chaînes optimales basé sur des techniques de graphes.

- F. Bergeron, J. Berstel, S. Brlek. Efficient Computation of Addition Chains. Journal de théorie des nombres de Bordeaux, tome 6, n° 1 (1994), pages 21-38.

Présentation de diverses stratégies de calcul de chaînes d'addition, et comparaison de ces stratégies.