Outils logiques et algorithmiques – TP/DM – Mobiles de Calder

Instructions. *Ce TP est commencé en classe lors de la séance du 22 du 25 avril selon les groupes. Il doit ensuite être complété à la maison pour le 10 mai. Le travail est individuel, et est à rendre sur ecampus, dans la rubrique correspondant à votre groupe de TD. Le rendu doit prendre la forme d'un unique fichier* `.ml` *portant votre nom, dans lequel les réponses aux questions doivent apparaître dans l'ordre. Lorsqu'une question demande d'écrire une fonction caml, l'introduire par un commentaire donnant le numéro de la question, puis donner le code. Par exemple :*

```
(* Question 7 *)
let rec size t = match t with
  ...
```

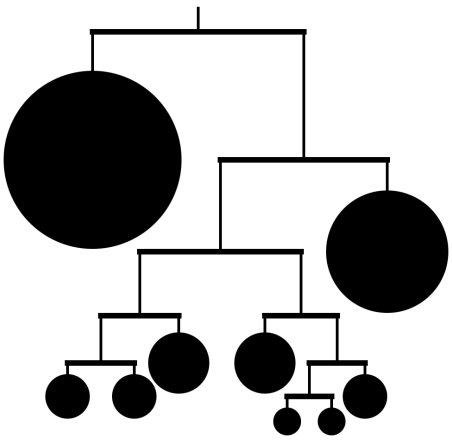
Lorsqu'une question demande une réponse rédigée en français, donner l'intégralité de la réponse dans un commentaire. Par exemple :

```
(* Question 5
   Montrons par récurrence structurelle sur m que
     taille(m) = barres(m)+1
   Cas 0
     ...
   Cas
     ...
 *)
```

Après chaque question de code, il est judicieux d'inclure quelques tests. Les types imposés pour les structures de données et les fonctions sont à respecter ***impérativement*** *(sinon, c'est zéro).*

Contexte

Un *mobile* est formé par un ensemble d'objets, suspendus à des barres elles-mêmes suspendues en équilibre à d'autres barres, et ainsi de suite jusqu'à un unique point de suspension auquel pend l'ensemble de la structure.



Ces objets aériens ont fait la réputation du sculpteur américain Alexander Calder. Dans ce TP, nous allons voir ces mobiles comme des arbres binaires respectant certaines conditions d'équilibre, et réaliser des algorithmes permettant de générer des mobiles aléatoires et les afficher joliment.

1. Mobiles équilibrés

On décrit les mobiles comme des arbres binaires pouvant prendre l'une des deux formes suivantes :

— soit un objet `O(`w`)` de poids w,

— soit une barre horizontale `B(`m_1`, `m_2`)` aux extrémités de laquelle pendent deux sous-mobiles m_1 et m_2.

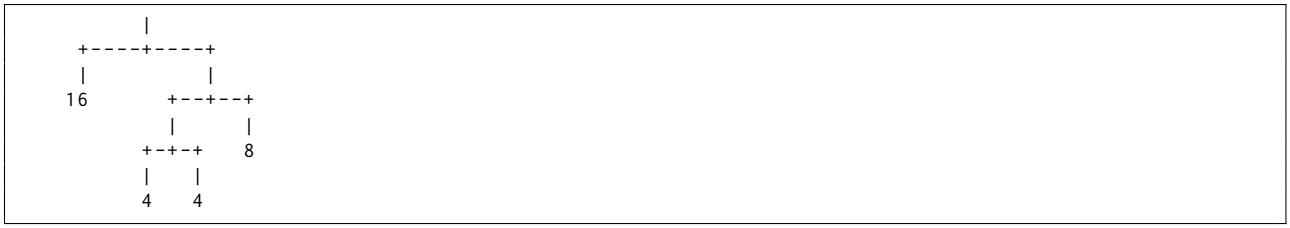
On les représente en caml à l'aide du type algébrique suivant :

```
type mobile =
  | O of int
  | B of mobile * mobile
```

Le *poids* d'un mobile est la somme des poids de tous ses objets. Un mobile est considéré comme *équilibré* lorsque, pour chacune de ses barres `B(`m_1`, `m_2`)`, les sous-mobiles m_1 et m_2 ont exactement le même poids. *Note : pour simplifier cette étude, on néglige le poids des barres et des fils auxquels sont suspendus les éléments.* On définit en outre la *taille* d'un mobile comme le nombre d'objets, et sa *hauteur* comme le nombre maximal de barres à traverser pour aller du point de suspension à un objet. Ainsi le mobile dessiné ci-dessus a une taille de 9 et une hauteur de 6.

Questions.

1. Donner une expression caml représentant le mobile suivant :



2. Dessiner le mobile m défini en caml par

```
let m = B(O(32), B(B(B(B(O(2), O(2)), O(4)), B(O(4), B(B(O(1), O(1)), O(2)))), O(16)))
```

   En ASCII, comme l'exemple de la question précédente.

3. On considère la famille de mobiles t_k définie par :

$$\begin{cases} t_0 &= \texttt{O(1)} \\ t_{k+1} &= \texttt{B(}t_k\texttt{, }t_k\texttt{)} \end{cases}$$

   (a) Dessiner les mobiles t_0, t_1, t_2 et t_3.

   (b) Exprimer la taille, la hauteur et le poids de t_k en fonction de k.

   (c) Écrire une fonction `t: int -> mobile` telle que l'appel `t k` génère le mobile t_k.

4. On considère la famille de mobiles u_k définie par :

$$\begin{cases} u_0 &= \texttt{O(1)} \\ u_{k+1} &= \texttt{B(O(}2^k\texttt{), }u_k\texttt{)} \end{cases}$$

   (a) Dessiner les mobiles u_0, u_1, u_2 et u_3.

   (b) Exprimer la taille, la hauteur et le poids de u_k en fonction de k.

   (c) Écrire une fonction `u: int -> mobile` telle que l'appel `u k` génère le mobile u_k.

5. Démontrer que tout mobile m de taille $|m|$ contient exactement $|m| - 1$ barres.

6. La taille $|m|$ d'un mobile m vous semble-t-elle être une mesure raisonnable en fonction de laquelle exprimer la complexité des algorithmes manipulant les mobiles ? Justifiez, par exemple en comparant avec d'autres mesures possibles.

7. Écrire trois fonctions `size`, `height` et `weight`, toutes de type `mobile -> int`, qui prennent en paramètre un mobile m et renvoient respectivement sa taille, sa hauteur et son poids.

8. Voici la définition d'une fonction `balanced: mobile -> bool` testant si un mobile est équilibré :

```
let rec balanced = function
  | L _       -> true
  | N(t1, t2) -> balanced t1 && balanced t2 && weight t1 = weight t2
```

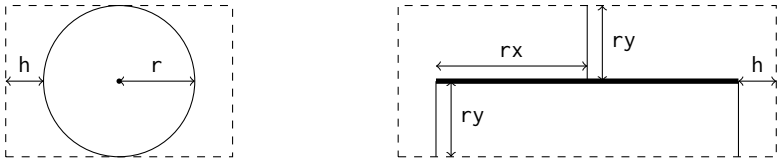
   (a) Quelle est sa complexité sur les mobiles u_k définis ci-dessus ? *Exprimer le résultat sous la forme d'un ordre de grandeur du nombre d'opérations en fonction de k, puis en fonction de la taille du mobile.*

   (b) Même question pour les mobiles t_k.

9. Donner une nouvelle définition pour une fonction `balanced': tree -> bool` vérifiant si un arbre est équilibré en un temps linéaire en la taille de l'arbre.
   Indication : vous pouvez utiliser une fonction auxiliaire qui aura un type légèrement différent.

2. Dimensionnement des éléments

On affichera un mobile en représentant chaque objet par un disque avec un certain rayon `r`, et chaque barre par la combinaison d'une barre horizontale et de trois portions de fil : un auquel la barre est suspendue par son milieu, et deux suspendus aux extrémités de la barre. On notera `rx` la moitié de la longueur de la barre, et `ry` la longueur des portions de fil. *Note : quand une barre est suspendue sous une autre barre, les fils qui se correspondent sont mis bout à bout (voir schémas plus bas).* En outre, on donne à chaque élément une marge horizontale constante `h`. Ainsi, un objet de rayon `r` est incrit dans un rectangle de hauteur `2r` et de largeur `2(r + h)`, et une barre et les portions de fil associées sont inscrites dans un rectangle de hauteur `2ry` et de largeur `2(rx + h)`.



Pour la suite, on définit la marge horizontale comme valant 1 pixel.

```
let h = 1
```

Le travail principal consistera à calculer ces différentes dimensions de sorte à pouvoir afficher le mobile sans que les éléments se recouvrent. En l'occurrence, on fera en sorte que les rectangles contenant chaque éléments soient tous disjoints. Le résultat de ce calcul prendra la forme d'un *mobile annoté*, dont chaque objet et chaque barre contient les valeurs calculées pour `r`, `rx` et `ry`.

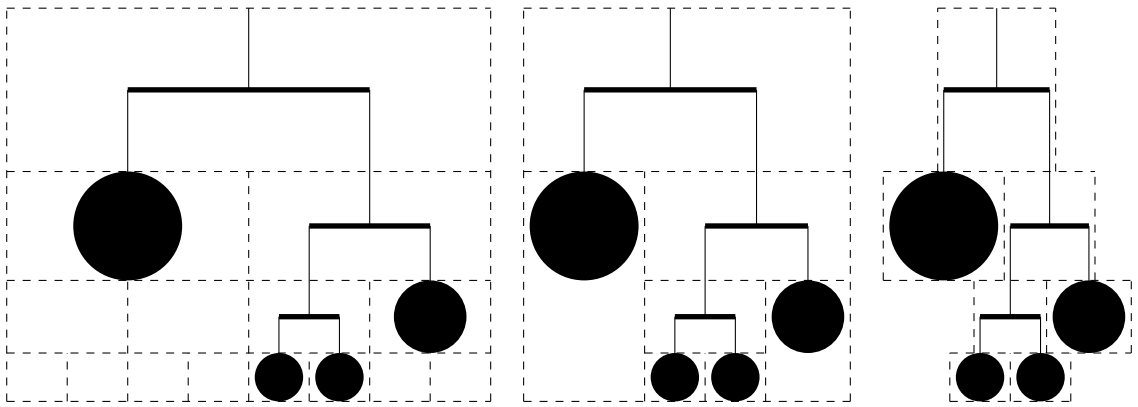
```
type annot_mobile =
  | AO of int                                          (*  AO(r)               *)
  | AB of int * int * annot_mobile * annot_mobile      (*  AB(rx, ry, m1, m2)  *)
```

Dans cette partie nous allons voir trois modes du calcul, du plus simple au plus élégant.

Uniforme : on prévoit une largeur fixe pour tout mobile d'un poids w donné, correspondant à la juxtaposition de w objets de poids 1. Chaque élément est ensuite centré dans l'espace horizontal qui lui est réservé.

Séparé : pour afficher `B(`m_1`,` m_2`)`, on juxtapose les deux sous-mobiles m_1 et m_2 en donnant à chacun un espace correspondant à sa largeur effective.

Emboîté : pour afficher `B(`m_1`,` m_2`)`, on juxtapose les deux sous-mobiles m_1 et m_2, en les approchant autant que possible sans que leurs éléments se recouvrent.



Dimensions des objets Pour un rendu réaliste, on donnera à chaque objet un rayon proportionnel à la racine carrée de son poids. Voici la définition d'une fonction `sqrt: int -> int` calculant la partie entière de la racine carrée de l'entier pris en argument.

```
let sqrt n =
  let rec loop r s =
  (*  Precondition: s = r*r  *)
    if s > n then r - 1
    else loop (r+1) (s+2*r+1)
  in
  loop 0 0
```

On fixe un *rayon unité*, qui définit le rayon d'un objet de poids 1, d'une valeur de 4 pixels (cela évite des aberrations dans l'affichage d'objets trop petits). Le calcul du rayon en fonction du poids pourra donc être fait par la fonction `radius: int -> int` définie ci-dessous.

```
let unit_radius = 4
let sq_unit = unit_radius * unit_radius
let radius w = sqrt (sq_unit * w)
```

En outre, cette valeur `radius` w définira également la dimension `ry` d'une barre formant un mobile de poids total w (à cette barre sont donc suspendus deux sous-mobiles, chacun de poids $w/2$). On assure ainsi que tous les éléments à un même étage du mobile ont précisément la même dimension verticale.

10. Démontrer que tous les appels à la fonction auxiliaire `loop` respectent bien la précondition énoncée dans le code.
11. Pourquoi n'a-t-on pas défini `radius w` par l'expression plus simple `unit_radius * sqrt w` ?

Questions : mode uniforme. Dansn le mode uniforme, la largeur d'un mobile m ne dépend que de son poids w, et peut être calculée comme la largeur de w éléments de poids 1 mis côte à côté.

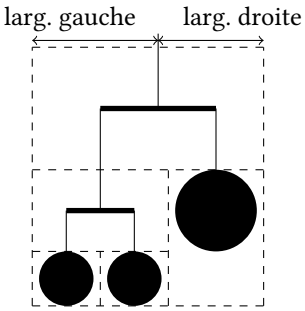
12. Définir une fonction `uniform_width: int -> int` telle que `uniform_width` w calcule la largeur d'un mobile de poids w.
13. Définir une fonction `annot_uniform: mobile -> annot_mobile` qui prend en entrée un mobile et en construit une version annotée en suivant le mode uniforme.
    L'annotation du mobile

    ```
    B(B(O(1), O(1)), O(2))
    ```

    devra produire le mobile annoté

    ```
    AB(10, 8, AB(5, 5, AO(4), AO(4)), AO(5))
    ```

Questions : mode séparé. Pour placer correctement les éléments, nous allons devoir calculer pour chaque sous-mobile sa *largeur gauche* et sa *largeur droite*, qui mesurent la différence d'abscisse entre le fil auquel est suspendu un mobile et ses extrémités respectives gauche et droite. Notez que le point de suspension correspond au centre de la barre principale, qui peut ne pas être au centre du mobile lui-même : les largeurs gauche et droite peuvent donc être différentes l'une de l'autre.



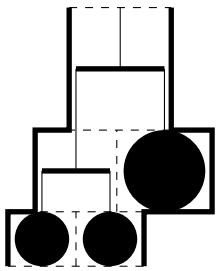
14. Définir une fonction `left_width: annot_mobile -> int` calculant la largeur gauche d'un mobile annoté. Définir symétriquement une fonction `right_width`. *Vous pouvez commencer par définir des équations, pouvant faire intervenir la largeur gauche d'un ou plusieurs sous-mobile. Ne pas oublier de compter la marge* `h`.
15. En déduire une fonction `annot_sep: mobile -> annot_mobile` produisant un mobile annoté en suivant le mode séparé.
    L'annotation du mobile

    ```
    B(B(O(1), O(1)), O(2))
    ```

    devra produire le mobile annoté

    ```
    AB(8, 8, AB(5, 5, AO(4), AO(4)), AO(5))
    ```

Questions : mode emboîté. Pour placer correctement les éléments, nous allons calculer pour chaque élément son *profil gauche* et son *profil droit*, c'est-à-dire la liste des différences d'abscisse entre le point de suspension et le point du mobile la plus à gauche (resp. le plus à droite) à chaque étage. Dans le schéma ci-dessous, les profils gauche et droit sont représentés chacun par une ligne brisée noire.



Remarquez que le profil droit peut être défini par des éléments du sous-mobile gauche, aux étages où le sous-mobile droit n'a pas d'éléments, et qu'un profil peut même prendre des valeurs négatives à certains étages (lorsque l'élément le plus à droite à cet étage est à gauche du points de suspension). Il en irait de manière symétrique pour le profil gauche. Pour déterminer la longueur de la barre à laquelle seront suspendus deux mobiles m_1 et m_2, il faut ensuite comparer le profil droit du mobile gauche m_1 et le profil gauche du mobile m_2.

16. Définir une fonction `left_profile: int -> annot_mobile -> int list` telle que l'application `left_profile` b m renvoie la liste `[`$b + d_0$`; ...; `$b + d_k$`]` où d_i est la distance entre le point de suspension de m et l'extrémité gauche de l'étage numéro i du mobile m. Définir symétriquement une fonction `right_profile`.
Note : l'étage numéro 0 *est la racine du mobile, l'étage numéro k est celui de l'objet le plus profond. Le paramètre b est un décalage de base qui va vous aider à calculer le profil d'un mobile composé à partir des profils de ses sous-mobiles. Vous pouvez définir une fonction auxiliaire pour combiner les profils de deux sous-mobiles.*
Partant du mobile annoté

```
AB(6, 8, AB(5, 5, AO(4), AO(4)), AO(5))
```

vous devrez obtenir les résultats suivants.

```
left_profile 0 am = [7; 12; 16]
right_profile 0 am = [7; 12; 4]
```

17. En déduire une fonction `annot_nest: mobile -> annot_mobile` produisant un mobile annoté en suivant le mode emboîté. *Vous pouvez définir une fonction auxiliaire pour convertir les profils de deux mobiles frères en une distance.*
L'annotation du mobile

```
B(B(O(1), O(1)), O(2))
```

devra produire le mobile annoté

```
AB(6, 8, AB(5, 5, AO(4), AO(4)), AO(5))
```

3. Affichage

Pour permettre la visualisation d'un mobile, nous allons générer un petit fichier graphique au format SVG [1]. C'est un format suffisamment simple pour être manipulé à la main, et reconnu nativement par les navigateurs web. Ainsi, une fois un fichier `.svg` créé, il suffit de l'ouvrir avec votre navigateur favori pour visualiser le résultat. Voici un exemple affichant un mobile minimal.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <line x1="60" y1="5" x2="60" y2="1" stroke="black"/>
  <line x1="39" y1="5" x2="81" y2="5" stroke-width="2px" stroke="black"/>
  <line x1="40" y1="5" x2="40" y2="25" stroke="black"/>
  <line x1="80" y1="5" x2="80" y2="25" stroke="black"/>
  <circle cx="40" cy="25" r="12" fill="black"/>
  <line x1="80" y1="25" x2="80" y2="22" stroke="black"/>
  <line x1="69" y1="25" x2="91" y2="25" stroke-width="2px" stroke="black"/>
  <line x1="70" y1="25" x2="70" y2="40" stroke="black"/>
  <line x1="90" y1="25" x2="90" y2="40" stroke="black"/>
  <circle cx="70" cy="40" r="8" fill="black"/>
  <circle cx="90" cy="40" r="8" fill="black"/>
</svg>
```

Pour générer de tels fichiers, nous allons simplement demander au programme d'écrire sur la sortie standard à l'aide de la bibliothèque `Printf`, puis rediriger vers le fichier souhaité. Ainsi, une fois que vous avez compilé votre programme en un exécutable `mobile-gen.exe`, il suffit d'exécuter

```
$ ./mobile-gen.exe > mobile.svg
```

dans un terminal pour créer un fichier `mobile.svg` contenant ce que votre programme a affiché avec `Printf.printf`. *Note : si un fichier* `mobile.svg` *existe déjà avant cette opération, son contenu est écrasé et ne pourra pas être récupéré.*

Pour générer ces différents éléments, voici quatre fonctions que vous pourrez réutiliser. Les trois fonctions `print_object`, `print_bar` et `print_thread` affichent les codes SVG correspondant aux différents éléments d'un mobile, en fonction de coordonnées et de dimensions. La fonction `generate_svg` prend en paramètre une fonction `printer: mobile -> unit` et un mobile `t`. Elle génère l'entête et la fin du fichier SVG, en insérant entre les deux l'affichage généré par `printer`.

1. `https://developer.mozilla.org/fr/docs/Web/SVG`

```
open Printf

(* Objet, centre (cx,cy) et rayon r *)
let print_object cx cy r =
  printf "␣␣<circle␣cx=\"%d\"␣cy=\"%d\"␣r=\"%d\"␣fill=\"black\"/>\n" cx cy r

(* Barre horizontale, extrémité gauche (x1,y1) et droite (x2,y2) *)
let print_bar x1 y1 x2 y2 =
  printf
   "␣␣<line␣x1=\"%d\"␣y1=\"%d\"␣x2=\"%d\"␣y2=\"%d\"␣stroke-width=\"2px\"␣stroke=\"black\"/>\n"
   (x1-1) y1 (x2+1) y2

(* Fil vertical, entre (x1,y1) et (x2,y2) *)
let print_thread x1 y1 x2 y2 =
  printf "␣␣<line␣x1=\"%d\"␣y1=\"%d\"␣x2=\"%d\"␣y2=\"%d\"␣stroke=\"black\"/>\n" x1 y1 x2 y2

(* Affichage d'un mobile *)
let generate_svg printer t =
  printf "<svg␣xmlns=\"http://www.w3.org/2000/svg\">\n";
  printer t;
  printf "</svg>\n"
```

À noter : les coordonnées sont exprimées en pixels, le point de coordonnées $(0, 0)$ *étant le point en haut à gauche de la page. Ainsi, la dimension « `x` » va de la gauche vers la droite, et la dimension « `y` » va du haut vers le bas.*

Questions.

18. Définir une fonction `print_annot_mobile: int -> int -> annot_mobile -> unit` telle que l'appel `print_mobile` x y m génère les instructions d'affichage pour le mobile annoté m, en prenant comme point de suspension le point de coordonnées (x, y). *Indication : cette fonction pourra être récursive.*
19. Définir une fonction `print_mobile: mobile -> unit` qui génère le fichier SVG complet pour le mobile pris en entrée, après l'avoir annoté selon le mode de votre choix.

4. Génération de mobiles équilibrés aléatoires

On propose deux algorithmes simple pour la génération de mobiles aléatoires.

Génération par fission. Pour générer un mobile équilibré de poids 2^k :

- si $k = 0$, générer un objet de poids 1,
- sinon, selon le résultat d'un tirage aléatoire, soit générer un objet de poids 2^k, soit générer une barre combinant deux sous-mobiles aléatoires de poids 2^{k-1}.

Génération par fusion. Pour générer un mobile équilibré de poids 2^k :

- si $k = 0$, générer un objet de poids 1,
- sinon, générer deux mobiles aléatoires m_1 et m_2 de poids 2^{k-1} puis :
  - si m_1 et m_2 sont tous les deux des objets, selon le résultat d'un tirage aléatoire soit les fusionner en un seul objet, soit générer une barre combinant m_1 et m_2,
  - dans tous les autres cas, générer une barre combinant m_1 et m_2.

Note : caml vous fournit une fonction `Random.int: int -> int` telle que `Random.int` k renvoie un entier aléatoire dans l'intervalle $[0, k[$. Ainsi, si vous voulez réaliser un tirage aléatoire avec une probabilité de 30%, il suffit d'un code de la forme `if Random.int 10 < 3 then ... else ...`.

Questions.

20. Réaliser une fonction `gen_fission: int -> mobile` telle que l'appel `gen_fission` k génère un mobile équilibré aléatoire de poids 2^k avec la méthode de génération par fission.
21. Réaliser une fonction `gen_fusion: int -> mobile` telle que l'appel `gen_fusion` k génère un mobile équilibré aléatoire de poids 2^k avec la méthode de génération par fusion.

Énumérations et génération uniforme. Pour générer un mobile aléatoire de poids 2^k de manière *uniforme*, c'est-à-dire en donnant à chaque mobile équilibré la même probabilité d'être tiré, il faut être capable d'énumérer l'ensemble des mobiles équilibrés d'un poids donné.

Questions.

22. On note $\#_k$ le nombre de mobiles équilibrés de poids 2^k. On affirme que ce nombre vérifie les équations suivantes :

$$\begin{aligned} \#_0 &= 1 \\ \#_{k+1} &= 1 + (\#_k)^2 \end{aligned}$$

    Justifier ces équations, et en déduire une fonction caml `count: int -> int` telle que `count` k calcule $\#_k$.

23. Montrer que pour tout k on a $\#_k \geq 2^k$.

24. Écrire une fonction caml `enum: int -> mobile list` telle que `enum` k renvoie la liste de tous les mobiles équilibrés de poids 2^k (peu importe l'ordre). *Indication : la longueur de cette liste doit correspondre au nombre calculé par la fonction* `count` *!*

25. En déduire une fonction `random_mobile: int -> mobile` telle que `random_mobile` k renvoie un mobile tiré au hasard de manière uniforme parmi les mobiles de poids 2^k.

Défis bonus

26. Écrire une fonction `random_mobile'` répondant à la même spécification que la précédente, mais sans construire la liste de tous les mobiles.

27. Écrire une fonction `mk_mobile: int list -> mobile option` qui prend en entrée une liste de poids, et qui essaie de construire un mobile m dont les objets ont les poids donnés par la liste (dans l'ordre de gauche à droite). La fonction doit renvoyer :

    — `Some` m s'il existe un tel mobile m,

    — `None` s'il n'existe pas de tel mobile.

    Indication : il faut d'abord transformer la liste de poids `[`w_1`; ...;` w_n`]` *en une liste de mobiles* `[O(`w_1`); ...; O(`w_n`)]` *ne contenant que des objets, puis tenter de regrouper les objets en mobiles de plus en plus grands. Ceci est faisable en un temps linéaire en la taille de la liste.*

28. Écrire une fonction `print_mobile: mobile -> unit` qui prend en paramètre un mobile et l'affiche en ASCII. *Indication : l'affichage ligne par ligne n'est pas directement compatible avec la structure récursive naturelle des mobiles. Il va falloir trouver une astuce pour s'en sortir quand même !*