1. `mystere` doit renvoyer un `int` car `max_int` est un `int`. L'argument de `mystere` doit être un `int list` à cause du cas `| [a] -> a`. Donc `mystere : int list -> int`.

2. À chaque appel récursif de `mystere`, la taille de la liste en argument diminue de 1. Si 'z' est une liste de taille n, il faut donc n appels de `mystere` avant d'arriver sur le cas de base de la liste réduit à un élément.

3. Posons, pour $n \geq 1$:

$$H_n : \text{Si z est une liste d'entiers de taille } n, \text{ mystere z renvoie le minimum de z}$$

   On peut démontrer H_n par récurrence sur n.

4. Comme `mystere` est récursif terminal (l'appel récursif est la dernière opération réalisée), la fonction est automatiquement dérécursifiée. Il n'y a donc pas d'espace mémoire relatif à la pile d'appel. De plus, une liste étant persistante, chaque ajout à une liste (`...::y`) demande une complexité en espace O(1) (la mémoire est partagée : il n'y a pas besoin de copier la liste).
   Si on prend en compte l'espace mémoire pour stocker l'entrée (d'après l'énoncé), on obtient une complexité mémoire $O(n)$.

5. $d_{i,0} = i$ (il faut supprimer les i caractères de b) et $d_{0,j} = j$ (il faut ajouter les j caractères de c).

6. Si $b_i = c_j$, une solution optimale consiste à transformer $b_{1..i-1}$ en $c_{1..j-1}$ et laisser $b_i = c_j$ inchangé. Donc $d_{i,j} = d_{i-1,j-1}$. Sinon, il faut ajouter, supprimer ou substituer le dernier caractère :

$$d_{i,j} = \begin{cases} d_{i-1,j-1} \text{ si } b_i = c_j \\ 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \text{ sinon} \end{cases}$$

7.

```ocaml
let array_of_mot w =
    let t = Array.make (List.length w) ' ' in
    let rec aux l n = match l with
        | [] -> ()
        | e::q -> t.(n) <- e; aux q (n + 1) in
    aux w 0;
    t
```

8.

```ocaml
let distance b c =
    let b, c = array_of_mot b, array_of_mot c in
    let nb, nc = Array.length b, Array.length c in
    let d = Array.make_matrix nb nc (-1) in
    let rec aux i j = (* renvoie d(i, j) *)
        if i = -1 then j + 1
        else if j = -1 then i + 1
        else if d.(i).(j) <> -1 then d.(i).(j)
        else if b.(i) = c.(j) then (d.(i).(j) <- aux (i - 1) (j - 1); d.(i).(j))
        else (d.(i).(j) <- 1 + min (aux (i - 1) j) (min (aux (i - 1) (j - 1)) (aux i (j - 1)));
              d.(i).(j)) in
    aux (nb - 1) (nc - 1)
```

9. `distance b c` est en $O(n \times m)$ car chaque appel récursif remplit une case de `d`. Donc `List.filter (fun c -> (distance b c) <` est en $O(m \times n_{max} \times \gamma)$ où γ est la longueur de la liste `lc`.

10. Il suffit de considérer les lettres x_0, ..., x_{r-1} qui ne sont pas modifiées dans b dans une transformation en c et $\nvdash$, ..., $\diagdown$ comme les mots entre ces lettres.

11. cap, copie, copier, copies, cor, corde, corne, correct, correcte.

12.
    - Table de hachage avec insertion en O(1) en moyenne (sous hypothèse de hachage uniforme).
    - Arbre binaire de recherche équilibré (exemple : arbre rouge-noir) dont les éléments sont des couples (clé, valeur) avec insertion en $O(\log(n))$.

13.

```ocaml
let trie_vide = Node CharMap.empty
let trie_mot_vide = Node (CharMap.add '$' trie_vide CharMap.empty)
```

14.

```ocaml
let trie_singleton x = Node (CharMap.add x trie_mot_vide CharMap.empty)
```

15.

```ocaml
let rec trie_mem (c : char list) (Node tcm) = match c with
    | [] -> CharMap.find_opt '$' tcm <> None
    | e::q -> match CharMap.find_opt e tcm with
                | None -> false
                | Some a -> trie_mem q a
```

16.

```ocaml
let rec trie_add c (Node tcm) = match c with
    | [] -> Node (CharMap.add '$' trie_vide tcm)
    | e::q ->
        let t_e = match CharMap.find_opt e tcm with
            | None -> trie_vide
            | Some a -> a in
        Node (CharMap.add e (trie_add q t_e) tcm)
```

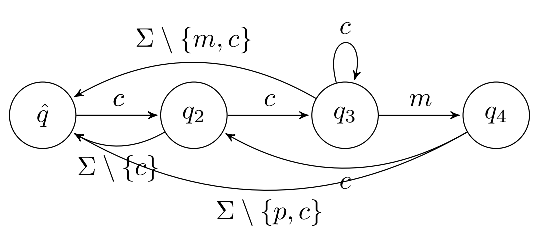
17. Il y a toujours un seul exemplaire de `trie_vide`, partagé par tous les nœuds qui n'ont pas de fils. En effet, comme `char_map` est persistant (donc non modifiable), il n'est pas utile d'avoir plusieurs exemplaires de `trie_vide`.

18.

```ocaml
let rec trie_trim (Node tcm:trie) : trie =
  let filtre (x:char) (y:trie) : trie option =
    let y' = trie_trim y in
    if x = '$' && y' = trie_vide then None
    else Some y'
  in
  Node(CharMap.filter_map filtre tcm)
```

19. Un mot à distance au plus k de b peut-être obtenu en choisissant au plus i lettres de b ($\binom{|b|}{i}$ possibilités) et les remplacer par une autre lettre de $\Sigma \backslash \{\$\}$ ($|\Sigma| - 2$ possibilités). On a donc un nombre de mots possibles, pour $|b| \geq \Sigma$: $\sum_{i=0}^{k} \binom{|b|}{i} (|\Sigma| - 2)^i \leq \sum_{i=0}^{k} |b|^i = (|b| + 1)^{k+1} = O(|b|^k)$.

20.



21.

```
let rec trie_filter (qchapeau:etat) (delta:syst_trans) (Node tcm:trie) : trie =
  let filtre (x:char) (y:trie) : trie option =
    match delta qchapeau x with
      | None -> None
      | Some q -> Some (trie_filter q delta y)
  in
  Node(CharMap.filter_map filtre tcm)
```

22. `trie_filter qchapeau delta t` effectue un nombre constant d'opération sur chaque nœud de `t`, donc est en $O(n)$ où n est le nombre de nœuds de `t`.