PROG
TP#4

2021-W41

Générateurs & permutations aléatoires

Le but de cet exercice est d'implémenter un « bon » générateur pseudo-aléatoire, ainsi que deux algorithmes permettant de générer une permutation aléatoire des entiers de 1 à N (ceci se généralisant facilement à n'importe quel ensemble de taille N).

Q.1 : RC4 est un générateur de nombres pseudo-aléatoires développé dans les années 80 pour des applications cryptographiques. Bien qu'il soit trop faible pour apporter une sécurité suffisante, il est néanmoins simple à implémenter et offre des propriétés statistiques raisonnables ce qui en fait un choix acceptable pour des applications non critiques. Le pseudo-code de RC4 est le suivant :

```
/****************************************
initialisation//////////////////////////
input: uint8_t S[256] = [0,...,255]
input: uint8_t seed[16]
input: i,j
****************************************/
j = 0;
for (i = 0; i < 256)
{
        j = (j + S[i] + seed[i mod 16]) mod 256;
        swap(S[i], S[j]);
}
i = 0;
j = 0;

/****************************************
génération d'un octet//////////////////////
input: uint8_t S[256], i, j
output: un octet pseudo-aléatoire
****************************************/
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
swap(S[i], S[j])
return S[(S[i] + S[j]) mod 256]
```

Vous devez implémenter ce générateur dans un fichier `rc4_prng.c` et exposer une interface à haut niveau dans un fichier `rc4_prng.h`. Plus précisément, vous devez (au moins) implémenter les fonctions suivantes :

— `void rc4_init(uint8_t seed[16]);`, qui effectue l'initialisation décrite ci-dessus ;

— `uint64_t rc4_random(void);`, qui retourne 64 bits d'aléa générés comme ci-dessus.

Vous ferez attention aux points suivants :

— Si aucun appel d'initialisation n'a été fait avant un appel à `rc4_random()`, une initialisation par défaut *aléatoire et de bonne qualité statistique* doit être effectuée, par exemple en utilisant `/dev/urandom` ou `getrandom()` (ou sous BSD ou MacOS, `arc4random()`).

— À l'inverse, si une initialisation a été effectuée explicitement via `rc4_init()`, celle-ci ne doit pas être « écrasée ».

Ces deux points peuvent tout deux être résolus grâce à une utilisation judicieuse de variables globales (locales au fichier `rc4_prng.c`, par exemple déclarée avec le qualifieur de type `static`).

N'oubliez pas de tester vos fonctions !

Q.2 : En supposant que les octets générés par RC4 suivent une loi uniforme, implémentez une génération d'aléa *uniforme* sur $[\![0, 999\,999\,999]\!]$:

1. Expliquez pourquoi simplement renvoyer `rc4_random() % 1000000000` ne produit pas une distribution uniforme.
2. Proposez une solution.
3. Montrez que si la distribution obtenue en 1 n'est pas uniforme, elle en est néanmoins « très proche ».
4. Implémentez la fonction `rc4_random999999999(void);` de la façon qui vous semble la plus appropriée.

Q.3 : On souhaite maintenant générer une permutation aléatoire de $[\![1, N]\!]$ de la façon suivante : 1) on initialise un tableau avec N paires (r_i, i) où i prend les N valeurs dans $[\![1, N]\!]$ et où les r_i sont des nombres tirés uniformément dans un ensemble de taille N' [1] ; 2) on trie la liste suivant les valeurs des r_i ; 3) on retourne les valeurs i dans l'ordre trié [2]. On peut montrer que quand $N' \approx N^2$, la distribution des permutations générées par cet algorithme est proche de l'uniforme.

Implémentez l'algorithme ci-dessus pour $N < 2^{64}$ dans une fonction :

```
void shuffle1(uint64_t p[], uint64_t nelem);
```

où `p` est alloué en dehors de la fonction, et devra contenir à l'issue de son exécution une permutation des entiers entre 1 et `nelem`. Cette fonction devra utiliser (de façon interne) la structure :

```
struct uints
{
        uint64_t key;
        uint64_t dat;
};
```

pour représenter les paires d'éléments et la fonction `qsort` de la bibliothèque standard pour effectuer le tri.

N'oubliez pas de tester votre fonction (par exemple sur de petits cas).

1. Par exemple pour $N = 4$, $N' = 16$, un tableau possible serait $[(7, 1); (3, 2); (13, 3); (11, 4)]$.
2. Pour le tableau de l'exemple précédent, on obtiendrait donc $[2, 1, 4, 3]$.

Q.4 :

1. Mesurez le temps d'exécution (par exemple en utilisant `time` en ligne de commande) pour des permutations de 1 000 000, 10 000 0000, 20 0000 000, 40 000 000, 80 000 000, 160 000 000 éléments.
2. Refaites les mêmes tests avec une autre implémentation du tri (si disponible sur votre machine), par exemple `heapsort`.

Pour chacun des points ci-dessus, quels commentaires pouvez-vous faire ?

On s'intéresse maintenant à l'algorithme suivant, donné en pseudo-code :

```
/*****************************************
input: T = [1,...,N]
*****************************************/
for (i = 0; i < N - 1)
{
        s = uniform_random(i, N-1);
        swap(T[i], T[s]);
}
```

Q.5 : Implémentez l'algorithme ci-dessus dans une fonction `shuffle2` similaire à `shuffle1`.
Mesurez le temps d'exécution comme dans **Q.2** et comparez les résultats. Quelles raisons peuvent expliquer les différences ?