

## Mini-projet algorithmique en C & OCaml

### Reverse d'une liste/tableau et Recherche Dichotomique

#### Objectifs

Ce mini-projet vous permettra de pratiquer à la fois :

- la manipulation de tableaux (C) et de listes (OCaml) ;
- les fonctions récursives et itératives ;
- une réflexion algorithmique sur deux exercices fondamentaux :
  1. **reverse** : inversion d'une liste/tableau ;
  2. **recherche dichotomique** : recherche efficace dans un tableau trié.

Les deux exercices sont réalisables en C et en OCaml. Vous devrez produire les deux versions.

## 1 Reverse d'une liste/tableau

L'objectif est d'écrire une fonction qui, donnée une liste (OCaml) ou un tableau (C), produit son inversion.

Qu'est-ce que *renverser* une liste ? Exemple :  $[5, 2, 9, 1] \longrightarrow [1, 9, 2, 5]$ .

**Question 0** : Cette opération modifie-t-elle la structure ? Le nombre d'éléments ?

### 1.1 Version C : inversion d'un tableau

On considère un tableau d'entiers `int a[n]`.

**Exercice 1** : Compléter la fonction suivante (C) :

Listing 1 – Reverse d'un tableau en C

```
1 void reverse(int *a, int n) {  
2     /* A compléter :  
3         - utiliser deux indices i et j  
4         - echanger a[i] et a[j]  
5         - avancer i, reculer j  
6         - s'arreter quand i >= j  
7     */  
8 }
```

**Question 1** :

1. Sur un tableau de taille 9, combien d'échanges faut-il ?
2. Que se passe-t-il si vous utilisez une boucle allant simplement de 0 à n-1 ?

## 1.2 Version OCaml : inversion d'une liste

On rappelle le type de listes OCaml : `type 'a list = [] | x :: l`

**Exercice 2 :** Écrire une fonction **réursive** qui renverse une liste.

Listing 2 – Reverse d'une liste en OCaml

```

1 let rec reverse l =
2   (* A compléter :
3    - écrire une fonction auxiliaire tail-réursive avec un accumulateur
4    - accumulator = liste retournée progressivement
5   *)
6   ;;

```

**Question 2 :**

1. Pourquoi une version naïve `reverse (xs @ [x])` est-elle inefficace ?
2. Pourquoi l'accumulateur rend-il la fonction *tail-réursive* ?

## 1.3 Tests

- liste vide ;
- liste à un élément ;
- liste de 8–10 éléments.

**Exercice 3 :** En C comme en OCaml, vérifier que `reverse(reverse(l)) = l`.

## 1.4 Vérifier si une liste/tableau est un palindrome

### Liste palindrome

Une structure (liste ou tableau) est dite **palindrome** si elle se lit de la même façon de gauche à droite et de droite à gauche.

Esemples : `[1, 2, 3, 2, 1]` est un palindrome et `[1, 2, 3, 4]` ne l'est pas.

L'idée générale pour tester si une liste/tableau est un palindrome consiste à :

1. produire son inversion (grâce à la fonction `reverse` écrite précédemment) ;
2. comparer la structure initiale à sa version inversée.

**Exercice 4** Écrire en C une fonction qui teste si un tableau d'entiers est un palindrome.

**Question 4 :**

1. Pourquoi faut-il réaliser une *copie* du tableau avant de l'inverser ?
2. Serait-il possible d'écrire une fonction **sans copie**, en comparant `a[i]` et `a[n-1-i]` ?

**Exercice 5** Écrire en OCaml une fonction qui teste si une liste est un palindrome.

**Question 5 :**

1. Comment comparer deux listes ? (Indice : écrire une fonction réursive `egal l1 l2`.)
2. Est-il possible d'écrire une version sans utiliser `reverse` ?

**Exercice 6 :** Montrer expérimentalement que `est_palindrome(l) = 1  $\iff$  l = reverse(l)`.

## 2 Partie B — Recherche Dichotomique

### Recherche dichotomique

La **recherche dichotomique** (ou *binary search*) est une méthode de recherche extrêmement efficace pour savoir si une valeur  $x$  est présente dans un tableau **trié**.

L'idée fondamentale est la suivante : au lieu de parcourir le tableau élément par élément, on regarde directement l'élément du milieu et, selon sa valeur, on élimine la moitié du tableau d'un seul coup.

Concrètement :

1. On choisit deux indices : `low` (début) et `high` (fin).
2. On calcule l'indice du milieu : `mid = (low + high) / 2`.
3. On compare `a[mid]` avec la valeur cherchée  $x$  :
  - si `a[mid] == x`, on a trouvé;
  - si `a[mid] < x`, alors  $x$  ne peut se trouver que dans la moitié droite : on met `low = mid + 1`;
  - si `a[mid] > x`, alors  $x$  ne peut se trouver que dans la moitié gauche : on met `high = mid - 1`.
4. On recommence tant que `low <= high`.

Ainsi, à chaque étape, la taille du segment à explorer est divisée par 2. C'est ce qui rend l'algorithme extrêmement rapide.

**Question 0 :** Pourquoi faut-il que la liste ou le tableau soit trié pour appliquer la recherche dichotomique ?

### 2.1 Version C : tableau trié

**Exercice B1 :** Compléter la fonction suivante :

Listing 3 – Recherche dichotomique en C

```

1 int binary_search(int *a, int n, int x) {
2     /* A compléter :
3         - utiliser deux indices low et high
4         - tant que low <= high :
5             - calculer mid
6             - si a[mid] == x : retourner mid
7             - si a[mid] < x : chercher dans la moitié droite
8             - sinon : chercher dans la moitié gauche
9         - si non trouve : retourner -1
10    */
11 }
```

**Question 1 :**

1. Que vaut `mid` si `low = 0` et `high = 9` ?
2. Pourquoi ne pas utiliser `(low + high) / 2` dans des contextes de grande taille ? (Indice : dépassement d'entier.)

## 2.2 Version OCaml : liste triée

**Exercice 2 :** Écrire une recherche dichotomique en OCaml sur une **liste triée**.

### Attention

Comme les listes OCaml ne donnent pas accès direct à l'indexation, vous devez :

- soit convertir la liste en array d'abord ;
- soit écrire une version purement récursive qui maintient le type `list`.

Listing 4 – Idée générale en OCaml

```
1 let rec binary_search_list l x =
2   (* A compléter : plusieurs stratégies possibles *)
3   *)
4   ;;
```

**Question 2 :**

1. Quels sont les avantages/inconvénients d'utiliser un tableau pour faire la recherche ?
2. La recherche dichotomique sur listes pures OCaml est-elle efficace ? Pourquoi ?

## 2.3 Tests

Tester votre fonction sur :

- tableau/liste vide ;
- valeur présente au début, au milieu, à la fin ;
- valeur absente.

## 3 Synthèse

**Questions finales :**

1. Quels sont les points communs entre la version C et OCaml des deux exercices ?
2. Quels aspects facilitent l'implémentation en OCaml ?  
Quels aspects sont plus naturels en C ?