

DS 1 d'informatique MP2I corrigé

2 heures

Les calculatrices, ordinateurs et documents de cours sont interdits.

Toutes les complexités seront exprimées avec la notation $O(\dots)$ et **doivent être justifiées/prouvées**.

Vous avez le droit d'admettre une question pour passer à la suivante.

Les exercices sont indépendants et vous pouvez les traiter dans l'ordre que vous préférez.

I Minimum et maximum

Écrire une fonction `minmax` telle que, si `l` est une liste d'entiers, `minmax l` renvoie un couple `(mini, maxi)` où `mini` est le minimum de `l` et `maxi` son maximum.

Bonus : l'écrire en utilisant (à ± 1 près) $\frac{3n}{2}$ comparaisons (utilisations de `<` ou `<=`), où n est la taille de `l`.

Solution : Version simple en $2n$ comparaisons (`min` et `max` effectuent chacun une comparaison) :

```
let rec min_max = function (* renvoie (minimum, maximum) *)
| [] -> max_int, min_int
| e::q -> let mini, maxi = min_max q in
          (min e mini), (max e maxi)
```

Version avec $\frac{3n}{2}$ comparaisons, en regardant les éléments 2 par 2 pour "économiser" une comparaison :

```
let rec min_max = function (* renvoie (minimum, maximum) *)
| [] -> max_int, min_int
| [a] -> a, a
| a::b::q -> let mini, maxi = min_max q in
              if a < b then (min a mini), (max b maxi)
              else (min b mini), (max a maxi);;
```

II Tri rapide

1. Écrire une fonction `concat` : `'a list -> 'a list -> 'a list` telle que `concat l1 l2` renvoie une liste composée des éléments de `l1` suivi des éléments de `l2`, sans utiliser `@`.

Quelle est la complexité de `concat` ?

Solution :

```
let rec concat l1 l2 = match l1 with
| [] -> l2
| e::q -> e::concat q l2
```

`concat l1 l2` fait un appel récursif sur chaque élément de `l1`. Chacun de ces appels récursifs effectue un nombre constant d'opérations.

La complexité de `concat l1 l2` est donc $O(n_1)$, où n_1 est la taille de `l1`.

2. Écrire une fonction `partition` telle que, si `l` est une liste d'entiers et `p` un entier, `partition l p` renvoie un couple `(l1, l2)` où :

- `l1` est une liste contenant les éléments de `l` inférieurs strictement à `p`
- `l2` est une liste contenant les éléments de `l` supérieurs ou égaux à `p`

Quelle est la complexité de `partition` ?

Solution :

```
let rec partition l pivot = match l with
| [] -> [], []
| e::q -> let l1, l2 = partition q pivot in
          if e < pivot then e::l1, l2
          else l1, e::l2;;
```

Soit l une liste de taille n . `partition l` fait autant d'appels récurifs que d'éléments dans l . Pour chacun de ces n appels récurifs, `partition` effectue un nombre constant d'opérations (`e::`, `e < pivot`).

La complexité de `partition l` est donc $O(n)$.

Le tri rapide d'une liste l consiste à :

- Choisir un élément (appelé pivot) de l . Ici on prendra le premier élément p de l comme pivot.
 - Séparer les éléments de l autres que p en deux listes : la liste $l1$ des éléments strictement inférieurs à p et la liste $l2$ des éléments supérieurs à p .
 - Trier récursivement $l1$ et $l2$ pour obtenir des listes triées $l1'$ et $l2'$.
 - Renvoyer la concaténation de $l1'$, p et $l2'$.
3. Écrire une fonction `quicksort` : 'a list -> 'a list triant une liste avec le tri rapide.

Solution : On peut utiliser au choix `@` ou la fonction `concat` précédente :

```
let rec quicksort = function
| [] -> []
| p::q -> let l1, l2 = partition q p in
          (quicksort l1) @ (p::quicksort l2);;
```

4. Quelle est la complexité de `quicksort l` pour une liste l de taille n déjà triée dans l'ordre croissant¹ ?

Solution : Dans tous les appels récurifs, $l1$ est vide et $l2$ contient tous les éléments de l sauf p .

Dans ces appels récurifs, `partition` est appelé avec une liste de taille n , puis $n-1$, ..., jusqu'à 1 (il y a un total de n appels récurifs). Donc le nombre total d'opérations effectuées par `partition` est $O(n) +$

$$O(n-1) + \dots + O(1) = O\left(\frac{n(n+1)}{2}\right) = O(n^2).$$

De plus, la liste à gauche de chaque concaténation étant vide, la complexité de chaque concaténation est $O(1)$. La complexité de toutes les concaténations sont donc $O(n)$.

Donc la complexité totale est $O(n^2) + O(n) = O(n^2)$.

5. Quelle est la complexité de `quicksort` sur une liste de taille n quand la partition est toujours équilibrée dans les appels récurifs (les deux listes $l1$ et $l2$ sont de même taille)² ?

Solution : Soit $C(n)$ cette complexité. Le calcul est très similaire à celui du tri fusion.

À chaque appel récurif, on appelle `partition l` en $O(n)$, on fait une concaténation en $O(n)$ et deux appels récurifs sur des tailles $\frac{n}{2}$. Au total, ces deux instructions effectuent au plus Kn opérations, avec K constante.

$$\begin{aligned} C(n) &= Kn + 2C\left(\frac{n}{2}\right) (*) \\ &= Kn + 2K\frac{n}{2} + 4C\left(\frac{n}{2}\right) = 2Kn + 4C\left(\frac{n}{2}\right) \\ &= \dots \\ &= pKn + 2^p C\left(\frac{n}{2^p}\right)_{p=\log_2(n)} = O(n \log_2(n)) \end{aligned}$$

Où on a appliqué $\log_2(n)$ (± 1) fois la relation (*).

¹On peut montrer qu'il s'agit du pire cas

²On peut montrer qu'il s'agit du meilleur cas

III Recherche par trichotomie

1. On considère deux entiers i et j tels que $0 \leq i \leq j$.

Exprimer, en fonction de i et j , des entiers m_1 et m_2 qui partagent les entiers entre i et j en 3 ensembles de même taille (à ± 1 près). Plus précisément, m_1 et m_2 doivent vérifier :

- $i \leq m_1 \leq m_2 \leq j$
- Les trois ensembles suivants contiennent le même nombre d'entiers (à ± 1 près) :

$$\{i, i+1, \dots, m_1\}, \{m_1+1, m_1+2, \dots, m_2\}, \{m_2+1, m_2+2, \dots, j\}$$

Solution : Le nombre d'éléments de $\{i, \dots, j\}$ est $j - i + 1$. Donc on peut choisir :

$$m_1 = \lfloor i + \frac{j-i+1}{3} \rfloor = \lfloor \frac{2i+j+1}{3} \rfloor$$

$$m_2 = \lfloor i + 2\frac{j-i+1}{3} \rfloor = \lfloor \frac{i+2j+2}{3} \rfloor$$

2. Écrire une fonction `tricho` telle que `tricho t e` détermine si `e` appartient à un tableau trié `t`, en utilisant une méthode similaire à la recherche par dichotomie mais en découpant l'intervalle en 3 plutôt que 2.

Solution :

```
let tricho t e =
  let rec aux i j =
    if i > j then false
    else let m1 = (2*i + j + 1)/3 in
          let m2 = (i + 2*j + 2)/3 in
          if t.(m1) = e || t.(m2) = e then true
          else if e < t.(m1) then aux i (m1 - 1)
          else if e < t.(m2) then aux (m1 + 1) (m2 - 1)
          else aux (m2 + 1) j in
    aux 0 (Array.length t - 1)
```

3. Donner la complexité de `tricho` et comparer avec la recherche par dichotomie.

Solution : À chaque appel récursif, la taille de la zone de recherche est divisé par 3. Donc au bout de p appels récursifs sur un tableau de taille p , il reste $\lfloor \frac{n}{3^p} \rfloor$ éléments.

Après $\lfloor \log_3(n) \rfloor$ appels récursifs, il ne reste donc plus d'élément à regarder ($\lfloor \frac{n}{3^p} \rfloor = 0$).

Donc le nombre d'appels récursifs est au plus $\log_3(n)$. Comme à chaque appel récursif, un nombre constant d'opérations (indépendant de n) est effectué, la complexité finale est $O(\log_3(n))$.

En terme de $O(\dots)$, la complexité est donc la même que la recherche par dichotomie. Il faudrait compter le nombre exact d'opérations pour comparer plus précisément, ce qui n'est pas évident étant donné que $\log_3 < \log_2$ mais que la trichotomie fait plus d'opérations que la dichotomie à chaque appel récursif.

IV Méthode des deux pointeurs

Écrire une fonction `somme2 : int array -> int -> int*int` telle que, si `t` est un tableau trié de taille n , `somme2 t p` renvoie un couple (i, j) tel que $i \neq j$ et $t.(i) + t.(j) = p$. Si un tel couple n'existe pas, on renverra $(-1, -1)$. `somme2 t p` doit être en complexité $O(n)$ et ne pas créer de nouvelle structure de donnée (pas de création de tableau, liste...)³.

Indice : Utiliser deux références `i` et `j` valant initialement 0 et $n-1$. Que peut-on faire si $t.(i) + t.(j) < p$? Et si $t.(i) + t.(j) > p$?

Solution : si $t.(i) + t.(j) < p$, il faut intuitivement augmenter la valeur de $t.(i) + t.(j)$ en augmentant `i`. D'où la solution récursive :

³Autrement dit, la complexité en mémoire doit être $O(1)$

```

let somme2 t p =
  let rec aux i j =
    if i > j then (-1, -1)
    else if t.(i) + t.(j) = p then (i, j)
    else if t.(i) + t.(j) < p then aux (i + 1) j
    else aux i (j - 1) in
  aux 0 (Array.length t - 1)

```

On peut aussi l'écrire en itératif :

```

let somme2 t p =
  let i = ref 0 in
  let j = ref (Array.length t - 1) in
  let res = ref (-1, -1) in
  while !i < !j && !res = (-1, -1) do
    if t.(!i) + t.(!j) = p then res := (!i, !j)
    else if t.(!i) + t.(!j) > p then decr j
    else incr i
  done;
  !res

```

Pour le **while** précédent, on peut montrer l'**invariant de boucle** : "si il existe k et l tels que $t.(k) + t.(l) = p$ alors k et l sont entre $!i$ et $!j$ ".

V Élément majoritaire

Dans cet exercice, on veut trouver un élément strictement majoritaire dans un tableau de n entiers naturels, c'est à dire un élément apparaissant strictement plus de $\frac{n}{2}$ fois.

1. Écrire une fonction `occ : 'a -> 'a array -> int` telle que `occ e t` renvoie le nombre d'apparitions de `e` dans `t`.

Par exemple, `occ 2 [|1; 2; 6; 2; 2; 8|]` doit renvoyer 3.

Solution :

```

let occ e t =
  let res = ref 0 in
  for i=0 to Array.length t - 1 do
    if t.(i) = e then incr res
  done;
  !res

```

2. En déduire une fonction `maj` pour trouver un élément majoritaire dans un tableau. Si `t` n'a pas d'élément majoritaire, `maj t` renverra -1.

Solution : On peut appliquer `occ` en $O(n)$ sur chaque élément de la liste, ce qui est en $O(n^2)$.

```

let maj t =
  let res = ref (-1) in
  let n = Array.length t in
  for i=0 to n - 1 do
    if occ t.(i) > n/2 then res := i
  done;
  !res

```

3. Quelle est la complexité de `maj t` sur un tableau `t` de taille n ?

Solution : `maj t` utilise n fois `occ` dont la complexité est $O(n)$.

Donc la complexité de `maj t` est $n \times O(n) = O(n^2)$.

On considère maintenant la fonction suivante :

```

let vote t =
  let e = ref t.(0) in
  let k = ref 1 in
  for i = 0 to Array.length t - 1 do
    if t.(i) = !e then incr k else decr k;
    if !k = 0 then (e := t.(i); k := 1)
  done;
  !e

```

On rappelle que `incr k`/`decr k` augmente/diminue la valeur de la référence `k` de 1.

4. Supposons que le tableau `t` ait un élément strictement majoritaire `m`. Montrer que `vote t` renvoie `m`.

Indice : considérer $c = k$ si `!e = m`, $c = -k$ sinon (c change donc au cours de l'algorithme).

Solution : À chaque fois que `m` est rencontré, c augmente de 1. À chaque fois qu'un autre élément est rencontré, c diminue de 1 ou augmente de 1.

Comme `m` est majoritaire, à la fin de l'algorithme, c a augmenté plus de fois qu'il n'a diminué donc $c > 0$, ce qui n'est possible que si `!e = m`, par définition.

5. En déduire une fonction `maj2` renvoyant un élément strictement majoritaire d'un tableau de taille n en complexité $O(n)$. On renverra -1 s'il n'y a pas d'élément strictement majoritaire.

Solution :

```

let maj2 t =
  let v = vote t in
  if occ e t > Array.length t / 2 then v
  else -1

```

Si `t` est de taille n , `vote t` et `occ e t` étant en complexité $O(n)$, la complexité de `maj2 t` est bien $O(n) + O(n) = \boxed{O(n)}$.