Génération de permutations

Licence MIAGE

October 2, 2013

On s'intéresse dans ce TP à l'écriture d'un paquetage de manipulation de permutations. Etant donné un entier N strictement positif, une permutation de taille N est définie comme une liste de N éléments non nécessairement ordonnés compris entre 1 et N, sans doublon. Par exemple, une permutation de taille 5 est $(1, 5, 4, 2, 3)$ ou bien $(1, 2, 3, 5, 4)$. On appelle permutation identité la permutation dont les éléments sont triés par ordre croissant. Par exemple, pour une taille de 5 la permutation identité est $(1, 2, 3, 4, 5)$.

Le type permutation sera défini ainsi

```
type PERMUTATION is array (POSITIVE range <>) of POSITIVE;
```

On voudra disposer des primitives suivantes dans le paquatage en plus de celles demandées dans les questions ci-dessous:

```
-- genere la permutation identite
function identite (N : in POSITIVE) return PERMUTATION;

-- affiche la permtation sous la forme (1 2 4 5 3)
procedure affiche (p: in PERMUTATION);
```

Dénombrement

Question Pour N donné, combien existe-t-il de permutations ? Justifiez, pour cela vous pouvez donner une équation de récurrence par exemple.

Génération de permutation aléatoire

On veut disposer d'une procédure dont la signature est la suivante :

```
procedure permutation_aleatoire (p : in out PERMUTATION);
```

permettant la génération aléatoire d'une permutation.

Pour obtenir un nombre aléatoire en ADA vous devez procéder ainsi (on admet que `p` est de type PERMUTATION) :

```
    ....
    -- definir le sous-intervalle sur lequel on veut tirer un nombre aleatoire
    subtype INTERVALLE is POSITIVE range p'range;
    -- creer un paquetage de fonctions de generation de nombres aleatoires
    package RANDOM_INTERVALLE is new Ada.Numerics.Discrete_Random(INTERVALLE);
    use Random_Intervalle;
    -- declarer la graine initialisant le generateur
    seed : GENERATOR;
    ....
begin
    ....
    -- initialise le generateur
    reset(seed);
    ....
    -- met dans n un nombre aleatoire tire sur INTERVALLE
    n := random(seed);
    ....
```

Question

1. Trouver un algorithme.
2. Implémenter et tester.
3. Donner la complexité en espace et en temps.

Question Donnez un nouvel algorithme de complexité linéaire si ce n'est pas le cas de votre algorithme précédent.

Question Montrez que votre algorithme linéaire génére de manière équiprobable toutes les permutations en admettant que le générateur de nombres aléatoires donne également chaque entier avec la même probabilité. Vous pouvez procéder par récurrence.

Génération de toutes les permutations

Le problème suivant est d'être capable de générer l'ensemble des permutations qui existent pour une taille N donnée. On s'intéresse à cela afin d'évaluer la complexité moyenne d'algorithme.

Imaginer un algorithme qui réalise ce travail n'est pas aisé. Lorsque nous énumérons à la main un ensemble de permutations à partir de la permutation identité, nous procédons de manière systématique. C'est ce processus que l'algorithme donné après, capable de donner la permutation suivante, reproduit.

Mais avant cela nous aurons besoin d'une procedure annexe :

```
procedure Foo (T : in out PERMUTATION; P, Q : POSITIVE) is
   Tmp : POSITIVE;
begin
   for I in P .. Q - 1 loop
      for J in I+1 .. Q loop
         if T(I) > T(J) then
            Tmp  := T(J);
            T(J) := T(I);
            T(I) := Tmp;
         end if;
      end loop;
   end loop;
end Foo;
```

Question Que fait `Foo` ?

Algorithme de génération de la permutation suivante :

1. cherche le plus grand indice i tel que $p(i) < p(i+1)$
2. chercher l'indice j tel que $p(j)$ est le plus petit élément parmi $p(i+1)..p(N)$ vérifiant $p(j) > p(i)$
3. échanger les éléments i et j
4. exécuter `Foo` sur $p(i+1..N)$

Question

1. dérouler l'algorithme à la main pour générer quelques permutations suivantes de $(1, 2, 3, 4)$
2. implémenter et tester l'algorithme proposé
3. donner la complexité en espace et en temps de l'algorithme dans le meilleur et dans le pire cas
4. en remarquant que `Foo` est toujours appliqué sur une tranche de permutation **déja anti-trié** ("trié dans l'ordre inverse"), proposer un algorithme linéaire pour `Foo`.