

1 Somme et partition de multi-ensembles

1.1 Fonctions préliminaires

```
1. bool tous_positifs(multiens S) {  
    for (int i = 0; i < S.taille; i++) {  
        if (S.elements[i] < 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
2. int somme(multiens S) {  
    int s = 0;  
    for (int i = 0; i < S.taille; i++) {  
        s += S.elements[i];  
    }  
    return s;  
}
```

```
3. int occurrences(multiens S, int x) {  
    int c = 0;  
    for (int i = 0; i < S.taille; i++) {  
        if (S.elements[i] == x) { c++; }  
    }  
    return c;  
}
```

```
4. bool est_sous_ensemble(multiens S, multiens S1) {  
    for (int i = 0; i < S1.taille; i++) {  
        int x = S1.elements[i];  
        if (occurrences(S,x) < occurrences(S1,x)) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
5. bool est_partition(multiens S, multiens S1, multiens S2) {  
    if (!est_sous_ensemble(S,S1) || !est_sous_ensemble(S,S2)) {  
        return false;  
    }  
    for (int i = 0; i < S.taille; i++) {  
        int x = S.elements[i];  
        if (occurrences(S,x) != occurrences(S1,x) + occurrences(S2,x)) {  
            return false;  
        }  
    }  
    return true;  
}
```

1.2 Résolution naïve

6. (a) Si $S.taille = n$, il y a 2^n sous-ensembles $S1$ de S dont les éléments apparaissent dans le même ordre que dans S .
En effet, pour chaque élément x de $S.elements$, on a deux choix indépendants : soit on garde x dans $S1.elements$, soit on ne le garde pas.
- (b) Puisqu'il y aurait un nombre exponentiel de sous-ensembles $S1$ à tester, et que pour chacun d'entre eux, le calcul de $somme(S1)$ se fera en temps linéaire, alors dans le pire des cas la réponse est **false** et il faudra tous les tester : on aura donc une complexité temporelle exponentielle.
7. Tout d'abord, on remarque que si $0 \leq n_max < S.taille$, l'accès à $S.elements[n_max]$ (ligne 8) ne produira pas d'erreur.

Ensuite, si on passe par le **return** de la ligne 8, on effectue au plus deux appels récursifs, avec $n_max' = n_max - 1$. Donc la valeur de n_max décroît strictement à chaque appel récursif. De plus, cette valeur est entière, donc elle va atteindre une valeur strictement négative en un nombre fini d'étapes. Or la fonction `sous_somme` termine immédiatement si $n_max < 0$ (lignes 5 et 6).

Ainsi, l'appel à `sous_somme(S, t, n_max)` termine pour tout multi-ensemble S d'entiers positifs et tous entiers $n_max < S.taille$ et $t \in \mathbb{N}$.

8. On montre cette propriété par récurrence sur n_max :
- cas $n_max < 0$: dans ce cas, $S.elements[0:n_max+1]$ est vide, donc un tel sous-ensemble n'existe pas et il est correct de renvoyer **false** (lignes 5 et 6) ;
 - cas $n_max \geq 0$:
 - si $t < 0$, comme S est supposé ne contenir que des entiers positifs, un tel sous-ensemble n'existe pas, et il est correct de renvoyer **false** (lignes 5 et 6) ;
 - si $t = 0$, le sous-ensemble vide convient, et il est correct de renvoyer **true** (lignes 2 et 3) ;
 - sinon :
 - Soit un tel sous-ensemble $S1$ existe et contient la valeur $S.elements[n_max]$: dans ce cas, en enlevant cette valeur à $S1$, on obtient un sous-ensemble de $S.elements[0:n_max]$ dont la somme des éléments vaut $t - S.elements[n_max]$. Ainsi, par hypothèse de récurrence, le premier appel récursif de la ligne 8 renvoie **true**, donc `sous_somme(S, t, n_max)` renvoie **true**, ce qui est correct.
 - Soit un tel sous-ensemble $S1$ existe et ne contient pas la valeur $S.elements[n_max]$: dans ce cas, $S1$ est un sous-ensemble de $S.elements[0:n_max]$ dont la somme des éléments vaut t . Ainsi, par hypothèse de récurrence, le second appel récursif de la ligne 8 renvoie **true**, donc `sous_somme(S, t, n_max)` renvoie **true**, ce qui est correct.
 - Soit un tel sous-ensemble n'existe pas : dans ce cas, par hypothèse de récurrence, les deux appels récursifs de la ligne 8 vont renvoyer **false**, et `sous_somme(S, t, n_max)` renvoie **false**, ce qui est correct.

9. (a)
- ```
bool partition(multiens S) {
 int t = somme(S);
 int n_max = S.taille - 1;
 if (t%2 == 1) {
 return false;
 }
 return sous_somme(S, t/2, n_max);
}
```

- (b) Dans le pire des cas, à chaque passage dans la ligne 8 de la fonction `sous_somme`, on effectuera deux appels récursifs, ce qui donnera une complexité temporelle exponentielle.  
Dans ce cas, les différents appels récursifs correspondent au test de chaque sous-ensemble  $S1$  de  $S$  dont les éléments apparaissent dans le même ordre que dans  $S$ .

### 1.3 Résolution par programmation dynamique

```
10. bool** init_false(int n, int m) {
 bool** T = (bool**)malloc(n * sizeof(bool*));
 for (int i = 0; i < n; i++) {
 T[i] = (bool*)malloc(m * sizeof(bool));
 for (int j = 0; j < m; j++) {
 T[i][j] = false;
 }
 }
 return T;
}
```

```
11. void free_matrice(bool** T, int n, int m) {
 for (int i = 0; i < n; i++) {
 free(T[i]);
 }
 free(T);
}
```

```
12. bool sous_somme_dyna(multiens S, int t) {
 bool** T = init_false(S.taille+1, t+1);
 // première colonne
 for (int i = 0; i < S.taille+1; i++) {
 T[i][0] = true;
 }
 for (int i = 1; i < S.taille+1; i++) {
 for (int j = 1; j < t+1; j++) {
 int j2 = j - S.elements[i-1];
 if (j2 < 0) {
 T[i][j] = T[i-1][j];
 }
 else {
 T[i][j] = T[i-1][j] || T[i-1][j2];
 }
 }
 }
 bool res = T[S.taille][t];
 free_matrice(T, S.taille+1, t+1);
 return res;
}
```

```
13. bool partition_dyna(multiens S) {
 int t = somme(S);
 if (t%2 == 1) {
 return false;
 }
 return sous_somme_dyna(S, t/2);
}
```

## 2 Tri rapide et algorithme de la médiane des médianes

### 2.1 Le tri rapide

14. 

```
let échange t i j =
 let x = t.(i) in
 t.(i) <- t.(j) ;
 t.(j) <- x
```

15. 

```
let partition_en_3 t p g d =
 let i_s = ref g in
 (* déplace les éléments < p au début *)
 for i = g to d-1 do
 if t.(i) < p then
 begin
 échange t i !i_s ;
 incr i_s
 end
 done ;
 (* déplace les éléments = p juste après *)
 let i_e = ref !i_s in
 for i = !i_s to d-1 do
 if t.(i) = p then
 begin
 échange t i !i_e ;
 incr i_e
 end
 done ;
 (!i_s, !i_e)
```

16. (a) 

```
let tri_rapide1 t =
 let rec aux g d = match g < d-1 with
 | false -> ()
 | true -> let ip,_ = partition_en_3 t t.(g) g d in
 aux g ip ; aux (ip+1) d
 in aux 0 (Array.length t)
```

- (b) Dans le pire des cas, (si le tableau est déjà trié), une portion du tableau de taille  $n$  est partitionnée en une portion de taille 1 et une portion de taille  $n - 1$ . La fonction `partition_en_3` ayant une complexité linéaire, la complexité du tri rapide vérifie donc :  $C(n) = C(n - 1) + \mathcal{O}(n)$ .  
D'où  $C(n) = \mathcal{O}(n^2)$ .

### 2.2 Calcul de la médiane d'un tableau

17. L'utilisation de la fonction `mediane` permettrait de garantir de partitionner une portion du tableau de taille  $n$  en deux portions de taille  $\frac{n}{2}$ . Ainsi, si la fonction `mediane` est de complexité  $f(n)$ , la complexité du tri rapide vérifierait :  $C(n) = 2 \cdot C(n/2) + f(n)$ .
- (a) Si  $f(n) = \mathcal{O}(n)$ , on obtient la même relation que pour la complexité du tri fusion, d'où  $C(n) = \mathcal{O}(n \log n)$ .
- (b) Si  $f(n) = \mathcal{O}(n^2)$ , les coûts des appels récursifs deviennent négligeables devant l'appel à la fonction `mediane`, et on obtient alors :  $C(n) = \mathcal{O}(n^2)$ .

18. (a) 

```
let mediane t k =
 let t' = Array.copy t in
 let rec aux g d =
 let p = t'.(g) in
 let i_s, i_e = partition_en_3 t' p g d in
 match i_s = k || (i_s < k && k < i_e) with
 | true -> t'.(k)
 | false when k < i_s -> aux g (i_s-1)
 | false -> aux i_e d
 in aux 0 (Array.length t)
```

- (b) Dans le pire des cas, (si le tableau est déjà trié), une portion du tableau de taille  $n$  est partitionnée en une portion de taille 1 et une portion de taille  $n - 1$ , et l'appel récursif se fera sur la portion de taille  $n - 1$ . La fonction `partition_en_3` ayant une complexité linéaire, la complexité de la fonction `mediane` vérifie donc :  $C(n) = C(n - 1) + \mathcal{O}(n)$ . D'où  $C(n) = \mathcal{O}(n^2)$ .

### 2.3 L'algorithme de la médiane des médianes

19. (a) Soit  $n$  la taille de  $\mathbf{t}$ . Parmi les  $\frac{n}{5}$  groupes, la moitié (donc  $\frac{n}{10}$ ) ont leur médiane plus petite que  $p$ . De plus, pour chacun de ces  $\frac{n}{10}$  groupes, deux éléments sont plus petits que leur médiane, donc 3 éléments de chacun de ces groupes sont plus petits que  $p$ . Ainsi, on a au moins  $\frac{3}{10}n$  éléments de  $\mathbf{t}$  inférieurs à  $p$ . De manière analogue, on a au moins  $\frac{3}{10}n$  éléments de  $\mathbf{t}$  supérieurs à  $p$ .
- (b) Dans le pire des cas, on va donc partitionner une portion du tableau de taille  $n$  en une portion de taille  $\frac{3}{10}n$  et une portion de taille  $\frac{7}{10}n$ , et l'appel récursif se fera dans le pire des cas sur la portion de taille  $\frac{7}{10}n$ .
- (c) Au final, la complexité de notre algorithme vérifiera la relation :

$$C(n) \leq \underbrace{C\left(\frac{1}{5} \cdot n\right)}_{\substack{\text{Calcul de la} \\ \text{médiane des} \\ \frac{n}{5} \text{ médianes}}} + \underbrace{C\left(\frac{7}{10} \cdot n\right)}_{\substack{\text{Recherche de l'élément} \\ \text{par récurrence sur un} \\ \text{morceau de la partition}}} + \underbrace{\alpha \cdot n}_{\substack{\text{calcul des} \\ \frac{n}{5} \text{ médianes} \\ + \text{partition}}}$$

- (d) Montrons par récurrence sur  $n$  que  $\forall n \in \mathbb{N}, C(n) \leq 10 \cdot \alpha \cdot n$ .
- Initialisation : quitte à prendre un  $\alpha$  assez grand, la propriété sera vraie pour les premières valeurs de  $n$ .
  - Hérédité :

$$\begin{aligned} C(n) &\leq C\left(\frac{1}{5} \cdot n\right) + C\left(\frac{7}{10} \cdot n\right) + \alpha \cdot n \\ &\leq 10 \cdot \alpha \cdot \frac{n}{5} + 10 \cdot \alpha \cdot \frac{7}{10} \cdot n + \alpha \cdot n \\ &= (2 + 7 + 1) \cdot \alpha \cdot n \\ &= 10 \cdot \alpha \cdot n \end{aligned}$$

La complexité de cet algorithme sera donc bien **linéaire**.

20. 

```
let partition5 t g d = (* on utilise le tri par sélection *)
 for i = g to d do
 let im = ref i in
 for j = i+1 to d do
 if t.(j) < t.(!im) then im := j
 done ;
 echange t i !im
 done ;
 (g+d)/2
```

21. (a) 

```
val pivot : 'a array -> int -> int -> int = <fun>
val select : 'a array -> int -> int -> int -> int = <fun>
```

- (b) La fonction `pivot` parcourt la portion donnée du tableau par groupes de 5, et calcule la médiane de chacun des groupes à l'aide de `partition5`. Elle déplace au fur et à mesure ces médianes au début de la portion du tableau, et calcule la médiane de ces médianes grâce à l'appel à `select` (on renvoie l'indice de cet élément).

La fonction `select t g d k` renvoie le  $k$ -ème plus petit élément de  $t[g:d]$ . Pour cela, elle utilise la fonction `pivot` pour calculer l'indice de la médiane des médianes de  $t[g:d]$ , puis s'en sert comme pivot pour partitionner la portion avec `partition_en_3`. Au final, les éléments du tableau se situant entre les indices  $i_s$  et  $i_e$  sont à leur place finale (et sont tous égaux), donc :

- Si  $i_s = k$  ou  $(i_s < k < i_e)$ , le  $k$ -ème plus petit élément du tableau est désormais à l'indice  $k$ , on peut donc renvoyer  $k$ ;
- sinon :
  - si  $k < i_s$ , l'élément recherché se trouve strictement avant l'indice  $i_s$ ;
  - sinon,  $i_e \leq k$ , et l'élément recherché se trouve après l'indice  $i_e$ ;

Dans les deux cas, on fait un appel récursif sur la portion du tableau adéquate.

- (c) Puisque les fonctions `pivot` et `select` s'appellent mutuellement, il suffit de montrer par exemple que `select` termine pour montrer du même coup que `pivot` termine.

Notons  $n = d - g$ . L'appel `select t g d k` effectue l'appel `pivot t g d` (ligne 18) qui va lui-même effectuer un appel récursif à `select` sur une portion de taille  $n' = d' - g < d - g = n$  (ligne 14). Elle effectuera ensuite une autre appel récursif (ligne 23 ou 24) sur une portion de taille  $\leq \frac{7}{10}n$  (d'après la question 19b). Tous les appels récursifs à `select` se font donc bien sur des portions du tableau strictement plus petites que  $n$ .

En plus des appels récursifs, on effectue un appel à la fonction `partition_en_3` (ligne 20) qui termine, et une boucle `while` (lignes 5 à 11) qui termine car  $i$  augmente strictement à chaque passage dans la boucle.

Ainsi, la fonction `select` (et la fonction `pivot`) termine(nt).

- (d) L'appel à `partition_en_3`, ainsi que la boucle `while` des lignes 5 à 11, s'effectuent en  $\mathcal{O}(n)$  (où  $n = d - g$ ), car les appels à `partition5` s'effectuent en  $\mathcal{O}(1)$  (morceaux de taille au plus 5). De plus, l'appel récursif à `select` de la ligne 14 s'effectue sur une portion de taille  $\frac{n}{5}$ , et l'appel récursif à `select` de la ligne 23 ou 24 s'effectue sur une portion de taille au plus  $\frac{7}{10}n$ .

Ainsi, d'après la question 19d, la fonction `select` est de complexité **linéaire**.

22. 

```
let selection_rapide t k =
 let t' = Array.copy t in
 let i = select t' 0 (Array.length t' - 1) k in
 t'.(i)
```

23. 

```
let tri_rapide2 t =
 let rec aux g d = match g<d-1 with
 | false -> ()
 | true ->
 let p = selection_rapide t ((g+d)/2) in
 let i_s, _ = partition_en_3 t p g d in
 aux g i_s ;
 aux (i_s+1) d
 in aux 0 (Array.length t)
```