I Algorithme naïf

I.1 Description

1. Pour i de 1 à $(n-1)$
   (a) Pour k de $i+1$ à n
      i. Si $\sigma(i) > \sigma(k)$ on rajoute une inversion.

I.2 Complexité

$O(n^2)$ car $\sum_{k=1}^{n} k$ comparaisons.

I.3 Implémentation

```
let count_inversions sigma =
        let rec inloop sigma_i sigmas = match sigmas with
                | []   -> 0
                | h::t -> if h < sigma_i
                          then 1 + (inloop sigma_i t)
                          else (inloop sigma_i t)
                in
        let rec outloop sigmas = match sigmas with
                | []   -> 0
                | [a]  -> 0
                | h::t -> (outloop t) + inloop h t
                in
        outloop sigma;;
```

Le test :

```
assert (count_inversions [2;4;3;1;5] = 4)
```

II Algorithme de fusion

II.1 Description

Un example

$$\begin{cases} 1 \mapsto 5 \\ 2 \mapsto 2 \\ 3 \mapsto 1 \\ 4 \mapsto 3 \\ 5 \mapsto 4 \end{cases} = \texttt{[5; 2; 1; 3; 4]} \quad 5 \text{ inversions}$$

On coupe : `[5; 2; 1]` et `[3; 4]`.
Trions `[5; 2; 1]`.

- On échange `1`
- On fusionne `[5]` et `[1; 2]`
- `[1; 2; 5]`

Trions `[3; 4]`

Un autre exemple

`[5; 3; 1; 2; 4]`

En gros On incrémente un compteur quand il faut inverser tout en triant la liste par fusion.

II.2 Compléxité

Tri par fusion $\Theta(n \log_2 n)$
Compte de permutation

II.3 Implémentation

```
let split l =
        let rec aux l n acc = match l with
        | [] -> [], []
        | h::t when n=0 -> (List.rev acc, h::t)
        | h::t -> aux t (n-1) (h::acc)
        in aux l (List.length l / 2) [];;

let merge l1 l2 =
let rec aux l1 l2 _acc = match l1, l2 with
| [], _ -> l2
| _, [] -> l1
| h1::t1, h2::t2 -> if h1 < h2
        then h1::(aux t1 l2 _acc)
        else h2::(aux l1 t2 (_acc + List.length l1))
in aux l1 l2 0;;
```