Problème

Je dispose d'une pièce pour jouer à pile ou face ; je lance la pièce jusqu'à ce que j'obtienne une série de N faces. Par exemple, si N = 4, je m'arrête lorsque j'ai obtenu 4 faces d'affilée.

La question que je me suis posé est la suivante : en moyenne, combien de lancés je dois réaliser en fonction de N ?

Solution

Je choisis de travailler sur des suites binaires, où 1 représente le résultat « face » et 0 représente « pile ».

Commençons par N = 1 et notons les suites possibles en ordre croissant de longueur, ainsi que leur probabilité de survenir :

Suite	Probabilité	Nombre de lancés
1	1/2	1
01	1/4	2
001	1/8	3
0001	1/16	4
…	…	…
0……..01	$\frac{1}{2^n}$	n

Le pattern est très clair, et pour obtenir la moyenne de lancés, il suffit de calculer la somme pesée, dont la valeur est bien connue (cf https://en.wikipedia.org/wiki/List_of_mathematical_series) :

$$\sum_{n=1}^{\infty} \frac{n}{2^n} = 2$$

Passons maintenant à N = 2 :

Suite	Nombre de suites	Probabilité	Nombre de lancés
11	1	1/4	2
011	1	1/8	3
0011 1011	2	1/16	4
00011 01011 10011	3	1/32	5
000011 001011 010011 100011 101011	5	1/64	6
0000011 0001011 0010011 0100011 0101011 1000011	8	1/128	7

1001011 1010011			
…		…	…
……011	Fibonacci(n-1)	$\frac{1}{2^n}$	n

J'avoue avoir été assez stupéfait de voir émerger une série de Fibonacci !

La moyenne donne donc :

$$\sum_{n=2}^{\infty} Fibonacci(n-1)\frac{n}{2^n}$$

Pour en calculer la valeur, j'ai utilisé le programme en Annexe 1, qui donne une valeur de 5.99999993 en calculant jusqu'à n=100. On peut assez tranquillement affirmer que cette série converge donc à 6.

Autrement dit, il faut lancer en moyenne la pièce 6 fois pour obtenir une suite de 2 faces d'affilée.

Bon, après avoir vu émerger la série de Fibonacci, j'ai eu un peu peur et j'ai changé d'approche, optant pour une simulation de Montercarlo, avec le programme en Annexe 2, qui a donné ceci :

```
1 2.00008349
2 6.00048314
3 13.99854985
4 29.99942706
5 62.00827439
6 126.0053699
7 253.9596006
8 509.8903546
9 1021.7156187
10 2046.0056432
```

Les résultats sont les suivants :

N	Nombre moyen de lancés
1	2
2	6
3	14
4	30
5	62
…	…
N	$2^{N+1}-2$

La solution du problème est donc claire : pour obtenir N faces d'affilée il faut réaliser en moyenne $2^{N+1}-2$ lancés.

Je reste un peu sur ma faim car je n'ai pas trouvé un raisonnement « pur » pour arriver à la solution. Si vous le trouvez, n'hésitez pas à me l'indiquer.

Pour améliorer ma compréhension, j'ai modifié un peu le problème initial :

Problème 2

Je dispose d'une pièce pour jouer à pile ou face ; je lance la pièce jusqu'à ce que j'obtienne une série de N faces *ou* piles. Par exemple, si N = 4, je m'arrête lorsque j'ai obtenu 4 faces *ou* 4 piles d'affilée.

En moyenne, combien de lancés je dois réaliser en fonction de N ?

Solution 2

La simulation en Annexe 3 donne :

```
1 1
2 2.9998872
3 7.00057544
4 14.99974569
5 30.99736197
6 63.00951648
7 126.98714508
8 254.9830601
9 510.9026758
10 1022.8947224
```

Les résultats sont les suivants :

N	Nombre moyen de lancés
1	1
2	3
3	7
4	15
5	31
…	…
N	$2^N - 1$

La solution est donc $2^N - 1$, exactement la moitié que dans le problème précédent, ce qui est logique vue que j'ai en quelque sorte doublé le nombre de « cas de sortie » en rendant le problème plus générique.

Reste à trouver un raisonnement logique qui amène à ce résultat… mais bon, c'est ça l'utilité des simulations :-)

Allez, allons encore plus loin dans la généralisation :

Problème 3

Je dispose d'un dé avec k faces ; je lance le dé jusqu'à ce que j'obtienne une série de N symboles identiques.

En moyenne, combien de lancés je dois réaliser en fonction de N et de k ?

Solution 3

Continuons sur la même approche, avec le simulateur en Annexe 3 :

```
2 2 3.0003191
2 3 7.0004895
2 4 15.0019223
3 2 3.9995303
3 3 13.0002215
3 4 39.9749993
4 2 4.9987915
4 3 20.9961661
4 4 85.0202601
5 2 6.0012863
5 3 31.0124243
5 4 155.9544148
6 2 6.9983229
6 3 42.9962023
6 4 258.9671778
```

La généralisation saute aux yeux :

$$n(k,N) = \sum_{i=0}^{N-1} k^i = \frac{1-k^N}{1-k}$$

Vérifions le cas de pile ou face avec k = 2 : ça donne :

$$n(2,N) = \sum_{i=0}^{N-1} 2^i = \frac{1-2^N}{-1} = 2^N - 1$$

Avec k = 6 (lancé de dé traditionnel) :

$$n(6,N) = \sum_{i=0}^{N-1} 6^i = \frac{1-6^N}{-5} = \frac{6^N - 1}{5}$$

ANNEXE 1

```
const fibonacci_series = (n) => {
  if (n===1) {
    return [0, 1];
  } else {
    var s = fibonacci_series(n - 1);
    s.push(s[s.length - 1] + s[s.length - 2]);
    return s;
  }
};

const fib = fibonacci_series(100);

const ser2 = (N) => {
    let sum = 0;
    for (let i = 1; i <= N; i++) {
        sum = sum + (fib[i]*((i+1) / (2**(i+1))));
    }
    return sum;
}

const s2 = ser2(100);

console.log(s2);
```

ANNEXE 2

```
const unOuZero = () => {
    if (Math.random() > 0.5) {
        return 1;
    }
    return 0;
}

const runSimu = (nseq, N) => {
    let consec1 = 0;
    for (let i = 1; i <= N; i++) {
        const res = unOuZero();
        if (res) {
            consec1++;
            if (consec1 >= nseq) {
                return i;
            }
        }
        else {
            consec1 = 0;
        }
    }
    return N;
}

const runNsimu = (nseq, N1, N2) => {
    let sum = 0;
    for (let i = 1; i <= N1; i++) {
        let res = runSimu(nseq, N2);
        sum += res;
    }
    return (sum / N1);
}

for (let i = 1; i <= 10; i++) {
    let s1;
    if (i <= 5) {
        s1 = runNsimu(i, 100000000, 10000);
    }
    else if (i <= 7) {
        s1 = runNsimu(i, 50000000, 100000);
    }
    else {
        s1 = runNsimu(i, 10000000, 1000000);
    }
    console.log(i, s1);
}
```

ANNEXE 3

```
const throwDice = (k) => {
    const pas = 1/k;
    const res = Math.random();

    for (let i = 1; i < k; i++) {
        if (res < (i*pas)) {
            return i;
        }
    }
    return k;
}

const runSimu = (nseq, N, k) => {
    let consec = {};
    for (let j = 1; j <= k; j++) {
        consec[j] = 0;
    }

    for (let i = 1; i <= N; i++) {
        const res = throwDice(k);

        // consec[i] = 0 for i != res, consec[res]++;
        for (let t = 1; t <= k; t++) {
            if (t !== res) {
                consec[t] = 0;
            }
        }
        consec[res]++;

        if (consec[res] >= nseq) {
            return i;
        }
    }
    return N;
}

const runNsimu = (nseq, N1, N2, k) => {
    let sum = 0;
    for (let i = 1; i <= N1; i++) {
        let res = runSimu(nseq, N2, k);
        sum += res;
    }
    return (sum / N1);
}

[2,3,4,5,6].forEach((k) => {
```

```
    for (let i = 2; i <= 5; i++) {
        let s1 = runNsimu(i, 10000000, 10000, k);
        console.log(k, i, s1);
    }
});
```