TP n°10 - Étude expérimentale de complexité Entrées/Sorties.

Tous les codes de ce TP devront être remis sur Moodle - TP10 - avant jeudi 08/12 23h59

Pour ce TP, sur les machines du lycée, il faudra travailler dans un répertoire `TP10` situé en dehors du dossier `Documents`.

Exercice 1 (Question rapidité - OCaml - 10 minutes max).

On réfléchira d'abord en pseudo-code sur papier.

Écrire en OCaml une fonction `merge : 'a list -> 'a list -> 'a list` qui reçoit deux listes **triées** et qui renvoie une liste triée formée des éléments des deux listes.

Par exemple :

- `merge [1;3;5] [2;4;6;8;10]` renvoie `[1;2;3;4;5;6;8;10]`
- `merge ['b';'b';'e';'k';'p'] ['a';'a';'g','z']` renvoie `['a';'a';'b';'b';'e';'g';'k';'p']`

On ne demande pas dans un premier temps l'implémentation d'une récursivité terminale.

Pour les plus rapides :

- tester votre fonction sur des tableaux d'entiers pseudo-aléatoires triés (cf TPs précédents).
- rendre cette fonction récursive terminale. On pourra utiliser la fonction `rev` du module `List` (équivalent de la fonction `miroir` que nous avions codée)

Cette fonction nous resservira lorsque nous verrons l'algorithme de tri fusion au second semestre.

Corrigé de l'exercice 1. *[Retour à l'énoncé]*

Le minimum attendu était :

```
# (* precondition: l1 et l2 sont triées *)
let rec merge l1 l2 =
  match (l1, l2) with
  | ([], []) -> []
  | ([], _) -> l2
  | (_, []) -> l1
  | (t1::q1, t2::q2) when (t1 < t2) -> t1::(merge q1 l2)
  | (t1::q1, t2::q2)                -> t2::(merge l1 q2);;
val merge : 'a list -> 'a list -> 'a list = <fun>
# let l1 = [1;3;5];;
val l1 : int list = [1; 3; 5]
# let l2 = [2;4;6;8;10];;
val l2 : int list = [2; 4; 6; 8; 10]
# merge l1 l2;;
- : int list = [1; 2; 3; 4; 5; 6; 8; 10]
# let l1 = ['b'; 'b'; 'e'; 'k'; 'p'];;
val l1 : char list = ['b'; 'b'; 'e'; 'k'; 'p']
# let l2 = ['a'; 'a'; 'g'; 'z'];;
val l2 : char list = ['a'; 'a'; 'g'; 'z']
# merge l1 l2;;
- : char list = ['a'; 'a'; 'b'; 'b'; 'e'; 'g'; 'k'; 'p'; 'z']
```

On note la symétrie du code, en lien avec la symétrie des entrées : les deux entrées doivent pouvoir être échangées en conservant le même résultat, ce qui se voit dans la symétrie des cas implémentés.
Voici les corrigés des améliorations possibles, avec génération de tests aléatoires :

```
# Random.self_init();;
- : unit = ()
# let rec genere_liste_alea a b n =
  match n with
  | n when n < 0 -> failwith "Taille negative"
  | 0            -> []
  | _ when a > b -> (b + Random.int (a - b + 1))::(genere_liste_alea b a (n-1)) (*résilience du code si l'utilisateur donne a > b*)
  | _            -> (a + Random.int (b - a + 1))::(genere_liste_alea a b (n-1));;
val genere_liste_alea : int -> int -> int -> int list = <fun>
# let rec tri_insertion l =
  (* on définit insere comme une fonction auxiliaire
     locale à la fonction principale,
     mais cela n'est pas obligatoire *)
  let rec insere v l_trie =
    match l_trie with
    | [] -> [v]
    | h::t when v < h -> v::l_trie
    | h::t -> h :: (insere v t)
  in
  match l with
  |[] -> []
  |h::t -> insere h (tri_insertion t);;
val tri_insertion : 'a list -> 'a list = <fun>
# let l1 = tri_insertion (genere_liste_alea 0 30 10);;
val l1 : int list = [2; 9; 9; 9; 12; 14; 17; 21; 25; 29]
# let l2 = tri_insertion (genere_liste_alea 0 50 15);;
val l2 : int list =
  [0; 1; 11; 14; 15; 16; 19; 22; 23; 25; 27; 29; 29; 31; 46]
# merge l1 l2;;
- : int list =
[0; 1; 2; 9; 9; 9; 11; 12; 14; 14; 15; 16; 17; 19; 21; 22; 23; 25; 25; 27;
 29; 29; 29; 31; 46]
```

et avec réécriture de la récursivité sous forme terminale pour éviter tout débordement de pile :

```
# let  merge_terminale l1 l2 =
  let rec aux l_acc l1 l2 =
  match (l1, l2) with
  | ([], []) -> List.rev l_acc
  | ([], _) -> (List.rev l_acc)@l2
  | (_, []) -> (List.rev l_acc)@l1
  | (t1::q1, t2::q2) when (t1 < t2) -> aux (t1::l_acc) q1 l2
  | (t1::q1, t2::q2)                -> aux (t2::l_acc) l1 q2
  in
  aux [] l1 l2;;
val merge_terminale : 'a list -> 'a list -> 'a list = <fun>
# let l1 = [1;3;5];;
val l1 : int list = [1; 3; 5]
# let l2 = [2;4;6;8;10];;
val l2 : int list = [2; 4; 6; 8; 10]
# merge_terminale l1 l2;;
- : int list = [1; 2; 3; 4; 5; 6; 8; 10]
# let l1 = ['b'; 'b'; 'e'; 'k'; 'p'];;
val l1 : char list = ['b'; 'b'; 'e'; 'k'; 'p']
# let l2 = ['a'; 'a'; 'g'; 'z'];;
val l2 : char list = ['a'; 'a'; 'g'; 'z']
# merge_terminale l1 l2;;
- : char list = ['a'; 'a'; 'b'; 'b'; 'e'; 'g'; 'k'; 'p'; 'z']
# let l1 = tri_insertion (genere_liste_alea 0 30 10);;
val l1 : int list = [0; 6; 11; 14; 16; 19; 21; 21; 27; 28]
# let l2 = tri_insertion (genere_liste_alea 0 50 15);;
val l2 : int list = [1; 2; 2; 3; 4; 12; 14; 16; 21; 22; 29; 32; 37; 45; 48]
# merge_terminale l1 l2;;
- : int list =
[0; 1; 2; 2; 3; 4; 6; 11; 12; 14; 14; 16; 16; 19; 21; 21; 21; 22; 27; 28; 29;
 32; 37; 45; 48]
```

Sous Linux, dans le terminal, il est possible de mesurer le temps d'exécution d'une commande en la faisant précéder de la commande `time`. Par exemple, la commande

```
time ./mon_prog 8
```

permet d'obtenir le temps d'exécution du programme `mon_prog` qui a été lancé avec 8 comme unique

entrée.

Exercice 2 (Analyse du temps d'exécution du tri par insertion).

1. Copier le code C de l'algorithme de tri par insertion **iteratif** du TP n°9 dans votre dossier de travail. Nous allons faire fonctionner ce code sur de très grands tableaux, je vous invite donc à commenter les affichages de tableaux.
2. Exécuter l'algorithme de tri par insertion en mesurant le temps d'exécution sur des tableaux d'entiers aléatoires de tailles n, avec les valeurs de n (imposées) suivantes : 1000, 10000, 25000, 50000, 75000, 100000, 125000, 150000. Dans un fichier texte `mesures.txt`, sauvegarder les informations relevées pour chaque exécution : la taille de tableau donnée en entrée et le temps d'exécution **réel**.
3. Écrire un script Python `trace_complexite.py` qui trace la courbe reliant le temps de calcul obtenu en fonction de la taille n du tableau à trier.
4. Observer la courbe obtenue. Quelle conjecture pouvez-vous faire sur la complexité temporelle de l'algorithme ?
5. La méthodologie utilisée vous parait-elle satisfaisante ? Quels aspects de cette petite étude peuvent être améliorés ?

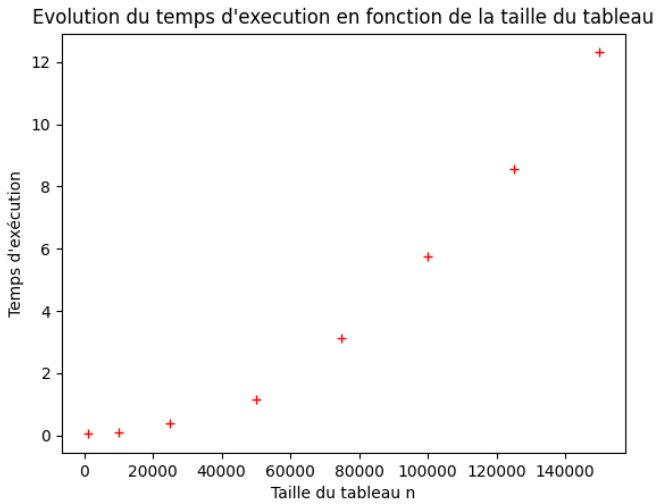
Un prochain travail permettra de définir une méthodologie plus rigoureuse et automatisée pour cette étude de complexité temporelle expérimentale.

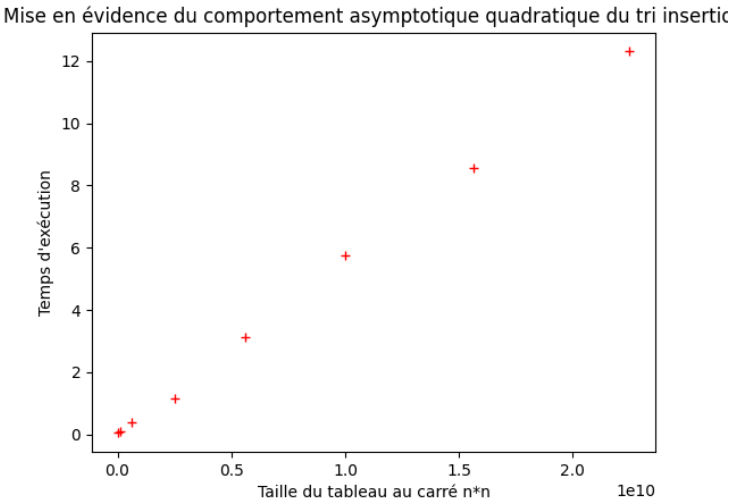
Corrigé de l'exercice 2. *[Retour à l'énoncé]*

Voici le script Python contenant un exemple de mesures de temps d'exécution et la mise en évidence du comportement asymptotique quadratique du temps d'exécution :

```python
import matplotlib.pyplot as plt
import numpy as np

# taille du tableau généré pour chaque exécution
n = [1000, 10000, 25000, 50000, 75000, 100000, 125000, 150000]
# temps de calcul mesuré expérimentalement pour chaque exécution
t_insertion = [0.043, 0.104, 0.380, 1.146, 3.112, 5.748, 8.553, 12.312]


plt.plot(n, t_insertion, "r+")
plt.xlabel("Taille du tableau n")
plt.ylabel("Temps d'exécution")
plt.title("Evolution du temps d'execution en fonction de la taille du tableau")
plt.show()

ncarre = [k*k for k in n]
plt.plot(ncarre, t_insertion, "r+")
plt.xlabel("Taille du tableau au carré n*n")
plt.ylabel("Temps d'exécution")
plt.title("Mise en évidence du comportement asymptotique quadratique du tri insertion")
plt.show()
```

Voici les courbes obtenues :





La méthodologie utilisée est rudimentaire :

- l'intégralité du temps d'exécution est mesuré (temps de chargement, d'allocation du tableau, de génération des valeurs aléatoires...) alors que l'on ne s'intéresse qu'au temps du tri par insertion
- on ne fait qu'une mesure pour chaque taille de tableau ce qui peut amener beaucoup d'irrégularité selon la génération aléatoire
- et bien sûr, le temps d'exécution dépend de la machine, de l'état d'occupation du processeur au moment de chaque test...

Exercice 3 (Analyse d'un fichier en langage C).
Les fonctions suivantes seront codées en C dans un fichier `analyse_fichier_texte.c`
On fait l'hypothèse que tous les fichiers textes manipulés sont au format ASCII.

1. Écrire une fonction `compte_lignes` qui compte le nombre de lignes d'un fichier. Le chemin du fichier à analyser (chemin absolu ou relatif) sera donné sur la ligne de commande par l'utilisateur *Indication : on atteint la fin d'une ligne quand on lit un caractère de retour à la ligne.*
2. Tester votre code sur différents fichiers. Vous pouvez créer vous-mêmes quelques fichiers de tests assez courts sous `emacs` pour commencer. Une base de fichiers de tests est à disposition sur Moodle.
3. Réfléchir sur papier à une fonction `compte_mots` qui prend en entrée le nom d'un fichier texte, et renvoie en sortie le nombre de mots du fichier. *Indication : On considère que les mots sont séparés par un* ***ou plusieurs*** *espaces et/ou retours à la ligne. Les caractères de ponctuation collés aux mots sont comptés comme faisant partie d'un mot.*
4. Coder cette fonction dans votre fichier `analyse_fichier_texte.c`
5. Tester comme pour la première fonction, avec de petits fichiers que vous vous créerez puis de plus gros fichiers, par exemple ceux fournis sur Moodle. Vous pouvez comparer vos résultats avec ceux de la commande Linux `wc` (*word count*) dont vous pourrez aller lire le manuel [a].

a. Pour obtenir le manuel : `man wc`

Corrigé de l'exercice 3. *[Retour à l'énoncé]*

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h>

int compte_lignes(char *chemin_fichier)
{
  FILE *f = fopen(chemin_fichier, "r"); // on ouvre le fichier et on récupère
  //le descripteur de fichier permettant, clé d'accès au canal.

  assert(f != NULL); // prog defensive: on verifie que l'ouverture du canal
                     //vers le fichier en lecture a été effectuée avec succès

  int nb_lignes = 0; // compteur de lignes initialisé à 0
  char caractere_lu; // variable qui contiendra les caractères lus successivemement

  while( feof(f) == 0 ) // tant que le caractère EOF de fin de fichier n'a pas été lu
  {
    fscanf(f, "%c", &caractere_lu); // lire un caractère et stocker sa valeur
                                    //dans la variable caractere_lu

    if (feof(f) == 1) // si on vient de lire le caractère de fin de fichier.
      break; // on casse la boucle

    if (caractere_lu == '\n') // si on a lu un retour à la ligne
      nb_lignes = nb_lignes + 1; // on compte une ligne de plus
  }

  fclose(f); // on n'oublie pas de refermer le canal de lecture depuis le fichier

  return (nb_lignes); // on retourne le nombre de lignes comptées
}
```

```c
int compte_mots(char *chemin_fichier)
{
  FILE *f = fopen(chemin_fichier, "r");

  assert(f != NULL);

  int nb_mots = 0;
  char caractere_lu;
  bool deb = true;


  while (feof(f) == 0)
  {
    fscanf(f, "%c", &caractere_lu);

    // on avance jusqu'à trouver un caractère non blanc ou jusqu'à la fin du fichier
    while (feof(f) == 0 && (caractere_lu == ' ' || caractere_lu == '\n'))
       fscanf(f, "%c", &caractere_lu);

    if (feof(f)) // si on a atteint la fin du fichier, on casse la boucle et on sort
      break;


    // sinon, si c'est la premiere fois qu'on retombe sur un caractere non blanc,
    //c'est le debut d'un mot
    if (deb == true)
    {
      // on incrémente le compteur de mots
      nb_mots = nb_mots + 1;
      deb = false;
    }

    // on lit la suite du mot et on arrête au prochain caractere blanc
    while (feof(f) == 0 && (caractere_lu != ' ' && caractere_lu != '\n'))
       fscanf(f, "%c", &caractere_lu);

    deb = true; // on se prépare pour la lecture du début d'un nouveau mot

  }

  // on n'oublie pas de fermer le canal de communication avec le fichier
  fclose(f);

  return nb_mots;
}
```

```c
int main(int argc, char **argv)
{

  assert(argc == 2); // prog defensive: on vérifie que l'utilisateur a bien
  // donné un nom de fichier sur la ligne de commande

  char *chemin_fichier = argv[1];

  int nl = compte_lignes(chemin_fichier);
  int nm = compte_mots(chemin_fichier);

  printf("Le fichier %s possède:\n %d lignes\n %d mots\n", chemin_fichier, nl, nm);

  return 0;

}
```

```
golivier@ordiprof:~/doc/MPII/TP10$ wc samples/sample1.txt
  4  88 608 samples/sample1.txt
golivier@ordiprof:~/doc/MPII/TP10$ ./analyse_fichier_texte samples/sample1.txt
Le fichier samples/sample1.txt possède:
 4 lignes
 88 mots
golivier@ordiprof:~/doc/MPII/TP10$ wc samples/sample2.txt
  12  423 2859 samples/sample2.txt
golivier@ordiprof:~/doc/MPII/TP10$ ./analyse_fichier_texte samples/sample2.txt
Le fichier samples/sample2.txt possède:
 12 lignes
 423 mots
golivier@ordiprof:~/doc/MPII/TP10$ wc samples/sample3.txt
  20  546 3541 samples/sample3.txt
golivier@ordiprof:~/doc/MPII/TP10$ ./analyse_fichier_texte samples/sample3.txt
Le fichier samples/sample3.txt possède:
 20 lignes
 546 mots
golivier@ordiprof:~/doc/MPII/TP10$ wc samples/sample4.txt
   5696  322392 2167737 samples/sample4.txt
golivier@ordiprof:~/doc/MPII/TP10$ ./analyse_fichier_texte samples/sample4.txt
Le fichier samples/sample4.txt possède:
 5696 lignes
 322392 mots
golivier@ordiprof:~/doc/MPII/TP10$ wc samples/exemple.txt
  1  66 379 samples/exemple.txt
golivier@ordiprof:~/doc/MPII/TP10$ ./analyse_fichier_texte samples/exemple.txt
Le fichier samples/exemple.txt possède:
 1 lignes
 66 mots
```