Labo 2

Groupe:
- Valentin Ricard
- Alexandre Philibert

Idée de l'implémentation

En suivant le pseudo-code fourni, on se rend compte que la fonction permet de générer les `k` permutations de manière déterministique, avec $0 \leq k \leq n!$ ou n est la taille de la séquence.

Il est ensuite possible de répartir la tache de travail en séparant l'ensemble $[0, n!]$ en `nb_threads` sous-ensembles, et en laissant chaque thread gérer leur ensemble, sans chevauchement.

Pour gérer la fin des threads sans calculer la totalité des possibilités (ou attendre la fin de l'execution de chacun des threads), il est possible d'ajouter le snippet suivant dans la boucle d'execution du thread :

```
if (pManager->finished || PcoThread::thisThread()->stopRequested())
{
    return;
}
```

En termes de gestion de la concurrence, cette opération est raisonnable, car nous sommes garanti / nous acceptons que:

- Une seule permutation est valide (pas de course pour l'écriture), et plusieurs threads ne peuvent pas tomber sur la même permutation valide. Cela nous oblige a limiter le nombre de threads total pour les petites séquences ($\text{nbThreads} = \max(\text{nbThreads}, n!)$)
- Une execution supplémentaire de la boucle est moins coûteuse que le coût d'une lecture atomique pour chaque itération.

Valeurs testées

Sequence length	1	1	10
Seed	0	0	4
Number of threads	1	5	24
Result	✓	✓	✓

Lors de nos tests, nous avons découvert le cas ou $\text{nbThreads} > \text{len}(\text{Sequence})!$. Le document présent au dessus tient compte de cette découverte.