Programmation impérative et fonctionnelle avec OCaml

TP 1

Objectifs :

- Prise en main de l'environnement de développement :
  - terminal avec `bash` ;
  - éditeur `emacs` ;
  - interpréteur `ocaml`, compilateurs `ocamlc` et `ocamlopt`, documentation.
- Programmation impérative :
  - références ;
  - tableaux ;
  - boucles `while` ;
  - itérateurs `iter`, `iteri` et `map` du module `Array`.
- Évaluation de temps de calcul et comparaison d'algorithmes.

Recherche d'un élément dans un tableau trié

On souhaite retrouver l'index[1] d'un élément dans un tableau trié le plus efficacement possible.

Recherche dichotomique

1. Écrire une fonction `rand_array: int -> int array` qui prend la taille n du tableau en paramètre et renvoie un tableau trié (par ordre croissant) de taille n. Les nombres seront générés aléatoirement entre 0 et $10n - 1$ grâce à la fonction `Random.int`. Le tableau sera trié (en place) avec la fonction `Array.sort`.

   ```
   # let t = rand_array 10;;
   val t : int array = [|0; 4; 20; 21; 39; 41; 44; 70; 82; 85|]
   ```

2. Écrire une fonction `dicho: int array -> int -> int` qui prend un tableau d'entiers t et un entier x en paramètre et qui renvoie l'index k de x dans t tel que `t.(k) = x` en utilisant une recherche dichotomique. La fonction `dicho` doit être écrite en style **impératif** à l'aide de (trois) références et d'une boucle `while`, sans allouer de tableau auxiliaire. Dans le cas où l'élément n'est pas présent dans le tableau, on renverra l'entier[2] -1.

   ```
   # dicho t 82;;
   - : int = 8
   # dicho t 40;;
   - : int = -1
   ```

1. Plus généralement, ce type de recherche permet de retrouver la donnée associée à une clé quand on manipule une table d'association.

2. C'est une mauvaise pratique en général. On verra ultérieurement qu'il vaut mieux utiliser une exception ou un « type somme » pour rendre compte d'un cas particulier, plutôt qu'une valeur particulière du type renvoyé dans le cas général.

Recherche par interpolation (linéaire)

Quand on cherche un nom commençant par `'V'` dans un annuaire, on n'utilise pas une recherche dichotomique en commençant par le milieu de l'annuaire (vers la lettre `'M'`), mais plutôt directement vers la fin grâce à une estimation plus précise de la position du nom dans l'annuaire, en supposant que l'ensemble des noms suit une distribution uniforme. Ainsi, pour un tableau `t` de taille 100, si `t.(0)=0` et `t.(99)=1000`, on peut estimer que le nombre 250 devrait se situer à proximité de la case d'index 24.

1. En recopiant (c'est mal !) la fonction `dicho`, écrire une fonction `interpol` de même type qui utilise l'interpolation linéaire plutôt que le milieu du tableau pour renvoyer l'index de l'élément recherché. Si le nombre recherché ne se trouve pas dans le tableau, cette technique nécessite des tests supplémentaires dans la condition de la boucle `while` pour que l'algorithme termine correctement ; afficher (à l'aide de la fonction `Printf.printf`) dans le corps de la boucle la valeur des références utilisées pour la contrôler et l'estimation de la position de l'élément pour comprendre comment corriger la fonction.

```
# interpol t 82;;
- : int = 8
# interpol t 40;;
- : int = -1
```

Complexité

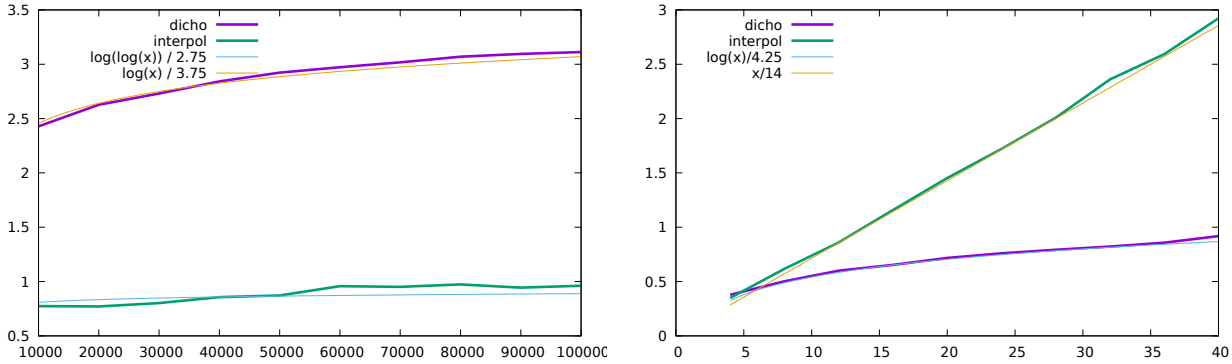
La complexité temporelle moyenne de la recherche par interpolation, lorsque les données sont uniformément distribuées, est en $O(\log \log n)$. En revanche, si `t.(i)` est, par exemple, exponentiel en fonction de `i`, l'estimation linéaire est complètement fausse (sous-estimée) et la complexité en pire cas est en $O(n)$. On souhaite comparer les temps de calcul des deux types de recherche pour vérifier en pratique leur complexité théorique à l'aide de graphiques similaires à ceux de la figure 1.

1. Pour comparer les deux algorithmes équitablement, on se limitera à ne chercher que des éléments présents dans le tableau (sinon les tests supplémentaires de la recherche par interpolation sont susceptibles d'accélérer « artificiellement » l'algorithme). Écrire une fonction `rand_elt: int array -> int` qui prend un tableau en paramètre et renvoie un élément de ce tableau choisi aléatoirement.

```
# rand_elt t;;
- : int = 21
```

2. Pour comparer les complexités des deux algorithmes, on va tracer des courbes en utilisant k points correspondant à k tableaux de taille croissante $(i+1)\times\frac{n_{\max}}{k}, \forall i \in [0, k-1]$ avec $n_{\max}$ la taille maximale des tableaux. Pour avoir des temps de calculs significatifs (assez longs), m recherches doivent être exécutées avec divers éléments pour chaque taille. Écrire une fonction équivalente au `main` du C (i.e. `let () = ...`) pour que votre programme puisse prendre ces différents paramètres en ligne de commande à l'aide du tableau `Sys.argv` et de la fonction de conversion `int_of_string: string -> int` :

```
./tp1.opt nmax m k
```



(a) Distribution uniforme

(b) Distribution exponentielle

FIGURE 1 – Temps de calcul (en secondes) des recherches dichotomique et par interpolation en fonction de la taille du tableau, et ajustage (à la main...) des complexités théoriques.

3. Pour éviter de prendre en compte le temps de création des tableaux, on construira au préalable la « matrice » `tabs` de k tableaux aléatoires de taille croissante $(i+1) \times \frac{n_{\max}}{k}$ dans le (pseudo) `main`. De plus, pour effectuer systématiquement les mêmes recherches d'élément avec les deux algorithmes, on construira également une matrice d'éléments à rechercher `xs`, dont chaque ligne `xs.(i)` d'index $i \in [0, k-1]$ contient un tableau de m éléments de `tabs.(i)`. Pour construire `xs` à partir de `tabs`, on utilisera l'itérateur `Array.map: ('a -> 'b) -> 'a array -> 'b array` et la fonction `rand_elt`.
4. Écrire une fonction `duration: 'a -> 'b array -> ('a -> 'b -> 'c) -> float` telle que `duration tabsi xsi search` renvoie le temps de calcul pour rechercher chacun des éléments du tableau `xsi` dans le tableau `tabsi` (grâce à l'itérateur `Array.iter: ('a -> unit) -> 'a array -> unit`) à l'aide de la fonction `search`. Pour se débarasser du résultat de la recherche[3], on pourra utiliser la fonction `ignore: 'a -> unit` qui évalue puis « oublie » son argument. On utilisera également la fonction `Unix.gettimeofday: unit -> float` pour obtenir la date courante (à la microseconde près), puis on calculera la différence entre deux dates pour obtenir un temps d'exécution. Pour pouvoir utiliser une fonction du module `Unix`, il faut utiliser l'une des commandes suivantes :

   ```
   ocaml unix.cma
   ocamlc unix.cma -o tp1.out tp1.ml
   ocamlopt unix.cmxa -o tp1.opt tp1.ml
   ```

5. En utilisant l'itérateur `Array.iteri: (int -> 'a -> unit) -> 'a array -> unit`, exécuter les deux algorithmes pour chaque taille de tableau puis afficher sur la sortie standard (avec `Printf.printf`) la taille, le temps de calcul du premier algorithme puis celui du second (avec des paramètres moindres selon la puissance du PC) :

   ```
   ./tp1.opt 100000 10000000 10
   ```

3. Un index entier est renvoyé alors que la fonction exécutée par `Array.iter` pour chaque élément du tableau doit renvoyer `()` de type `unit`.

```
10000 2.37914 0.77385
20000 2.61129 0.781833
30000 2.70024 0.769251
40000 2.78516 0.823497
50000 2.87623 0.802686
60000 2.90304 0.910346
70000 2.95824 0.90994
80000 2.997 0.935073
90000 3.05134 0.895043
100000 3.07376 0.924372
```

On pourra alors rediriger la sortie standard du programme dans un fichier pour pouvoir tracer les courbes des temps de calcul avec `gnuplot` :

```
./tp1.opt 100000 10000000 10 > unifplot.txt
```

Pour obtenir un graphique similaire à celui de la figure 1a, lancer alors `gnuplot` (depuis le même répertoire), puis, à l'invite de commande, exécuter :

```
plot "unifplot.txt" u 1:2 w l lw 3 title "dicho", "unifplot.txt" u 1:3
                                                  w l lw 3 title "interpol"
```

6. Écrire une fonction `exp_array: int -> int array` similaire à `rand_array` mais qui renvoie le tableau contenant e^i à l'index i. On pourra utiliser les fonctions `float: int -> float`, `exp: float -> float` et `truncate: float -> int`.

```
# exp_array 10;;
- : int array = [|1; 2; 7; 20; 54; 148; 403; 1096; 2980; 8103|]
```

7. Remplacer la fonction `rand_array` par `exp_array` et écrire les nouveaux temps de calculs dans un fichier `explot.txt`.

```
./tp1.opt 43 10000000 10
4 0.367463 0.350259
8 0.488662 0.606282
12 0.581151 0.859559
16 0.633143 1.1632
20 0.699223 1.46731
24 0.743741 1.75462
28 0.773504 2.02677
32 0.797566 2.32655
36 0.835934 2.63421
40 0.864929 2.93072
```

Comme la fonction `exp_array` calcule (au maximum) e^{n-1}, la taille maximale du tableau doit être inférieure à :

```
# truncate (log (float max_int)) + 1;;
- : int = 43
```

Comme dans la figure 1b, utiliser de nouveau `gnuplot` avec le fichier `explot.txt` pour comparer les temps de calculs des deux algorithmes sur ces nouvelles données.