PROG
TP#4

2019-09-25

Permutations aléatoires

Le but de cet exercice est d'implémenter 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).

On s'intéresse dans un premier temps à l'algorithme suivant : 1) on initialise un tableau avec N paires (r_i, i), $i \in [\![1, N]\!]$, où les r_i sont des nombres tirés au hasard ; 2) on trie la liste suivant les valeurs des r_i ; 3) on retourne les valeurs i dans l'ordre trié.

Q.1 : Implémentez l'algorithme ci-dessus dans une fonction de test :

```
void test_shuffle1(unsigned nelem);
```

en utilisant la structure :

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

pour les paires d'éléments et la fonction `qsort` de la bibliothèque standard pour effectuer le tri. (Vous n'êtes pas obligés de retourner explicitement la permutation, et vous pouvez donc vous arrêter après l'étape 2)

1. Choisissez avec précaution votre générateur aléatoire, et faites attention à son initialisation.
2. Intuitivement, quelle condition doit vérifier la taille de `key` par rapport à `nelem` pour que cet algorithme donne de bons résultats statistiques ?
3. Testez votre fonction sur de petits cas (par ex. 10 éléments).

Q.2 :

1. Mesurez le temps d'exécution (par ex. 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 ex. `heapsort`.
3. Refaites les mêmes tests en modifiant la structure `uints` pour qu'elle contienne deux `uint32_t`.

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.3 : Implémentez l'algorithme ci-dessus dans une fonction `test_shuffle2` similaire à `test_shuffle1`, en faisant particulièrement attention à la génération d'aléa.

Mesurez le temps d'exécution comme dans **Q.2** et comparez les résultats. Quelles raisons algorithmiques et architecturales peuvent expliquer les différences ?

Q.4 (bonus) : 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 les applications non critiques. Le pseudo-code de RC4 est le suivant :

```
/****************************************
initialisation//////////////////////////
input: uint8_t S[256] = [0,...,255]
input: uint8_t seed[8]
input: i,j
****************************************/
j = 0;
for (i = 0; i < 256)
{
        j = (j + S[i] + seed[i mod 8]) 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]
```

1. Implémentez ce générateur, en fournissant une fonction d'initialisation et une ou plusieurs fonctions de génération d'aléa (choisissez bien la taille des aléas à retourner en fonction de vos besoins).

2. Remplacez votre génération d'aléa des questions précédentes avec ce générateur, et comparez les résultats.
3. En supposant que les octets générés par RC4 suivent une loi uniforme, implémentez une génération d'aléa *uniforme* sur $[\![0, 240]\!]$.