Tri_a_bulle_et_tri_cocktail

May 21, 2019

1 Table of Contents

2 Exercice d'algorithmique - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2016-17

- *Date* : 29 août 2017
- *Auteur* : Lilian Besson

2.2 À propos de ce document

- Ceci est une *proposition* de correction, pour un exercice (simple) d'algorithmique, pour la préparation à l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [1]: Sys.command "ocaml -version";;
```

```
The OCaml toplevel, version 4.04.2
```

```
Out[1]: - : int = 0
```

2.3 Le problème : tri à bulles et tri cocktail

On s'intéresse à deux algorithmes de tris. Pour plus de détails, voir les pages Wikipédia (ou le [Cormen] par exemple) :

- Tri à bulle,
- Tri cocktail.

On veut implémenter les deux et les comparer, sur plein d'entrées.

2.4 Tri à bulles

Commençons par le plus classique.

```
In [2]: (** Un échange T[i] <-> T[j]. Utilise une valeur temporaire *)
        let swap tab i j =
            let tmp = tab.(i) in
            tab.(i) <- tab.(j);
            tab.(j) <- tmp
        ;;
```

```
Out[2]: val swap : 'a array -> int -> int -> unit = <fun>
```

Le tri à bulle a une complexité en $\mathcal{O}(n^2)$ dans le pire des cas et en moyenne. Il a l'avantage d'être en place.

```
In [3]: let tri_bulle cmp tab =
            let n = Array.length tab in
            for i = n - 1 downto 1 do
                for j = 0 to i - 1 do
                    if cmp tab.(j + 1) tab.(j) < 0 then
                        swap tab (j + 1) j;
                done
            done
        ;;
```

```
Out[3]: val tri_bulle : ('a -> 'a -> int) -> 'a array -> unit = <fun>
```

On utilise une fonction de comparaison générique. La fonction `compare` (`Pervasives.compare`) fonctionne pour tous les types par défaut.

Une version plus efficace existe aussi (voir ces explications).

```
In [4]: let tri_bulle_opt cmp tab =
            let n = Array.length tab in
            let i = ref 0 in
            let last_swap = ref (-1) in
            while !i < n - 1 do
                (* séquence d'échanges sur T[i..n]: *)
                last_swap := n - 1;
                for j = n-1 downto !i + 1 do
                    if cmp tab.(j) tab.(j-1) < 0 then begin
                        swap tab j (j-1);
                        last_swap := j - 1
                    end;
                    (* i avance "plus vite" :*)
                    i := !last_swap + 1;
                done
            done
        ;;

Out[4]: val tri_bulle_opt : ('a -> 'a -> int) -> 'a array -> unit = <fun>
```

2.5 Tests

On fait quelques tests.

2.5.1 Utilitaire pour des tests

```
In [5]: (** Taille max des éléments dans les tableaux aléatoires *)
        let maxint = int_of_float(1e3);;

Out[5]: val maxint : int = 1000


In [9]: (** Créer un tableau aléatoire. En O(n). *)
        let rand_array length =
            Array.init length (fun _ -> Random.int maxint)
        ;;

Out[9]: val rand_array : int -> int array = <fun>


In [10]: (** Vérifie que chaque élément du tableau n'y est qu'une seule fois (test l'égalité
             Mal codé, en O(n^2). *)
         let isInjArray tab =
             try begin
                 Array.iteri (fun i x ->
                     (Array.iteri (fun j y ->
```

```
                    assert( (x<>y) || (i=j) )
                ) tab)
            ) tab;
            true
            end
        with _ -> false
    ;;
```

```
Out[10]: val isInjArray : 'a array -> bool = <fun>
```

```
In [11]: (** En moyenne, est en O(n^2) si [maxint] est bien plus grand que [n]. *)
    let rand_array_inj = function
        | 0 -> [||]
        | length ->
            let tab = ref (rand_array length) in
            while not(isInjArray (!tab)) do
                tab := rand_array length;
            done;
            !tab
    ;;
```

```
Out[11]: val rand_array_inj : int -> int array = <fun>
```

Cette fonction prend un seul argument, la taille du tableau :

```
In [12]: rand_array_inj 12;;
```

```
Out[12]: - : int array = [|344; 685; 182; 641; 439; 500; 104; 20; 921; 370; 217; 885|]
```

```
In [13]: rand_array_inj 12;;
```

```
Out[13]: - : int array = [|949; 678; 615; 412; 401; 606; 428; 869; 289; 393; 14; 423|]
```

```
In [14]: rand_array_inj 12;;
```

```
Out[14]: - : int array = [|723; 544; 96; 803; 500; 570; 649; 212; 755; 211; 357; 197|]
```

Fonction de test :

```
In [22]: let print = Printf.printf;;

    let testtri mysort cmp length nbrun () =
        print "Test d'un tri, %i simulations avec des tableaux de longueur %i.\n " nbrun l
        for i = 0 to nbrun - 1 do
            let t1 = rand_array length in
```

```
                let t2 = Array.copy t1 in
                mysort cmp t1;
                Array.fast_sort cmp t2;
                try assert( t1 = t2 ) with _ -> begin
                    print "Avec des tableaux de taille %i, le test numéro %i a échoué." lengt
                    Format.printf "@[t1=[|"; Array.iter (fun i -> Format.printf " %i;" i) t1;
                    Format.printf "@[t2=[|"; Array.iter (fun i -> Format.printf " %i;" i) t2;
                end;
            done;
            flush_all ();
        ;;
```

```
Out[22]: val print : ('a, out_channel, unit) format -> 'a = <fun>
```

```
Out[22]: val testtri :
           ((int -> int -> int) -> int array -> 'a) ->
           (int -> int -> int) -> int -> int -> unit -> unit = <fun>
```

2.5.2 Tests du tri à bulle

La fonction de tri par défaut marche bien, évidemment.

```
In [23]: testtri Array.sort compare 10 10 ();;
```

```
Out[23]: - : unit = ()
```

```
 Test d'un tri, 10 simulations avec des tableaux de longueur 10.
```

```
In [24]: testtri tri_bulle compare 10 10 ();;
```

```
Out[24]: - : unit = ()
```

```
 Test d'un tri, 10 simulations avec des tableaux de longueur 10.
```

```
In [25]: testtri tri_bulle_opt compare 10 10 ();;
```

```
 Test d'un tri, 10 simulations avec des tableaux de longueur 10.
```

```
Out[25]: - : unit = ()
```

Ça marche !

```
In [26]: testtri tri_bulle compare 100 1000 ();;

 Test d'un tri, 1000 simulations avec des tableaux de longueur 100.


Out[26]: - : unit = ()
```

2.6 Tri cocktail

Il est très semblable au tri à bulle, sauf que le tableau sera parcouru alternativement dans les eux sens.

Le tri cocktail a une complexité en $\mathcal{O}(n^2)$ dans le pire des cas et en moyenne. Il a l'avantage d'être en place.

```
In [27]: let tri_cocktail cmp tab =
             let n = Array.length tab in
             let echange = ref true in
             while !echange do
                 echange := false;
                 (* Parcours croissant *)
                 for j = 0 to n - 2 do
                     if cmp tab.(j + 1) tab.(j) < 0 then begin
                         swap tab (j + 1) j;
                         echange := true
                     end;
                 done;
                 (* Parcours decroissant *)
                 for j = n - 2 downto 0 do
                     if cmp tab.(j + 1) tab.(j) < 0 then begin
                         swap tab (j + 1) j;
                         echange := true
                     end;
                 done;
             done;
         ;;

Out[27]: val tri_cocktail : ('a -> 'a -> int) -> 'a array -> unit = <fun>
```

Il existe aussi une version plus efficace, qui diminue la taille des parcours au fur et à mesure.

```
In [28]: let tri_cocktail_opt cmp tab =
             let n = Array.length tab in
             let echange = ref true in
             let debut = ref 0 and fin = ref (n - 2) in
             while !echange do
```

```
            echange := false;
            (* Parcours croissant *)
            for j = !debut to !fin do
                if cmp tab.(j + 1) tab.(j) < 0 then begin
                    swap tab (j + 1) j;
                    echange := true
                end;
            done;
            fin := !fin - 1;
            (* Parcours decroissant *)
            for j = !fin downto !debut do
                if cmp tab.(j + 1) tab.(j) < 0 then begin
                    swap tab (j + 1) j;
                    echange := true
                end;
            done;
            debut := !debut + 1;
        done;
    ;;
```

```
Out[28]: val tri_cocktail_opt : ('a -> 'a -> int) -> 'a array -> unit = <fun>
```

2.7 Tests

Et d'autres tests.

```
In [52]: testtri tri_cocktail compare 10 10 ();;
```

```
 Test d'un tri, 10 simulations avec des tableaux de longueur 10.
```

```
Out[52]: - : unit = ()
```

```
In [53]: testtri tri_cocktail_opt compare 10 10 ();;
```

```
 Test d'un tri, 10 simulations avec des tableaux de longueur 10.
```

```
Out[53]: - : unit = ()
```

Ça marche !

```
In [54]: testtri tri_cocktail compare 100 1000 ();;
```

```
 Test d'un tri, 1000 simulations avec des tableaux de longueur 100.
```

```
Out[54]: - : unit = ()
```

2.8 Comparaison

Avec ce morceau de code on peut facilement mesurer le temps d'exécution :

```
In [33]: let time f =
             let t = Sys.time() in
             let res = f () in
             Printf.printf "    Temps en secondes: %fs\n" (Sys.time() -. t);
             flush_all ();
             res
         ;;

Out[33]: val time : (unit -> 'a) -> 'a = <fun>
```

2.8.1 Première comparaison : $n = 100$

```
In [75]: time (testtri Array.sort compare 100 10000);;

Test d'un tri, 10000 simulations avec des tableaux de longueur 100.
     Temps en secondes: 2.612000s


Out[75]: - : unit = ()


In [76]: time (testtri tri_bulle compare 100 10000);;

Test d'un tri, 10000 simulations avec des tableaux de longueur 100.
     Temps en secondes: 2.976000s


Out[76]: - : unit = ()


In [79]: time (testtri tri_bulle_opt compare 100 10000);;

Test d'un tri, 10000 simulations avec des tableaux de longueur 100.
     Temps en secondes: 3.144000s


Out[79]: - : unit = ()


In [80]: time (testtri tri_cocktail compare 100 10000);;
```

```
Test d'un tri, 10000 simulations avec des tableaux de longueur 100.
     Temps en secondes: 3.380000s


Out[80]: - : unit = ()


In [41]: time (testtri tri_cocktail_opt compare 100 10000);;

Test d'un tri, 10000 simulations avec des tableaux de longueur 100.
     Temps en secondes: 2.724000s


Out[41]: - : unit = ()
```

Sur de petits tableaux, les versions "optimisées" ne sont pas plus forcément plus rapides (comme toujours).

On vérifie quand même que sur des tableaux aléatoires, le tri cocktail optimisé semble le plus rapide des quatre implémentations.

2.8.2 Autre comparaison : $n = 1000$

Ca suffit à voir que les quatre algorithmes implémentés ici ne sont pas en temps sous quadratique.

```
In [81]: time (testtri Array.sort compare 1000 1000);;

Test d'un tri, 1000 simulations avec des tableaux de longueur 1000.
     Temps en secondes: 3.072000s


Out[81]: - : unit = ()


In [82]: time (testtri tri_bulle compare 1000 1000);;

Test d'un tri, 1000 simulations avec des tableaux de longueur 1000.
     Temps en secondes: 26.116000s


Out[82]: - : unit = ()


In [83]: time (testtri tri_bulle_opt compare 1000 1000);;

Test d'un tri, 1000 simulations avec des tableaux de longueur 1000.
     Temps en secondes: 29.944000s


Out[83]: - : unit = ()
```

```
In [84]: time (testtri tri_cocktail compare 1000 1000);;

Test d'un tri, 1000 simulations avec des tableaux de longueur 1000.
     Temps en secondes: 27.296000s


Out[84]: - : unit = ()


In [85]: time (testtri tri_cocktail_opt compare 1000 1000);;

Test d'un tri, 1000 simulations avec des tableaux de longueur 1000.
     Temps en secondes: 22.584000s


Out[85]: - : unit = ()
```

2.9 Conclusion

Les deux algorithmes ne sont pas trop difficiles à implémenter, et fonctionnent de façon très similaire.

2.9.1 Question / exercice

- Prouver la correction de chaque algorithme, à l'aide d'invariants bien choisis.
- Prouver la complexité en temps.
- Pourquoi ces noms, à bulle et cocktail ?

2.9.2 Liens

Allez voir d'autres notebooks !