

TP n°9 - Récursivité et tableaux.

Tous les codes de ce TP devront être remis sur Moodle - TP9 - avant jeudi 01/12 23h59

Pour ce TP, sur les machines du lycée, il faudra travailler dans un répertoire TP9 situé en dehors du dossier Documents

Penser à déplacer à nouveau votre dossier au bon endroit un peu avant la fin de l'heure, et à le copier sur votre clé USB. Ne le faites pas au dernier moment.

Il n'est pas possible de générer des nombres réellement aléatoires avec des algorithmes déterministes. Toutefois, il existe des algorithmes permettant de générer des nombres pseudo-aléatoires. Les algorithmes de génération de nombre pseudo-aléatoires fonctionnent en utilisant des suites récurrentes ayant des comportements très particuliers¹. Ces suites sont initialisées avec une graine *seed* qui doit être différente à chaque exécution pour que la série de valeurs obtenues soit différente à chaque exécution.

En C, la bibliothèque standard `stdlib` contient deux fonctions permettant cela :

`void srand(unsigned int seed)` : cette fonction permet d'initialiser l'algorithme générateur de nombres pseudo-aléatoires en lui fournissant une nouvelle graine à chaque nouvelle exécution du code. Cette fonction n'est appelée qu'une fois au début du code pour initialiser l'algorithme. Cette graine doit cependant être différente à chaque nouvelle exécution, sinon, les différents appels à `rand` renverrons toujours les mêmes suites de valeurs aléatoires. En général, pour garantir le fait que `seed` est différent à chaque exécution, on initialise l'algorithme en utilisant le résultat de la fonction `time`, qui se trouve dans la bibliothèque `time.h`, et qui donne le nombre de secondes écoulées depuis le 1er janvier 1970.

```
srand( time(NULL) )
```

`int rand(void)` : pour utiliser cette fonction, il faut préalablement avoir initialisé l'algorithme en début de code avec `srand`. Chaque appel de la fonction `rand`, sans argument, retourne un nombre entier pseudo-aléatoire est compris entre 0 et une valeur maximale appelée `RAND_MAX`, définie dans la bibliothèque.

En OCaml, il existe un module `Random` dédié à la génération de nombres pseudo-aléatoires. Le choix de la graine permettant d'initialiser l'algorithme est automatiquement pris en charge par ce module par un unique appel à

```
Random.self_init()
```

en début de programme.

Ensuite, des appels à la fonction

```
Random.int n
```

permettent de générer des entiers pseudo-aléatoires compris entre 0 et $n - 1$ inclus

Pour tous les programmes qui suivent, on réfléchira sur papier, et, dès que l'esprit est moins clair, on s'attachera à bien définir les entrées et les sorties et à écrire les algorithmes avant de se lancer dans la programmation.

On veillera par ailleurs à appliquer les principes de la **programmation défensive** et à **tester abondamment** et intelligemment les fonctions (notamment en essayant de couvrir toutes les alternatives pouvant survenir).

Exercice 1 (Liste de valeurs aléatoires).

1. Dans un fichier `tri_insertion.ml`, créer une fonction OCaml **récursive** appelée `genere_liste_aleatoire` qui génère une **liste** d'entiers de taille n donnée dont les valeurs sont des entiers pseudo-aléatoires compris entre deux valeurs données a et b . On réfléchira sur papier aux entrées/sorties, aux spécifications et aux expressions mathématiques. On n'hésitera pas à faire un dessin et de petits tests sur papier ou dans le toplevel. Bien réfléchir mathématiquement à comment limiter la valeur aléatoire pour qu'elle reste entre a et b .
2. Tester cette fonction. Appliquer les principes de la programmation défensive.

1. https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_nombres_pseudo-al%C3%A9atoires

Exercice 2 (Tableau de valeurs aléatoires).

1. Dans un fichier `tri_insertion.c`, créer un squelette de code de manipulation de tableaux en copiant ou en codant à nouveau rapidement la fonction d'affichage d'un tableau codée dans un précédent TP.
2. On imagine que l'on peut récupérer un entier aléatoire r entre 0 et une valeur `RAND_MAX`. Écrire une formule mathématique permet d'utiliser r pour générer un nombre aléatoire **entier** entre a et b , $a < b$. *Indication : Deux méthodes sont possibles. Pour l'une d'entre elles, on utilisera le fait que tous les nombres réels compris entre a et b s'écrivent $a + \alpha \times (b - a)$ où α est un réel compris dans l'intervalle $[0, 1]$.*^a
3. En déduire une fonction `genere_tab_aleatoire` en langage C qui génère un **tableau** d'entiers de taille n donnée dont les valeurs sont des entiers pseudo-aléatoires compris entre deux valeurs a et b , $a < b$. *Indication : bien lire la description de `srand` ci-dessus. On pourra utiliser le transtypage (levez la main si vous ne vous rappelez plus de la signification de ce mot...) pour forcer une division flottante et/ou pour transformer un flottant en son plus proche entier inférieur.* Tester.

a. Pour les plus à l'aise, on réfléchira sur papier en termes statistiques à la probabilité de chaque valeur dans ce contexte et aux éventuels modifications ou garde-fous à mettre en place.

Exercice 3 (Tri par insertion).

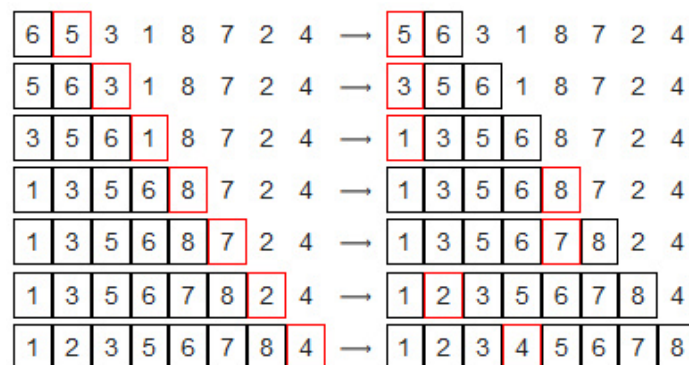
Vous avez travaillé sur l'algorithme du tri par insertion dans le DS n°2.

Le tri par insertion est le tri « du joueur de carte ». On parcourt les éléments du tableau. Pour chaque valeur, on va la replacer (l'insérer) au bon endroit parmi le sous tableau de gauche.

Pour cela, on **décale les éléments de gauche d'un cran vers la droite jusqu'à ce que la case correspondant à la bonne place de la valeur soit libérée.**

Puis on passe à la valeur suivante et on la replace au bon endroit dans le sous tableau de gauche...etc et ceci jusqu'à avoir réinséré tous les éléments.

Voici ci-dessous les étapes de l'algorithme du tri par insertion sur le tableau constitué des valeurs 6, 5, 3, 1, 8, 7, 2, et 4, dans cet ordre.



On montre uniquement le résultat de l'insertion de chaque élément sans montrer tous les décalages réalisés.

Par exemple, lors de l'insertion du 2 à l'avant dernière étape, pour placer le 2 au bon endroit, on a décalé vers la droite le 8, puis le 7, puis le 6, puis le 5, puis le 3.

1. Réécrire rapidement l'algorithme sous forme itérative en pseudo-code sur papier
2. Implémenter en C dans le fichier `tri_insertion.c` une fonction **iterative tri_insertion** qui implémente cet algorithme. Tester abondamment votre fonction en utilisant la fonction `genere_tab_aleatoire`.

Exercice 4.

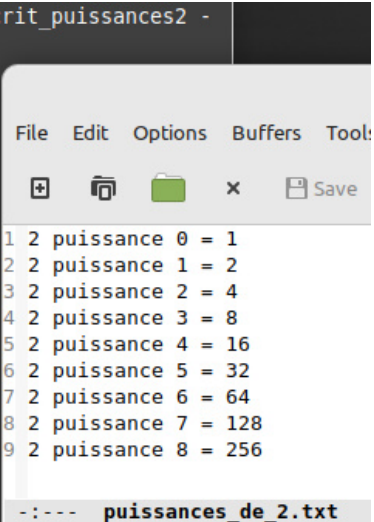
Le tri par insertion consiste en fait à réinsérer chaque valeur, l'une après l'autre, dans la suite des valeurs précédentes déjà triées.

1. Écrire en pseudo-code sur papier une version récursive de l'algorithme de tri par insertion. On suppose qu'on dispose d'un algorithme `insere` qui est capable d'insérer une valeur dans une liste déjà triée par ordre croissant.
2. Dans le fichier `tri_insertion.ml`, implémenter en OCaml la fonction `insere` sous forme **récursive**.
3. Implémenter maintenant l'algorithme de tri par insertion en OCaml dans le fichier `tri_insertion.ml`.
4. Tester le bon fonctionnement de votre algorithme récursif en utilisant la fonction `genere_liste_aleatoire`

Exercice 5 (Exercice sur les entrées/sorties I/O).

1. Écrire un code `ecrit_puissances2.c` qui crée un fichier texte ASCII `puissances_de_2.txt` dans le **répertoire courant**. Ce fichier contient la liste des puissances de 2 jusqu'à 2^n , où n est un paramètre d'entrée fourni par l'utilisateur. Voici ci-dessous un résultat d'exécution, avec la forme de fichier texte souhaitée.

```
golivier@ordiprof:~/doc/MP11/TP9$ gcc escrit_puissances2.c -o escrit_puissances2 -Wall
golivier@ordiprof:~/doc/MP11/TP9$ ./escrit_puissances2 8
golivier@ordiprof:~/doc/MP11/TP9$ cat puissances_de_2.txt
2 puissance 0 = 1
2 puissance 1 = 2
2 puissance 2 = 4
2 puissance 3 = 8
2 puissance 4 = 16
2 puissance 5 = 32
2 puissance 6 = 64
2 puissance 7 = 128
2 puissance 8 = 256
golivier@ordiprof:~/doc/MP11/TP9$ emacs puissances_de_2.txt &
[5] 17525
golivier@ordiprof:~/doc/MP11/TP9$
```



2. Tester le code précédent.
3. (A la maison) Améliorer le code pour que l'utilisateur puisse donner un autre chemin pour la création de son fichier. *Indication : bien appliquer les principes de la programmation défensive en vérifiant que le nouveau nom/chemin fourni par l'utilisateur s'est bien révélé accessible !*
4. Écrire un code `supprime-e.c` qui lit le fichier `exemple.txt` et le recopie dans un autre fichier appelé `exemple-sans-e.txt` en enlevant tous les e. *Indication : Nul besoin de stocker. On utilisera deux descripteurs de fichiers en même temps. On lira un à un les caractères du premier fichier en utilisant %c*