



Générateurs de nombres pseudo-aléatoires

BOUJOT Nicolas, DEMONCEAUX Mathis, KAPTAN Dogan

5ème semestre

Table des matières

1	Introduction	3
2	Les différents générateurs de nombres pseudo-aléatoires	4
2.1	La méthode de Von Neumann	4
2.2	La méthode de Fibonacci	4
2.3	Les générateurs congruentiels linéaires	5
3	Générateur congruentiel linéaire	6
3.1	Choix des paramètres	6
3.2	Test d'hypothèse	7
3.2.1	Test du Khi-2	7
3.2.2	Approche par p-value	8
3.3	Implémentation et résultats	9
4	Conclusion	11
A	Bibliographie	12
B	Annexes	13

Introduction

Un **générateur de nombres pseudo-aléatoires** est un algorithme qui génère une séquence de nombres présentant certaines propriétés du hasard. Par exemple, les nombres sont supposés être approximativement indépendants les uns des autres, et il est potentiellement difficile de repérer des groupes de nombres qui suivent une certaine règle (comportements de groupe).

Cependant, les sorties d'un tel générateur ne sont pas entièrement aléatoires ; elles s'approchent seulement des propriétés idéales des sources complètement aléatoires. La raison pour laquelle on se contente d'un rendu pseudo-aléatoire est : d'une part qu'il est difficile d'obtenir de « vrais » nombres aléatoires et que ce sont des générateurs particulièrement adaptés à une implantation informatique, donc plus facilement et plus efficacement utilisables.

Les méthodes pseudo-aléatoires sont souvent employées sur des ordinateurs, dans diverses tâches comme la méthode de Monte-Carlo, la simulation, les applications cryptographiques ou encore les jeux-vidéo.

La plupart des algorithmes pseudo-aléatoires essaient de produire des sorties qui sont uniformément distribuées. Il existe différents générateurs de nombres pseudo-aléatoire, que nous verrons dans notre première partie. Dans notre seconde partie, nous verrons une classe très répandue de générateurs : les générateurs congruentiel linéaire.

Les différents générateurs de nombres pseudo-aléatoires

Il existe une multitude d'algorithmes connus permettant une génération de nombres pseudo-aléatoires, plus ou moins fiables ou rapides. Pour qu'un algorithme soit valable, il doit suivre certains points :

- Une vitesse de génération acceptable,
- Qu'il soit reproductible,
- Qu'il ressemble bien sûr le plus possible au vrai hasard,
- Que la suite créée soit normale.

2.1 La méthode de Von Neumann

La méthode du carré médian (middle-Square method) a été décrite pour la première fois vers 1240 puis elle fut ensuite réinventée par John Von Neumann en 1946. Elle est très imparfaite (car elle converge toujours vers 0 ou bien qu'on retrouve très rapidement les mêmes nombres) mais elle fournit tout de même un moyen simple de produire des nombres pseudo-aléatoires.

Cette méthode est très simple : elle consiste à prendre un nombre, à l'élever au carré et à prendre les chiffres au milieu comme sortie. Celle-ci est utilisée comme graine pour l'itération suivante. Exemple du fonctionnement :

Soit le nombre « 1111 ».

1. $1111^2 = 1234321$
2. on récupère les chiffres du milieu : 3432. C'est la sortie du générateur.
3. $3432^2 = 11778624$
4. on récupère les chiffres du milieu : 7786.

et ainsi de suite.

2.2 La méthode de Fibonacci

Cette méthode est basée sur la suite de Fibonacci modulo la valeur maximale désirée. Elle demande peu de ressources et est très simple à implémenter. Toutefois, sa qualité dépend fortement des nombres utilisés pour l'initialiser.

On la formalise ainsi :

$$x_n = (x_{n-1} + x_{n-2}) \bmod M$$

avec x_0 et x_1 en entrée. C'est à dire que chaque terme de la suite est la somme des deux termes qui le précèdent, modulo un nombre M . C'est donc une congruence additive.

L'application ci-dessous illustre le fonctionnement de cet algorithme, en partant des nombres 87 et 65 ainsi que 100 comme modulo :

87, 65, 52, 17, 69, 86, 55, 41, 96, 37, 33, 70, 3, 73, 76, 49,
25, 74, 99, 73, 72, 45, 17, 62, 79, 41, 20, 61, 81, 42, 23, 65,
88, 53, 41, 94, 35, 29, 64, 93, 57, 50, 7, 57, 64, 21, 85, 6, ...

2.3 Les générateurs congruentiels linéaires

Il s'agit de l'algorithme le plus utilisé pour produire des nombres aléatoires. Il a été inventé en 1948 par le mathématicien américain Derrick Henry Lehmer.

Pour fabriquer un générateur à congruence linéaire, il faut 4 nombres :

- Un germe x_0 qui doit être un nombre entier positif,
- Un multiplicateur a
- Un incrément b qui doit être un nombre entier non nul,
- Un nombre m qui définira la taille du plus grand entier possible.

Ces 4 paramètres (que nous étudierons plus en profondeur plus tard) nous permette de construire la formule de récurrence suivante :

$$X_{n+1} = (a * X_n + c) \bmod m$$

L'application ci-dessous illustre le fonctionnement de cet algorithme, en partant de $x_0 = 21$, $a = 12$, $b = 25$, $m = 100$:

21, 77, 49, 13, 81, 97, 89, 93, 41, 17, 29, 73, 1, 37, 69, 53,
61, 57, 9, 33, 21, 77, 49, 13, 81, 97, 89, 93, 41, 17, 29, 73,
1, 37, 69, 53, 61, 57, 9, 33, 21, 77, 49, 13, 81, 97, 89, 93, ...

Nous allons voir plus en détail ce générateur dans la partie suivante.

Générateur congruentiel linéaire

Dans cette partie nous allons voir que le choix des paramètres du générateur congruentiel linéaire est très important, même primordiaux. Ceux-ci permettent de maximiser la période du générateur, c'est à dire le nombre de récurrences à atteindre avant de n'avoir en résultat que des valeurs nulles ou une répétition.

Nous effectuerons ensuite un test d'hypothèse afin de vérifier que ce générateur s'approche suffisamment d'une loi uniforme pour le valider. Enfin nous implémenterons ce générateur en C++ et commenterons les résultats observés.

Ci-dessous un rappel de la formule de ce générateur :

$$X_{n+1} = (a * X_n + c) \bmod m$$

3.1 Choix des paramètres

Comme dit précédemment, avec cet algorithme, le choix de a , b et m sont primordiaux.

La période de ce générateur est au maximum de m , c'est-à-dire qu'elle est relativement courte puisque m est souvent choisi de manière à être de l'ordre de la longueur des mots sur l'ordinateur (par exemple : 2^{32} sur une machine 32 bits). Mais cette méthode présente un avantage : on connaît les critères sur les nombres a , c et m qui vont permettre d'obtenir une période maximale (égale à m).

Pour maximiser cette période, on observera les conditions suivantes :

- b doit être premier avec m , c'est à dire $PGCD(b, m) = 1$.
- Pour chaque nombre premier p divisant m , $(a - 1)$ est un multiple de p .
- $(a - 1)$ doit être un multiple de 4 si m en est un.

Dans le cas où b est nul, la période est maximale si :

- m est premier,
- a^{m-1} est un multiple de m ,
- a^{j-1} n'est pas divisible par m pour $j = 1, j = 2, \dots, j = m - 2$

Le terme initial, X_0 est appelé la graine (*seed* en anglais). C'est elle qui va permettre de générer une suite apparemment aléatoire. Pour chaque graine, on aura une nouvelle suite. Cependant, il est possible que certaines graines permettent d'obtenir une suite plus aléatoire que d'autres. Le fait de pouvoir générer la même suite aléatoire en gardant la même graine est intéressant dans certains domaines tels que les jeux-vidéo, où deux personnes peuvent générer la même carte en paramétrant la même graine de génération.

Ces précédentes indications permette de comprendre qu'il n'est pas une bonne idée de choisir ces paramètres "au hasard", voici quelques exemples avec le générateur congruentiel linéaire de paramètres $a = 25$, $c = 16$, $m = 256$

- avec $X_0 = 10$, la suite : 10, 10, 10, 10, 10, ...
- avec $X_0 = 11$, la suite : 11, 35, 123, 19, 235, 3, 91, 243, 203, 227, 59, 211, 171, 195, 27, 179, 139, 163, 251, 147, 107, 131, ...
- avec $X_0 = 12$, la suite : 12, 60, 236, 28, 204, 252, 172, 220, 140, 188, 108, 156, 76, 124, 44, 92, **12, 60, 236, 28, 204, 252, 172, ...**

Il est clair que ces suites ne peuvent être considérées comme aléatoires.

Voici une liste non exhaustive des choix de paramètres les plus connus :

— **RANDU** : $a = 65539$, $c = 0$, $m = 2^{31}$

Implanté sur les IBM System/370, les premiers ordinateurs des années 60, il est réputé mauvais.

— **Générateur de Robert Sedgewick** : $a = 31415821$, $c = 1$, $m = 10^8$

Cet algorithme fonctionne et n'a pas de défauts apparents, mais lorsqu'on le teste, on peut voir que tous les nombres sortis finissent toujours par un dernier chiffre plus grand que le précédent

— **Générateur d'UNIX** : $a = 1103515245$, $c = 12345$, $m = 2^{31}$

Le comité ANSI C a recommandé de ne conserver que les 16 bits de poids fort, car les bits de poids faible des nombres produits ne sont pas vraiment aléatoires.

— **Standard minimal** : $a = 16807$, $c = 0$, $m = 2^{31} - 1$

Inventé par Park Miller, il est très largement utilisé grâce à la facilité de son implémentation sur quasi toutes les machines.

3.2 Test d'hypothèse

3.2.1 Test du Khi-2

Le test du χ^2 permet, partant d'une hypothèse et d'un risque supposé au départ, de rejeter l'hypothèse si la distance entre deux ensembles d'informations est jugée excessive, permettant ainsi de comparer ladite hypothèse à une loi de probabilité.

En 1900, un mathématicien britannique, Karl Pearson, mit en évidence qu'une grande différence entre la loi théorique et la mesure réelle a plus d'importance que plusieurs petites différences. Cela a donné le test du χ^2 qui est un cas particulier de test statistique d'hypothèse qui fournit une méthode pour déterminer la nature d'une répartition, qui peut être continue ou discrète.

Nous nous occuperons ici de déterminer si une répartition est uniforme dans le cas discret.

Dans ce cadre, la méthode que nous allons utiliser est la suivante :

1) On répartit les valeurs de l'échantillon (de taille n) dans k classes distinctes et on calcule les effectifs de ces classes. Il faut vérifier que pour les i de 1 à k , on a $np_i(1 - p_i) \geq 5$ (éventuellement répartir les valeurs autrement). Appelons o_i ($i = 1, \dots, k$) les effectifs observés et e_i les effectifs théoriques.

2) On calcule $Q = \sum_{i=1}^k \frac{(o_i - e_i)^2}{e_i}$.

La statistique Q donne une mesure de l'écart existant entre les effectifs théoriques attendus et ceux observés dans l'échantillon. En effet, plus Q sera grand, plus le désaccord sera important. La coïncidence sera parfaite si $Q = 0$.

3) On compare ensuite cette valeur Q avec une valeur $\chi_{k-1, \alpha}^2$ issue d'un tableau (voir figure 3) à la ligne $k-1$ et à la colonne α . $k-1$ est le nombre de degrés de liberté et α la tolérance.

4) Si $Q > \chi_{k-1, \alpha}^2$, et si n est suffisamment grand, alors l'hypothèse d'avoir effectivement affaire à la répartition théorique voulue est à rejeter avec une probabilité d'erreur d'au plus α .

Démonstration :

On a utilisé le générateur congruentiel linéaire avec les paramètres : $a = 214013$, $c = 2531011$, $m = 2147483648$, $x_0(\text{graine}) = 27$ et un nombre d'essais = 100000, on a obtenu les issues 1 à 10 ($k = 10$) avec les effectifs suivants : 10141, 9949, 9757, 9998, 9856, 9961, 10054, 10191, 10028, 10065 (on a vérifié que 100 000 essais sont suffisants : $n(\frac{1}{10})(\frac{9}{10}) \geq 5$ implique que $n \geq \frac{500}{9}$).

Pour 100000 essais :		
1 :	10141	10.141%
2 :	9949	9.949%
3 :	9757	9.757%
4 :	9998	9.998%
5 :	9856	9.856%
6 :	9961	9.961%
7 :	10054	10.054%
8 :	10191	10.191%
9 :	10028	10.028%
10 :	10065	10.065%

Si le générateur congruentiel linéaire est bien aléatoire et suit une loi uniforme, on attend comme effectifs moyens théoriques 10 000 pour toutes les issues.

$$Q = \frac{(10141 - 10000)^2}{10000} + \frac{(9949 - 10000)^2}{10000} + \frac{(9757 - 10000)^2}{10000} + \frac{(9998 - 10000)^2}{10000} + \frac{(9856 - 10000)^2}{10000} + \frac{(9961 - 10000)^2}{10000} + \frac{(10054 - 10000)^2}{10000} + \frac{(10191 - 10000)^2}{10000} + \frac{(10028 - 10000)^2}{10000} + \frac{(10065 - 10000)^2}{10000} = 14.8198$$

Pour $k - 1 = 9$ degrés de liberté et un seuil de tolérance de 5%, la valeur $\chi_{k-1, \alpha}^2$ du tableau (figure 3) est 16.92. Cela signifie que la probabilité que Q soit supérieur à 16.92 est de 5%. Comme $14.8198 < 16.92$, on accepte l'hypothèse selon laquelle le générateur congruentiel linéaire suit une loi uniforme.

3.2.2 Approche par p-value

Dans un test statistique, la p-value est la probabilité pour un modèle statistique donné sous l'hypothèse nulle d'obtenir la même valeur ou une valeur plus extrême que celle observée.

Dans notre cas où $\alpha = 5\%$, le degré de liberté = 9 et par conséquent le $\chi_{seuil}^2 = 16.92$, cette valeur seuil est la valeur qui ne doit pas être dépassée pour ne pas rejeter l'hypothèse nulle.

Nous pouvons par ailleurs rechercher la probabilité associée à la valeur du critère χ_{obs}^2 que nous venons de calculer à partir du tableau, nous trouvons que χ_{obs}^2 qui vaut 14.8198 est encadré par les deux valeurs qui sur la ligne à 9 degré de liberté sont 14.68 et 16.92. La probabilité p-value correspond donc à une valeur comprise entre $5\% < p - value < 10\%$.

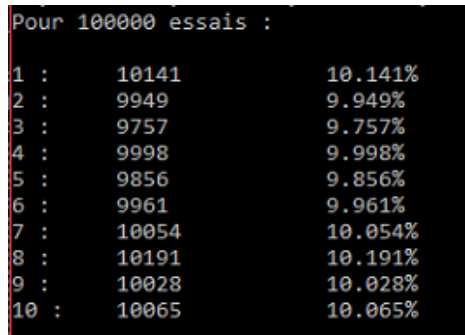
Étant donné que $\chi_{obs}^2 < \chi_{seuil}^2$ pour $\alpha = 5\%$, ce qui revient au même que la p-value $> 5\%$: on peut accepter l'hypothèse nulle et conclure que le générateur congruentiel linéaire suit une loi uniforme.

3.3 Implémentation et résultats

Nous avons eu pour consigne d'implémenter un générateur congruentiel linéaire dans le langage de programmation de notre choix, nous avons donc choisi le C++, en particulier pour sa puissance.

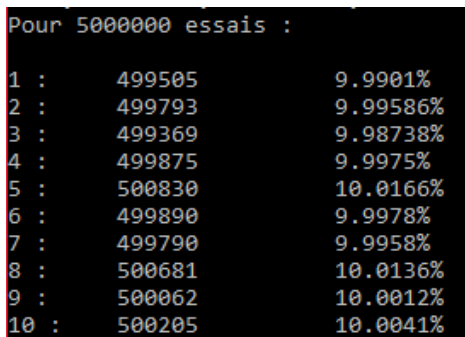
Nous avons créé une classe *congruentialGenerator*, possédant les paramètres a, c, m et X_0 et une fonction *next()* renvoyant le prochain nombre aléatoire compris entre $[0; 1[$ (Voir Figure 1 et 2 en Annexe). Cette implémentation nous a permis de vérifier et de manipuler le générateur congruentiel en l'imageant.

Nous avons d'abord commencé nos expérimentations avec les paramètres $a = 214013$, $c = 2531011$, $m = 2147483648$ et avec la graine $x_0 = 27$, en générant 100 000 nombres aléatoires de 1 à 10, voici les résultats observés :



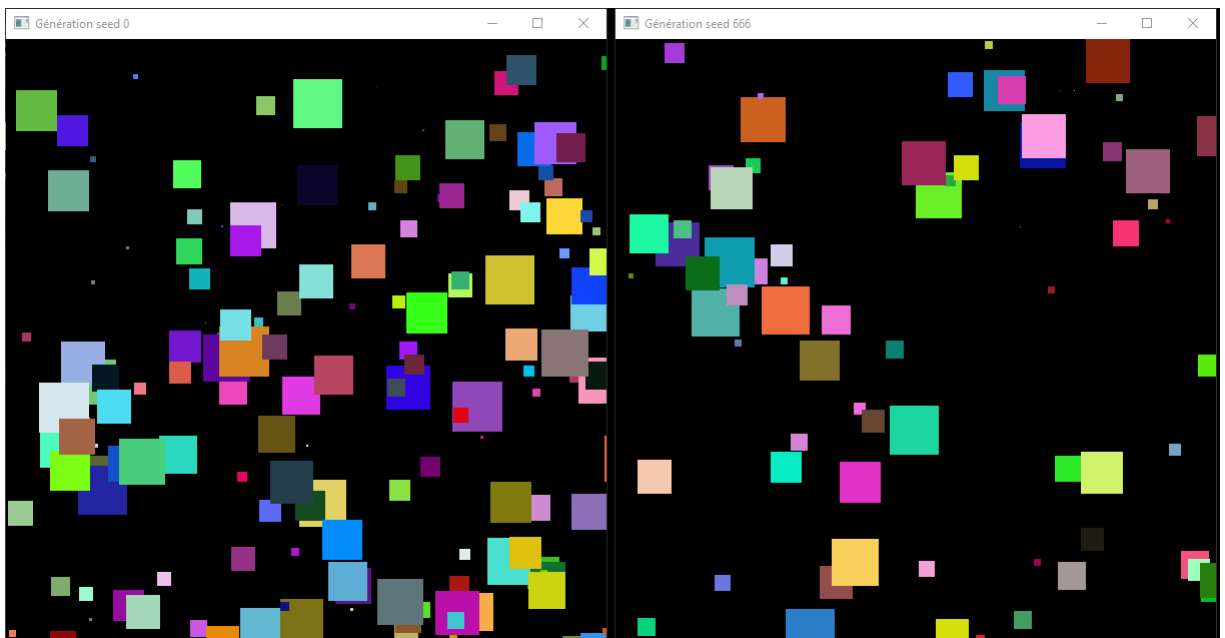
```
Pour 100000 essais :  
1 :      10141      10.141%  
2 :       9949       9.949%  
3 :       9757       9.757%  
4 :       9998       9.998%  
5 :       9856       9.856%  
6 :       9961       9.961%  
7 :      10054      10.054%  
8 :      10191      10.191%  
9 :      10028      10.028%  
10 :     10065      10.065%
```

On voit très bien (et comme on l'a confirmé plus haut avec le test de χ^2) qu'une loi uniforme semble bien être suivie. Nous avons fait aussi le test avec 5 000 000 nombres aléatoires générés et le résultat était encore plus exact :



```
Pour 5000000 essais :  
1 :      499505      9.9901%  
2 :      499793      9.99586%  
3 :      499369      9.98738%  
4 :      499875      9.9975%  
5 :      500830      10.0166%  
6 :      499890      9.9978%  
7 :      499790      9.9958%  
8 :      500681      10.0136%  
9 :      500062      10.0012%  
10 :     500205      10.0041%
```

Afin d'imager l'utilisation d'une graine similaire pour avoir les mêmes résultats, nous avons créé une fonction procédurale générant une image, en utilisant les paramètres énoncés au dessus, en changeant simplement la graine. Voici les images obtenus respectivement pour $X_0 = 0$ et $X_0 = 666$:



Nous avons donc bien toujours la même image générée, peu importe l'ordinateur utilisé, tant qu'on utilise la même graine de départ.

Conclusion

Nous avons vu que les paramètres d'un générateur congruentiel linéaire doivent être choisis précieusement afin de garantir la fiabilité de celui-ci. L'utilisation d'une même graine permet d'avoir les mêmes nombres générés, nous pouvons donc nous demander : comment avoir une graine aléatoirement, afin d'avoir un générateur non prévisible et sécurisé ? Il y a plusieurs solutions comme construire un nombre à partir d'informations très variables concernant l'ordinateur, comme la température du processeur, l'heure actuelle ou encore la place disponible dans la mémoire vive. Une solution très sécurisée serait d'utiliser un générateur reposant sur des phénomènes physiques, tel que l'utilisation d'une antenne radio pour capter des ondes, un souffle dans un micro ou encore une mécanique quantique, qui seraient théoriquement les meilleurs générateurs aléatoires.

Quoiqu'il en soit, il n'existe pas à l'heure actuelle un unique «bon» générateur dont l'utilisation systématique pourrait être recommandée inconditionnellement, la sensibilité d'une simulation aux défauts particuliers d'un procédé de génération étant difficile à apprécier a priori. Il est important d'adopter la bonne attitude vis-à-vis de l'utilisation de générateurs de nombres pseudo-aléatoires pour la simulation

Bibliographie

Donald Knuth, The Art of Computer Programming, Volume 2 : Seminumerical Algorithms, 3rd edition (Addison-Wesley, Boston, 1998).

George Marsaglia, Arif Zaman, Toward a universal random number generator, *Statistics & Probability Letters*, Volume 9, Issue 1, (1990).

J. E. Gentle, Random Number Generation and Monte Carlo Methods , Chapitre 1, (1998).

Chasse J.C., Debouzie D., Utilisation des tests de KIVELIOVITHC et VIALAR dans l'étude de quelques générateurs de nombres pseudo-aléatoires, (1974).

Coveyou R.R., Serial correlation in the generation of pseudo-random numbers, (1960).

Deltour J., Etude d'une distribution de nombres pseudo-aléatoires, (1967).

Devillers R., Dumont J.J., Latouche G., Tests de generateurs pseudo-aléatoires, (1973).

J.J. Claustrioux, Génération et contrôle de validité de nombres pseudo-aléatoires sur ordinateur à mots de 16 bits, (1976).

https://fr.wikipedia.org/wiki/Générateur_congruentiel_linéaire

https://benjaminbillet.fr/media/prng_mathreport.pdf

https://fr.wikipedia.org/wiki/Valeur_p

<http://www.sthda.com/french/wiki/table-de-khi2>

Annexes

FIGURE 1 – Fichier congruentialGenerator.h

```
#include "congruentialGenerator.h"

double congruentialGenerator::next()
{
    seed = (a * seed + c) % m;
    return (double) seed/m;
}
```

FIGURE 2 – Fichier congruentialGenerator.cpp

```
class congruentialGenerator
{
public:
    void setSeed(int newSeed) { seed = newSeed; };
    double next();
private:
    int seed;
    const static unsigned int a = 214013;
    const static unsigned int c = 2531011;
    const static unsigned int m = 2147483648;
};
```

FIGURE 3 – Table des valeurs critiques de la loi du Khi2

k	γ										
	0.995	0.990	0.975	0.950	0.900	0.500	0.100	0.050	0.025	0.010	0.005
1	0.00	0.00	0.00	0.00	0.02	0.45	2.71	3.84	5.02	6.63	7.88
2	0.01	0.02	0.05	0.10	0.21	1.39	4.61	5.99	7.38	9.21	10.60
3	0.07	0.11	0.22	0.35	0.58	2.37	6.25	7.81	9.35	11.34	12.84
4	0.21	0.30	0.48	0.71	1.06	3.36	7.78	9.94	11.14	13.28	14.86
5	0.41	0.55	0.83	1.15	1.61	4.35	9.24	11.07	12.83	15.09	16.75
6	0.68	0.87	1.24	1.64	2.20	5.35	10.65	12.59	14.45	16.81	18.55
7	0.99	1.24	1.69	2.17	2.83	6.35	12.02	14.07	16.01	18.48	20.28
8	1.34	1.65	2.18	2.73	3.49	7.34	13.36	15.51	17.53	20.09	21.96
9	1.73	2.09	2.70	3.33	4.17	8.34	14.68	16.92	19.02	21.67	23.59
10	2.16	2.56	3.25	3.94	4.87	9.34	15.99	18.31	20.48	23.21	25.19
11	2.60	3.05	3.82	4.57	5.58	10.34	17.28	19.68	21.92	24.72	26.76
12	3.07	3.57	4.40	5.23	6.30	11.34	18.55	21.03	23.34	26.22	28.30
13	3.57	4.11	5.01	5.89	7.04	12.34	19.81	22.36	24.74	27.69	29.82
14	4.07	4.66	5.63	6.57	7.79	13.34	21.06	23.68	26.12	29.14	31.32
15	4.60	5.23	6.27	7.26	8.55	14.34	22.31	25.00	27.49	30.58	32.80