Plan

Plan

Générateurs aléatoires

Un générateur aléatoire sur un ensemble E d'objets est une fonction g à valeur dans E non déterministe. La valeur renvoyée par g n'est pas nécessairement la même d'un appel à un autre.

On cherche dans la mesure du possible à faire en sorte que, étant donné un ensemble E d'objets, tout objet de E ait la même probabilité d'être renvoyé par le générateur aléatoire g. On parle alors de générateur aléatoire uniforme.

Par exemple, si E est l'ensemble des faces d'un dé à six faces, le procédé g qui consiste à lancer le dé et renvoyer la face visible constitue un générateur aléatoire uniforme (et non uniforme si le dé est pipé).

Intérêt des générateurs aléatoires

Les générateurs aléatoires offrent de nombreuses applications. Ils sont utilisés pour :

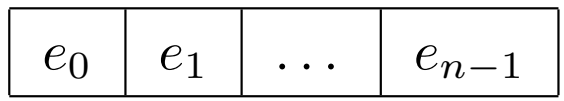
1. les **méthodes de Monte-Carlo** pour obtenir des solutions approchées à des problèmes ;
2. **tester des algorithmes** en générant des entrées de manière aléatoire (permutations, listes, arbres, graphes, *etc.*) ;
3. **rompre le caractère déterministe** d'un programme en y incluant des éléments imprévisibles (dans les jeux par exemple) ;
4. générer des **mots de passe** ou des **clés de chiffrement**.

Universalité des générateurs aléatoires d'entiers

Un générateur aléatoire d'entiers d'ordre n est un générateur aléatoire sur l'ensemble $\{0, 1, 2, \ldots, n-1\}$.

Pour construire un générateur aléatoire sur un ensemble E d'objets, il suffit en pratique d'avoir un générateur aléatoire d'entiers d'ordre $\#E$.

En effet, il suffit de placer les objets (ou mieux : les adresses des objets) de E dans un tableau



d'appeler g et de renvoyer l'objet du tableau figurant à l'indice spécifié par l'entier généré par g.

Ainsi, pour faire de la génération aléatoire d'objets, il est suffisant (en 1re approximation) de **savoir construire des générateurs aléatoires d'entiers** d'un ordre suffisamment grand.

Réduire l'ordre d'un générateur aléatoire d'entiers

L'opérateur « **modulo** » offre un moyen très simple pour **réduire l'ordre** d'un générateur aléatoire d'entiers.

En effet, supposons que l'on dispose d'un générateur aléatoire d'entiers g d'ordre n. On souhaite obtenir un générateur aléatoire d'entiers h d'ordre k avec $1 \leqslant k \leqslant n$. On pose pour cela

$$h := g \mod k.$$

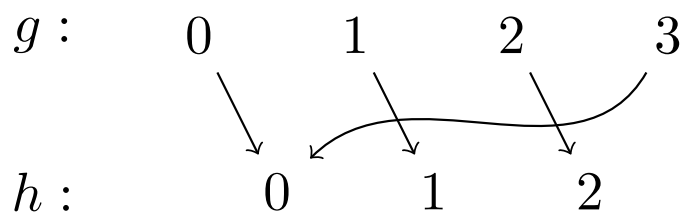
Il est clair que les entiers générés par h sont dans l'ensemble $\{0, \ldots, k-1\}$. On a ainsi réduit l'ordre du générateur g.

On dit que h est la réduction à l'ordre k de g.

Réduction de l'ordre et uniformité

Supposons que g soit un générateur aléatoire uniforme d'ordre $n := 4$ et considérons la réduction h à l'ordre $k := 3$ de g.

On a la situation suivante :



Il y a ainsi deux manières de générer 0 par h (probabilité de $\frac{1}{2}$) alors qu'il n'y en a qu'une seule pour 1 et 2 (probabilités resp. de $\frac{1}{4}$).

Ceci montre que le générateur aléatoire h perd la propriété d'uniformité.

Mesure de la non uniformité de la réduction

Soient g un générateur aléatoire uniforme d'ordre n et h la réduction de g à l'ordre k, où $k \leqslant n$.

Notons $\mathrm{gen}_h(i)$ le nombre de manières de générer l'entier $i \leqslant k-1$ par h. Alors, on voit facilement que

$$\mathrm{gen}_h(i) = \begin{cases} \lfloor \frac{n}{k} \rfloor + 1 & \text{si } i \in \{0, \ldots, (n \mod k) - 1\} \\ \lfloor \frac{n}{k} \rfloor & \text{sinon.} \end{cases}$$

La probabilité $\mathbb{P}_h(i)$ de générer l'entier $i \leqslant k-1$ par h vérifie

$$\mathbb{P}_h(i) = \frac{\mathrm{gen}_h(i)}{n}.$$

Les entiers i de l'ensemble $\{0, \ldots, k-1\}$ ont ainsi des probabilités différentes d'être générés. Ils se divisent en deux classes :

1. **les petits**, de 0 à $(n \mod k) - 1$, avec une probabilité plus grande ;
2. **les grands**, de $(n \mod k)$ à $k-1$, avec une probabilité plus petite.

Mesure de la non uniformité de la réduction

Comparons numériquement les probabilités de génération des petits et des grands nombres pour différentes valeurs de n et de k :

n	k	Prob. des petits	Prob. des grands	Rapport
4	3	0.5	0.25	2
400	300	0.005	0.0025	2
400	200	0.005	0.005	1
400	101	0.01	0.0075	$\simeq 1.3333$
400	99	0.0125	0.01	1.25
500	99	0.012	0.01	1.2
5000	99	0.0102	0.01	1.02
50000	99	0.01012	0.0101	$\simeq 1.00198$

On observe que plus k est **petit** par rapport à n, plus h se rapproche d'un générateur **uniforme**. Lorsque k est un diviseur de n, l'uniformité est immédiate.

En pratique, la réduction est considérée comme préservant l'uniformité, à condition de partir d'un générateur aléatoire d'entiers uniforme g d'ordre le plus grand possible.

Plan

Machines et aléatoire

Un ordinateur est par essence une machine déterministe : le résultat d'un programme est toujours le même s'il est exécuté deux fois dans les mêmes conditions.

Il est ainsi impossible d'écrire un programme qui implante un générateur aléatoire d'entiers.

La seule chose qu'il est possible de faire est de programmer une fonction qui **semble** le plus possible se comporter comme un générateur aléatoire. On parle alors de générateur pseudo-aléatoire.

Les entiers qu'un générateur pseudo-aléatoire génère lorsqu'on l'appelle plusieurs fois de suite forme une suite. On appelle cette suite une suite d'entiers pseudo-aléatoires.

Principe de fonctionnement

Un générateur pseudo-aléatoire g d'ordre n fonctionne de la manière suivante :

- il dispose d'une graine $g_1, g_2, \ldots, g_\ell$, qui est une suite d'entiers ;
- lorsqu'on l'appelle pour la r^{e} fois, il renvoie un entier $g_{\ell+r}$ calculé à partir des entiers $g_1, \ldots, g_\ell, \ldots, g_{\ell+r-1}$ précédemment calculés, selon une règle spécifiée.

Ainsi, g produit une suite d'entiers

$$g_{\ell+1}, g_{\ell+2}, g_{\ell+3}, \cdots$$

à partir de la graine dans laquelle figurent les **données initiales** et de la règle qui explique comment générer le **prochain entier** de la suite en se basant sur les **entiers précédemment générés**.

Qualités d'un générateur pseudo-aléatoire

Un bon générateur pseudo-aléatoire produit une suite d'entiers qui semble aléatoire.

Il est possible de mesurer la qualité d'un générateur aléatoire conformément à plusieurs critères :

1. **uniformité** de la génération ;
2. ses **périodes** (longueurs des cycles de génération) ;
3. **sensibilité à la graine** (existence de mauvaises graines) ;
4. **corrélation** entre un entier engendré et les précédents ;
5. **analyse de la distribution** en plusieurs dimensions.

Méthode du carré médian

Le générateur pseudo-aléatoire g utilisant la méthode du carré médian fonctionne de la manière suivante.

La graine de ce générateur est un entier compris entre 0 et 9999. Si g_{i-1} est l'entier précédemment généré (ou bien la graine), le prochain entier généré est

$$g_i := \left(\left\lfloor \frac{g_{i-1}}{10} \right\rfloor \mod 100 \right)^2.$$

On obtient p.ex. les suites

Graine	Suite
1234	529, 2704, 4900, 8100, 100, 100, ...
3733	5329, 1024, 4, 0, 0, ...
9999	9801, 6400, 1600, 3600, 3600, ...

C'est un **mauvais générateur pseudo-aléatoire**. Les périodes sont bien trop courtes et il y a des mauvaises graines (comme 0).

Méthode additive

Le générateur pseudo-aléatoire g utilisant la méthode additive fonctionne de la manière suivante.

On se fixe un ordre $n \geqslant 1$. La graine de ce générateur est un triplet (g_1, g_2, g_3) d'entiers compris entre 0 et $n-1$. La génération de g_i dépend des trois entiers précédemment générés g_{i-1}, g_{i-2} et g_{i-3}. Il est défini par

$$g_i := (g_{i-1} + g_{i-2} + g_{i-3}) \mod n.$$

On obtient p.ex. les suites

n	Graine	Suite
211	$(1, 1, 1)$	3, 5, 9, 17, 31, 57, 105, 193, 144, 20, 146, 99, 54, 88, 30, 172, 79
3099	$(1, 1, 1)$	3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 964, 1275, 1349
1347	$(600, 31, 1)$	632, 1263, 1148, 349, 66, 216, 631, 913, 413, 610, 589, 265, 117

Ce générateur pseudo-aléatoire est bien meilleur que le précédent mais échoue à de nombreux tests statistiques.

Générateurs congruentiels linéaires

Un générateur pseudo-aléatoire g congruentiel linéaire fonctionne de la manière suivante.

On se fixe un ordre $n \geqslant 1$ ainsi que deux entiers positifs a et b. La graine de ce générateur est un entier compris entre 0 et $n-1$. La génération de g_i dépend de l'entier précédemment généré g_{i-1} et est défini par

$$g_i := (a \times g_{i-1} + b) \mod n.$$

On obtient p.ex. les suites

n	a	b	Graine	Suite
128	3	1	1	4, 13, 40, 121, 108, 69, 80, 113, 84, 125, 120, 105, 60
128	7	1	1	8, 57, 16, 113, 24, 41, 32, 97, 40, 25, 48, 81, 56, 9, 64
2^{32}	1664525	1013904223	1	1015568748, 1586005467, 2165703038, 3027450565

Sous réserve de choisir des bons paramètres n, a et b, ce générateur pseudo-aléatoire est plutôt bon. Il est très utilisé en pratique.

Plan

Les fonctions `rand` et `srand`

La fonction

```
int rand();
```

du module `stdlib` implante un générateur pseudo-aléatoire d'ordre `RAND_MAX + 1`. C'est un générateur congruentiel linéaire. À chaque appel, il renvoie le prochain entier de la suite.

La graine de ce générateur peut-être affectée par la fonction

```
void srand(unsigned int g1);
```

Par défaut, la graine du générateur est `1`.

Modèle d'utilisation de `rand` et de `srand`

Tout projet qui utilise la fonction `rand` doit avoir une fonction principale `main` de la forme ci-contre.

```c
/* Main.c */
...
#include <stdlib.h>
#include <time.h>
...
int main() {
    ...
    srand(time(NULL));
    ...
}
```

L'appel à `srand` doit être fait le plus tôt possible.

La fonction `time_t time (time_t* timer);` du module `time`, lorsque appelée avec l'argument `NULL` renvoie le temps écoulé en secondes depuis le 1er janvier 1970 à minuit, UTC.

Considérer cette valeur est donc un bon moyen d'initialiser la graine du générateur.

Attention (*point très important*) : il est totalement erroné d'initialiser dans un même projet deux fois la graine. Le seul appel à `srand` doit figurer dans `main` et nulle part ailleurs.

Implantation avec variable globale

Une implantation possible des fonctions `rand` et `srand` s'appuie sur une **variable globale** pour mémoriser la dernière valeur générée.

```c
/* RandPerso.h */
#ifndef __RAND_PERSO__
#define __RAND_PERSO__

    #define RAND_PERSO_ORDRE 32767
    #define RAND_PERSO_A 1024
    #define RAND_PERSO_B 1
    #define RAND_PERSO_GRAINE 1

    int rand_perso();
    void srand_perso(int graine);

#endif
```

```c
/* RandPerso.c */
#include "RandPerso.h"

unsigned int valeur_rand =
    RAND_PERSO_GRAINE;

int rand_perso() {
    valeur_rand =
        (valeur_rand * RAND_PERSO_A
            + RAND_PERSO_B)
        % RAND_PERSO_ORDRE;
    return valeur_rand;
}

void srand_perso(int graine) {
    valeur_rand = graine;
}
```

Implantation avec variable statique

Dans cette implantation, on utilise une **variable statique** pour mémoriser la dernière valeur générée.

```c
/* RandPerso.h */
#ifndef __RAND_PERSO__
#define __RAND_PERSO__

    #define RAND_PERSO_ORDRE 32767
    #define RAND_PERSO_A 1024
    #define RAND_PERSO_B 1
    #define RAND_PERSO_GRAINE 1

    int rand_perso();

#endif
```

```c
/* RandPerso.c */
#include "RandPerso.h"

#include <time.h>

int rand_perso() {
    static int premier_appel = 1;
    static unsigned int valeur_rand;
    if (premier_appel) {
        valeur_rand = time(NULL);
        premier_appel = 0;
    }
    valeur_rand =
        (valeur_rand * RAND_PERSO_A
            + RAND_PERSO_B)
        % RAND_PERSO_ORDRE;
    return valeur_rand;
}
```

L'initialisation ne se fait qu'au 1er appel et l'utilisateur n'a pas à la faire explicitement. Avec cette méthode, il n'est pas possible de fixer la graine.

Retrouver le déterminisme

Pour pouvoir reproduire les résultats fournis par un programme utilisant la fonction `rand`, il est nécessaire de proposer un mode de fonctionnement déterministe dans lequel la graine a été fixée.

Pour cela, on met en place dans le module principal d'un projet et dans sa fonction `main` le mécanisme suivant :

```
/* Main.c */
...
#define DETERMINISTE
...
int main() {
    ...
#ifdef DETERMINISTE
    srand(0);
#else
    srand(time(NULL));
#endif
    ...
}
```

De cette manière, on peut imposer un comportement déterministe (**reproductible**) à l'exécution du projet en gardant la définition de la macro `DETERMINISTE`.

Pour éviter ce comportement, il suffit de commenter cette macro-définition.

Meilleure solution : utiliser l'option `-D` de `gcc`. Il suffit de supprimer la ligne `#define DETERMINISTE` et de compiler avec `-DDETERMINISTE` pour définir la macro en question.