Enum_Parties

April 3, 2023

1 Énumération des parties d'un ensemble

Marc Lorenzi

2 avril 2023

[1]:
```
import matplotlib.pyplot as plt
import random
```

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

Considérons le problème suivant. On se donne une liste $[x_1, \dots, x_n]$ d'entiers et un entier t. Existe-t-il un ensemble $A \subseteq [|1, n|]$ tel que

$$\sum_{i \in A} x_i = t \ ?$$

Ce problème est appelé **SUBSET SUM**. Une méthode évidente pour le résoudre est de considérer une à une toutes les parties A de $[|1, n|]$ et de regarder si l'une d'entre elles répond à la question.

Peut-on faire mieux ? La réponse est « non, pas vraiment », du moins en l'état de nos connaissances. Le problème **SUBSET SUM** fait partie d'une classe de problèmes appelée la classe **NP**. Pour faire simple, il est facile, si je vous donne un ensemble A, de vérifier qu'il convient. Voici d'ailleurs la fonction qui répond à la question (en supposant que A est bien une partie de E, ce que l'on pourrait aussi vérifier en temps raisonnable) :

[3]:
```
def check_subset(E, t, A):
    return sum(A) == t
```

[4]:
```
E = [19, 13, 4, 6, 12, 7, 7, 10, 2, 15, 19, 4, 19, 2, 11, 19, 18, 6, 8, 19]
check_subset(E, 110, [19, 13, 6, 12, 7, 7, 10, 2, 15, 19])
```

[4]:
```
True
```

La classe **NP** est la classe des problèmes pour lesquels il est facile de vérifier en « temps raisonnable » (raisonnable signifiant, pour faire vite, polynomial en la taille de l'entrée) une solution que l'on nous a donnée. Il se trouve qu'un certain nombre de problèmes de cette classe sont plus importants que les autres. Si on sait **résoudre** l'un de ces problèmes en temps raisonnable alors ont sait

résoudre **tous** les problèmes de la classe **NP** en temps raisonnable. Ces problèmes sont dits NP-complets. Il s'avère que **SUBSET_SUM** est NP-complet.

Le but de ce notebook est apparemment d'écrire une fonction qui résout **SUBSET SUM**. En réalité, ce qui m'intéresse c'est de parler de *codes de Gray*. La fonction que nous allons écrire sera de complexité exponentielle, bien évidemment. Sinon, nous serions en première ligne pour la médaille Fields. L'idée sera naïve : énumérer toutes les parties de l'ensemble E, calculer la somme des éléments de la partie. Si la somme est t, renvoyer la partie en question.

Clairement, nous sommes ramenés au problème suivant : comment énumérer aussi efficacement que possible toutes les parties d'un ensemble ?

1.1 1. L'approche récursive

Soit E un ensemble. Si E est vide, il n'a qu'une partie : l'ensemble vide. Sinon, soit x un élément de E. Soit $E' = E \setminus \{x\}$. Les parties de E sont les parties de E', ainsi que les parties de E' auxquelles on rajoute x. D'où, immédiatement, une fonction récursive qui résout la question.

[5]:
```
def liste_parties(E):
    if len(E) == 0: return [[]]
    else:
        P1 = liste_parties(E[1:])
        return P1 + [[E[0]] + A for A in P1]
```

[6]:
```
print(liste_parties(list(range(5))))
```

```
[[], [4], [3], [3, 4], [2], [2, 4], [2, 3], [2, 3, 4], [1], [1, 4], [1, 3], [1,
3, 4], [1, 2], [1, 2, 4], [1, 2, 3], [1, 2, 3, 4], [0], [0, 4], [0, 3], [0, 3,
4], [0, 2], [0, 2, 4], [0, 2, 3], [0, 2, 3, 4], [0, 1], [0, 1, 4], [0, 1, 3],
[0, 1, 3, 4], [0, 1, 2], [0, 1, 2, 4], [0, 1, 2, 3], [0, 1, 2, 3, 4]]
```

Si on réfléchit un peu on voit apparaître quelques inconvénients. On calcule toutes les parties de E et on les stocke dans une énorme liste. Cette fonction est donc non seulement de complexité exponentielle en temps mais aussi de complexité exponentielle en espace. De combien d'espace a-t-on besoin pour stocker la liste des parties de E ? Soit $n = |E|$, le cardinal de E. Pour $0 \leq k \leq n$, il y a $\binom{n}{k}$ parties de cardinal k, qui contiennent donc au total $k\binom{n}{k}$ éléments. Il reste à sommer sur k pour obtenir un total de

$$X_n = \sum_{k=0}^{n} k\binom{n}{k}$$

objets à stocker. Que vaut X_n ? Pour $1 \leq k \leq n$, on a $\frac{k}{n}\binom{n}{k} = \binom{n-1}{k-1}$. Donc

$$X_n = n\sum_{k=1}^{n} \frac{k}{n}\binom{n}{k} = n\sum_{k=1}^{n} \binom{n-1}{k-1}$$

Le changement d'indice $k' = k - 1$ donne

$$X_n = n \sum_{k=0}^{n-1} \binom{n-1}{k} = n2^{n-1}$$

C'est vraiment très gros. Pour $n = 20$ cela fait environ 10 millions d'objets.

[7]:
```
20 * 2 ** 19
```

[7]:
```
10485760
```

1.2 2. Fonction indicatrice

Tentons une autre approche. Une partie de la liste $[x_1, \dots, x_n]$ est caractérisée par les indices des x_i que nous voulons mettre dans cette partie. Ou, si l'on préfère, par un n-uplet $[\varepsilon_1, \dots, \varepsilon_n]$ où $\varepsilon_i = 0$ ou 1. Lorsqu'il vaut 1, on prend le x_i correspondant. Lorsqu'il vaut 0 on ne prend pas. Mais un tel n-uplet de 0 et de 1 peut être interprété comme la représentation binaire d'un entier entre 0 et $2^n - 1$. Énumérer les parties de E c'est donc énumérer les entiers entre 0 et $2^n - 1$. Et ça c'est très facile. Il faut juste écrire quelques fonctions de conversion.

La fonction `int_to_bin` prend deux entiers k et n en paramètres. Elle renvoie la représentation sur n bits de l'entier k, avec les bits de poids faible sur la gauche. Si vous préférez le contraire il n'y a qu'à faire un renversement de liste. La complexité temporelle de `int_to_bin` est clairement $O(n)$.

[8]:
```
def int_to_bin(k, n):
    s = []
    for i in range(n):
        s.append(k % 2)
        k = k // 2
    # s.reverse()
    return s
```

[9]:
```
int_to_bin(78, 12)
```

[9]:
```
[0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0]
```

La fonction `indic` prend en paramètres un ensemble E à n éléments et une liste c de bits. Elle renvoie le sous-ensemble de E correspondant. Ici encore, si E est de cardinal n, nous avons une complexité temporelle en $O(n)$.

[10]:
```
def indic(E, c):
    F = []
    for k in range(len(E)):
        if c[k] == 1:
            F.append(E[k])
    return F
```

[11]:
```
indic([0,1,2,3,4], [0,1,1,0,1])
```

[11]:
```
[1, 2, 4]
```

Il est maintenant facile d'énumérer les parties de E.

[12]:
```
def parties0(E):
    n = len(E)
    for k in range(2 ** n):
        yield indic(E, int_to_bin(k, n))
```

Que fait `yield`? Nous venons en fait de créer ce que l'on appelle un générateur. Exactement comme on peut écrire `for i in range(10): ...` ou `for x in s: ...` nous pouvons maintenant écrire la ligne `for P in parties0([1, 2, 3]): ....`

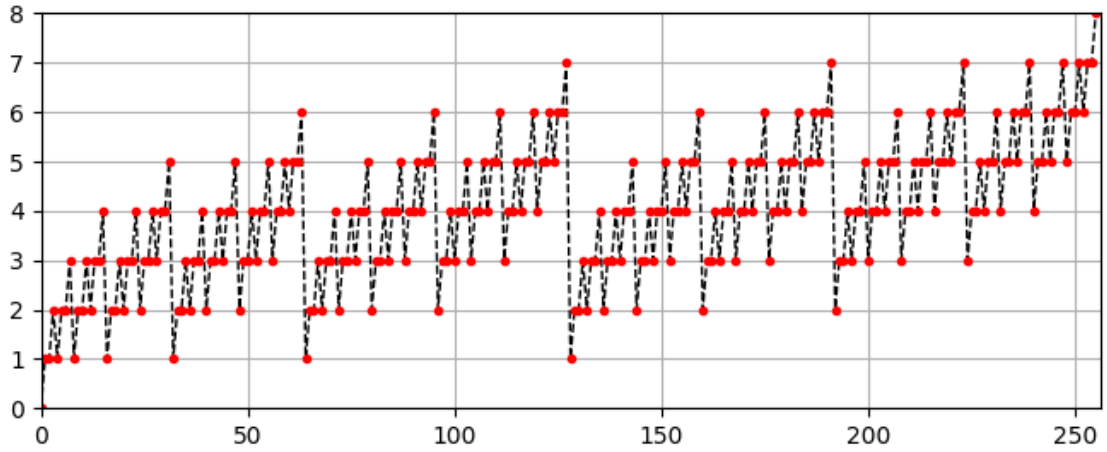
[13]:
```
for P in parties0([1, 2, 3, 4, 5]): print(P, end=' ')
```

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

Remarque. Si E est un ensemble de cardinal n, la complexité spatiale de l'énumération des parties de E est en $O(n)$ et la complexité temporelle en $O(n2^n)$.

Ci-dessous, un graphique représentant la taille des parties de E dans l'ordre où elles sont énumérées. À chaque étape, on ne peut monter que de 1, mais on peut redescendre beaucoup …

[14]:
```
n = 8
s = [len(p) for p in parties0(list(range(n)))]
plt.plot(s, '--k', lw=1)
plt.plot(s, 'or', ms=3)
plt.axis([0, 2 ** n, 0, n])
plt.grid()
plt.show()
```



1.3 3. Codes de Gray

1.3.1 3.1 Qu'est-ce qu'un code de Gray ?

Définition. Soit $n \in \mathbb{N}$. Un *code de Gray* de taille n est une énumération de tous les n-uplets de bits (i.e. 0 ou 1) telle que l'on passe d'un n-uplet au suivant en changeant la valeur d'un unique bit. Un tel code est *cyclique* lorsque, de plus, on passe du dernier n-uplet au premier en changeant aussi un unique bit.

Par exemple, un code de Gray cyclique de taille 2 est : 00, 10, 11, 01. Un code de Gray cyclique de taille 3 est 000, 100, 110, 010, 011, 111, 101, 001. Il existe de nombreuses familles de codes de Gray, et de nombreuses applications de ceux-ci. Nous allons nous intéresser aux codes de Gray dits cycliques réflexifs.

Si vous voulez une vision concrète d'un code de Gray, dessinez un cube dont les côtés ont pour coordonnées 0 ou 1. Les 8 sommets du cube ont pour coordonnées les éléments d'un code de Gray de taille 3. Puis parcourez les sommets de ce cube en partant de $[0, 0, 0]$ et en prenant les éléments successifs du code de Gray. Comme un seul bit est changé à chaque étape, on suit les arêtes du cube. On passe une et une seule fois par chaque sommet, et puis on revient au point de départ parce que le code est cyclique. On vient de faire un **parcours hamiltonien** du cube.

En résumé : un code de Gray cyclique de taille n est un cycle hamiltonien de l'hypercube en dimension n.

1.3.2 3.2 Créer des codes de Gray

Supposons créé un code de Gray de taille n tel que le premier n-uplet (le zéroième plutôt) soit

$$c_0 = [0, \dots, 0]$$

et le dernier (le $2^n - 1$-ième, donc) soit

$$c_{2^n-1} = [0, \dots, 0, 1]$$

Pour $i = 0, \dots 2^n - 1$, soit

- $c'_i = c_i$ auquel on rajoute un 0 à la fin, et
- $c''_i = c_i$ auquel on rajoute un 1 à la fin.

Il est alors clair que la suite $c'_0, \dots, c'_{2^n-1}, c''_{2^n-1}, \dots, c''_0$ est un code de Gray cyclique de taille $n + 1$ dont le premier code est $[0, \dots, 0]$ et le dernier est $[0, \dots, 0, 1]$. Aux soins du lecteur de vérifier les petits détails (le passage du dernier « prime » au premier « seconde » , le renversement d'un code de Gray en est-il encore un, etc.).

Nous venons de montrer par récurrence sur n que la fonction ci-dessous renvoie, pour tout n, un code de Gray cyclique de taille n dont le premier et le dernier élément sont $[0, \dots, 0]$ et $[0, \dots, 0, 1]$.

[15]:
```
def liste_gray(n):
    if n == 0: return [[]]
    else:
        g1 = liste_gray(n - 1)
        g2 = g1[:]
        g2.reverse()
        return [c + [0] for c in g1] + [c + [1] for c in g2]
```

[16]:
```
print(liste_gray(3))
```

```
[[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 1, 1], [1, 1, 1], [1, 0, 1],
[0, 0, 1]]
```

D'où une nouvelle fonction qui renvoie la liste des parties d'un ensemble E.

[17]:
```
def parties1(E):
    p = len(E)
    return [indic(E, c) for c in liste_gray(p)]
```

[18]:
```
print(parties1([1, 2, 3, 4, 5]))
```

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

Cela dit, il semblerait que nous soyons revenus au point de départ : la fonction qui renvoie les codes de Gray occupe un espace mémoire exponentiel. Ce qu'il faudrait, c'est trouver une formule magique qui fait passer d'un code de Gray au suivant, sans rien stocker à part le code et son numéro d'ordre. Regardons d'un peu plus près quel est le bit qui est changé à chaque étape.

1.3.3 3.3 Énumérer les codes de Gray

[19]:
```
print(liste_gray(4))
```

```
[[0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [0, 1, 0, 0], [0, 1, 1, 0], [1, 1, 1,
0], [1, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1], [0, 1,
1, 1], [0, 1, 0, 1], [1, 1, 0, 1], [1, 0, 0, 1], [0, 0, 0, 1]]
```

Pour le code à 4 bits, voici la liste des numéros du bit qui est changé pour passer du $i-1$ème code au $i$ème, $i = 1, 2, \ldots, 15$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0

Pour tout entier $i \geq 1$, notons $\nu_2(i)$ la *valuation dyadique* de i, c'est à dire le plus grand entier k tel que 2^k divise i.

Proposition. pour tout entier $n \geq 1$, pour tout i entre 1 et $2^n - 1$, on passe du $(i-1)$-ième code de Gray au i-ième code de Gray en modifiant le $\nu_2(i)$-ème bit du code.

Démonstration. On fait une récurrence sur n. Pour $n = 1$, c'est évident. Soit $n \geq 1$. Supposons la propriété vraie pour n. Soit $1 \leq i \leq 2^{n+1} - 1$. Il y a 3 cas à considérer :

- $1 \leq i \leq 2^n - 1$: le i-ème code de gray de taille $n+1$ est le i-ème de code de taille n auquel on a rajouté un 0 à la fin. De même pour le $i-1$-ième. L'hypothèse de récurrence s'applique directement.
- $i = 2^n$: c'est le dernier bit, c'est à dire le bit n qui est modifié pour passer du $(i-1)$-ième code au i-ième. Cela tombe bien puisque $\nu_2(2^n) = n$.
- $2^n + 1 \leq i \leq 2^{n+1} - 1$: là on se trouve dans la deuxième moitié de la liste, qui est la liste ***inversée*** des codes de taille n auxquels on a rajouté un 1. Il faut donc se demander comment on passe du $i' + 1$-ième code de taille n au i'-ième code de taille n, où $i' = 2^{n+1} - i - 1$. Inverser un bit est une opération involutive, donc par l'hypothèse de récurrence, cela se fait en changeant le bit numéro $\nu_2(i' + 1)$. Mais $\nu_2(i' + 1) = \nu_2(2^{n+1} - i) = \nu_2(i)$, car $i < 2^{n+1}$.

Maintenant c'est facile. La fonction `nu2` renvoie $\nu_2(n)$, où n est censé être un entier naturel non nul.

[20]:
```
def nu2(n):
    c = 0
    while n % 2 == 0:
        c += 1
        n = n // 2
    return c
```

[21]:
```
print([nu2(i) for i in range(1, 16)])
```

```
[0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0]
```

La fonction `next` prend un couple $u = (c, n)$ en paramètre, où c est le $n$ième code de Gray. Elle renvoie $(c', n+1)$ où c' est le $(n+1)$-ième code de Gray.

[22]:
```
def next(u):
    c, n = u
    j = nu2(n + 1)
    c [j] = 1 - c[j]
    return (c, n + 1)
```

[23]:
```
u = ([0, 0, 0, 0], 0)
print(u)
for k in range(15):
    u = next(u)
    print(u)
```

```
([0, 0, 0, 0], 0)
([1, 0, 0, 0], 1)
([1, 1, 0, 0], 2)
([0, 1, 0, 0], 3)
```

```
([0, 1, 1, 0], 4)
([1, 1, 1, 0], 5)
([1, 0, 1, 0], 6)
([0, 0, 1, 0], 7)
([0, 0, 1, 1], 8)
([1, 0, 1, 1], 9)
([1, 1, 1, 1], 10)
([0, 1, 1, 1], 11)
([0, 1, 0, 1], 12)
([1, 1, 0, 1], 13)
([1, 0, 0, 1], 14)
([0, 0, 0, 1], 15)
```

1.3.4 3.3 Un générateur de codes de Gray

La fonction `gray` ci-dessous prend un paramètre n en entrée. Elle initialise une variable `code` au n-uplet qui ne contient que des zéros. Puis elle effectue $2^n - 1$ fois le calcul de l'élément suivant du code de Gray, qu'elle place dans la variable `code`. Bref, elle énumère les codes de Gray.

[24]:
```
def gray(n):
    code = n * [0]
    yield code
    for k in range(1, 2 ** n):
        j = nu2(k)
        code[j] = 1 - code[j]
        yield code
```

[25]:
```
for c in gray(3): print(c)
```

```
[0, 0, 0]
[1, 0, 0]
[1, 1, 0]
[0, 1, 0]
[0, 1, 1]
[1, 1, 1]
[1, 0, 1]
[0, 0, 1]
```

1.3.5 3.4 Complexité

La complexité temporelle de l'énumération des codes de Gray est bien entendu exponentielle. On n'y peut rien, un ensemble de cardinal n a 2^n parties. Une majoration naïve de la complexité nous dit que `nu2` prend en pire cas un temps de l'ordre de n pour calculer la valuation dyadique d'un entier $k \leq 2^n$. La complexité en temps de l'énumération des codes de Gray est donc un $O(n2^n)$.

En fait, c'est mieux que cela. Le nombre d'opérations nécessaires au calcul de $\nu_2(k)$ est … $\nu_2(k)$. Le nombre total d'opérations effectuées est ainsi

$$C_n = \sum_{k=1}^{2^n-1} \nu_2(k)$$

Proposition. $C_n = 2^n - n - 1$.

Démonstration. Pour tout $i \in [|0, n-1|]$, soit

$$A_i = \{k \in [|1, 2^n - 1|] : \nu_2(k) = i\}$$

On a alors

$$C_n = \sum_{i=0}^{n-1} i|A_i|$$

Quels est le cardinal de A_i ? Soit $k \in [|1, 2^n - 1|]$. On a $k \in A_i$ si et seulement si il existe $p \in \mathbb{N}$ tel que $k = 2^i(2p+1)$ et

$$2^i(2p+1) < 2^n$$

ou encore

$$p < 2^{n-i-1} - \frac{1}{2}$$

Comme p est entier, ceci équivaut à

$$0 \leq p < 2^{n-i-1}$$

Ainsi, pour tout $i \in [|0, n-1|]$, $|A_i| = 2^{n-i-1}$. De là,

$$C_n = \sum_{i=0}^{n-1} i2^{n-i-1}$$

On vérifie alors facilement par récurrence sur n que $C_n = 2^n - n - 1$.

Finalement, la complexité temporelle de l'énumération des codes de Gray est un $O(2^n)$. Sachant qu'il y a 2^n codes, on peut difficilement faire mieux !

[26]:
```
def complexite_gray(n):
    s = 0
    for k in range(1, 2 ** n):
        s += nu2(k)
    return s
```

[27]:
```
for n in range(11):
    print('%3d%5d%5d' % (n, complexite_gray(n), 2 ** n - n - 1))
```

```
 0    0    0
 1    0    0
 2    1    1
 3    4    4
 4   11   11
 5   26   26
 6   57   57
 7  120  120
 8  247  247
 9  502  502
10 1013 1013
```

1.4 4. Parties de E, version définitve

1.4.1 4.1 Le code

Écrire un générateur des parties d'un ensemble est maintenant immédiat.

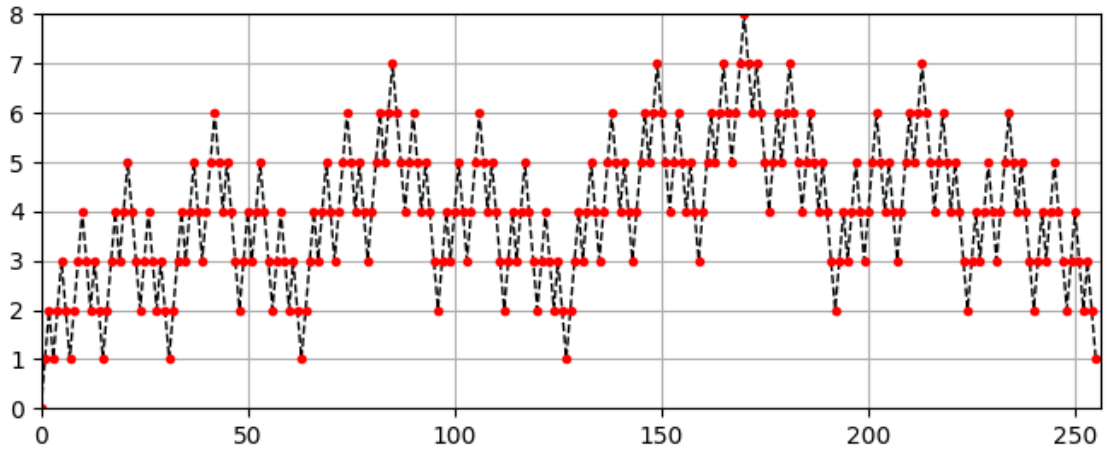
[28]:
```
def parties(E):
    for c in gray(len(E)):
        yield indic(E, c)
```

[29]:
```
for p in parties(range(6)): print(p, end=' ')
```

```
[] [0] [0, 1] [1] [1, 2] [0, 1, 2] [0, 2] [2] [2, 3] [0, 2, 3] [0, 1, 2, 3] [1,
2, 3] [1, 3] [0, 1, 3] [0, 3] [3] [3, 4] [0, 3, 4] [0, 1, 3, 4] [1, 3, 4] [1, 2,
3, 4] [0, 1, 2, 3, 4] [0, 2, 3, 4] [2, 3, 4] [2, 4] [0, 2, 4] [0, 1, 2, 4] [1,
2, 4] [1, 4] [0, 1, 4] [0, 4] [4] [4, 5] [0, 4, 5] [0, 1, 4, 5] [1, 4, 5] [1, 2,
4, 5] [0, 1, 2, 4, 5] [0, 2, 4, 5] [2, 4, 5] [2, 3, 4, 5] [0, 2, 3, 4, 5] [0, 1,
2, 3, 4, 5] [1, 2, 3, 4, 5] [1, 3, 4, 5] [0, 1, 3, 4, 5] [0, 3, 4, 5] [3, 4, 5]
[3, 5] [0, 3, 5] [0, 1, 3, 5] [1, 3, 5] [1, 2, 3, 5] [0, 1, 2, 3, 5] [0, 2, 3,
5] [2, 3, 5] [2, 5] [0, 2, 5] [0, 1, 2, 5] [1, 2, 5] [1, 5] [0, 1, 5] [0, 5] [5]
```

Maintenant, chaque partie renvoyée par le générateur a un rapport simple avec la précédente et la suivante.

[30]:
```
n = 8
s = [len(p) for p in parties(list(range(n)))]
plt.plot(s, '--k', lw=1)
plt.plot(s, 'or', ms=3)
plt.axis([0, 2 ** n, 0, n])
plt.grid()
plt.show()
```



1.4.2 4.2 Complexité

Si E est un ensemble de cardinal n, la fonction `indic` prend un temps de l'ordre de n pour créer un ensemble à partir d'une liste de 0 et de 1. Le temps cumulé de la création de toutes les parties de E se fait donc en temps $O(n2^n)$. Nous avons déjà vu que le temps cumulé du calcul des $\nu_2(k)$ est un $O(2^n)$. Ainsi, l'énumération de toutes les parties de E a une complexité temporelle en $O(n2^n)$.

Qu'en est-il de la complexité en espace ? Mis à part l'espace utilisé par E, il faut uniquement de l'espace pour stocker la variable `code`, qui est une liste de n bits, plus un espace constant pour les variables restantes. La complexité en espace est donc en $O(n)$ où n est le nombre d'éléments de E.

Résumons nous. Pour un ensemble de cardinal n,

- La complexité temporelle de l'énumération est $O(n2^n)$.
- La complexité spatiale de l'énumération est $O(n)$.

1.5 5. SUBSET SUM en trois lignes

La fonction `subset_sum` est maintenant évidente. Sa complexité en pire cas est exponentielle. Cela survient en particulier lorsque le problème n'a pas de solution.

[31]:
```
def subset_sum(E, t):
    for p in parties(E):
        if sum(p) == t: return (True, p)
    return (False, None)
```

[32]:
```
E = [random.randint(1, 10) for k in range(20)]
print(E)
print(subset_sum(E, 32))
```

```
[10, 9, 9, 10, 10, 5, 7, 7, 6, 9, 6, 4, 1, 8, 6, 7, 9, 5, 8, 2]
(True, [10, 10, 5, 7])
```