université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

semaine du 10 décembre 2021

Introduction à la programmation fonctionnelle
L2 Informatique

TP n° 6

Consignes les exercices ou questions marqués d'un ⋆ devront être rédigés sur papier (afin de se préparer aux épreuves écrites de l'examen). En particulier, il est recommandé d'être dans les même conditions qu'en examen : pas de document ni de calculatrice. Tous les TPs se font sous Linux.

1 Tris en folie

Le but de l'exercice est d'implémenter deux fonctions de tri sur les listes et d'en évaluer les performances. On évaluera aussi les performances de la fonction prédéfinie `List.sort` vue en cours.

1. Écrire une fonction `gen_list : (int -> 'a) -> int -> 'a list` telle que `gen_list f n` construise la liste

   ```
   [ (f 0); (f 1); ... ; (f (n-1))]
   ```

2. Utiliser la fonction `gen_list` pour créer les fonctions `int_list : int -> int list` qui construit la liste des entiers de 0 à $n-1$ et `rand_list : int -> int list` qui construit une liste d'entiers aléatoires compris entre 0 et `n-1`. On pourra utiliser la fonction `Random.int n` qui renvoie un entier dans cet intervalle.

3. Écrire une fonction `time : ('a -> 'b) -> 'a -> 'b * float`.L'appel à `time f x` calcule le temps mis pour exécuter `f x` et renvoie un couple formé du résultat et du temps d'exécution en millisecondes. On rappelle que la fonction `Unix.gettimeofday()` renvoie un flottant représentant l'heure qu'il est, exprimée en secondes écoulée depuis le 1er janvier 1970, 00 :00 :00 UTC (cette date est appelée *Epoch*). De plus, on effectuera un appel à `Gc.compact()` en début de fonction avant de mesurer le temps d'exécution. En effet, en OCaml la gestion de la mémoire est automatique. Un programme peut donc être interrompu aléatoirement lorsque le système décide de désallouer de la mémoire automatiquement. L'appel à `Gc.compact` permet de forcer ce « nettoyage » de la mémoire et éviter qu'il ne se produise à un moment inopportun, par exemple pendant l'exécution d'une fonction dont on cherche à mesurer le temps d'exécution.

   Attention : à partir de cette question, il faut utiliser la ligne de commande suivante pour compiler votre programme :

   ```
   ocamlopt -o test.exe unix.cmxa test.ml
   ```

   où `test.ml` est le nom de votre programme OCaml.

4. Écrire une fonction `insert_sort : ('a -> 'a -> int) -> 'a list -> 'a list` qui prend en argument une fonction de comparaison (cf. `List.sort`) et tri la liste passée en argument en utilisant l'algorithme du tri par insertion.
   - définir dans un premier temps une fonction imbriquée récursive `ins : 'a list -> 'a -> 'a list` qui prend en argument une liste et un élément et va insérer l'élément à la bonne position dans la list (en utilisant la fonction de comparaison).
   - une fois la fonction `ins` définie, recréer une liste triée en insérant à la bonne position tous les éléments de la liste initiale (on fera une utilisation judicieuse de `List.fold_lfet`).

5. Écrire une fonction `test_sort : (int list -> int list) -> string -> int list -> unit`. `test_sort :`
   - attend en argument une fonction `f` qui trie une liste d'entiers par ordre croissant
   - attend en argument une chaîne de caractères `n`
   - attend en argument une liste d'entiers `sizes`
   - Commence par appeler `Random.init 42`

— Affiche sur une ligne la chaîne `n`
— Et pour chaque entier s de la liste `sizes`, génère une liste des s premiers entiers de cette taille (avec `int_list`) et une liste aléatoire de cette taille (avec `rand_list`), calcule le temps mis pour la trier et affiche la taille de la liste et ce temps dans la console.
— Enfin la fonction affiche deux retours à la ligne.

Par exemple, si dans le code OCaml on appelle :

```
1  test_sort (insert_sort compare) "insert_sort" [0;1;10;20;50;100] ;;
```

on obtient l'affichage

```
insert_sort
insert_sort
0 (trié)  0.000000
0 (random)  0.000000
1 (trié)  0.000000
1 (random)  0.000954
10 (trié)  0.000000
10 (random)  0.000000
20 (trié)  0.003099
20 (random)  0.010014
50 (trié)  0.034809
50 (random)  0.009060
100 (trié)  0.066996
100 (random)  0.056028
```

6. Tester votre tri par insertion et déterminer à partir de quelle taille de liste on dépasse 16ms [1].
7. Écrire maintenant la fonction `quick_sort` vue en cours. La tester et déterminer de la même façon la taille des listes que l'on peut trier en moins de 16ms. Quel type de liste pose problème ? On solutionnera ce problème dans la partie 2 ci-dessous.
8. Tester la fonction `List.sort` et la comparer à `quick_sort` et `insert_sort`.

2 Un meilleur *QuickSort* (bonus)

Bien que très rapide « en pratique », l'algorithme QuickSort possède le défaut d'être quadratique (temps de calcul proportionnel à n^2 où n est la longueur de la liste) si les éléments sont déjà triés. En réalité, on peut même montrer que quelle que pour une stratégie de choix de pivot donnée, il existe une disposition de liste qui donne à la fonction ce mauvais comportement.

Dans la première partie, le pivot était simplement choisi comme étant le premier élément de la liste. Pour ce choix, plus la liste est triée, et pire sera le temps de calcule (le pire cas étant la liste complètement triée).

Un autre souci est le cas des listes contenant des éléments tous égaux où là aussi, quel que soit le choix du pivot, on obtient un comportement quadratique.

On va résoudre ces deux problèmes.

1. Recopier la fonction `quicksort` en une fonction `quicksort_opt`.
2. Modifier la sous-fonction `split` pour qu'elle ne renvoie pas deux listes (plus petit et plus grand que le pivot) mais trois listes (plus petit, égal et plus grand que le pivot). Modifier ensuite la fusion des listes pour tenir compte de cette modification.
3. Modifier en suite la fonction pour appeler une fonction `pivot_random : 'a list -> 'a * 'a list`. Cette fonction choisit aléatoirement un pivot. De la façon suivante
   — calculer la longueur l de la liste
   — tirer un entier aléatoire i entre 0 et $l-1$ inclus
   — effectuer un parcours récursif pour renvoyer le $i^{\text{ème}}$ élément de la liste ainsi que la liste privée de ce $i^{\text{ème}}$ élément
4. Utiliser cette fonction `pivot` dans `quick_sort_opt` et constater que le comportement quadratique sur les listes triées a disparu.

1. Pour donner une idée, c'est le temps de calcul auquel on a droit entre deux *frames* d'un jeu vidéo fluide affichant 60 images par secondes : $\frac{1000}{60} = 16.666$. Si le calcul (collisions, logique du jeu, points de vie, ...) prend plus de temps, le jeu laggue.

Remarque Comme on choisit aléatoirement le pivot, la probabilité de tomber sur la pire permutation possible est de $\frac{1}{n!}$ (il y a $n!$ permutations possibles d'une liste). Pour une liste de taille 10, vos chance sont plus élevées que votre ordinateur soit frappé par la foudre que de tomber sur cette mauvaise permutation.