I Algorithme naïf

I.1 Description

Soit $d = \sqrt{(\texttt{fst}l_1 - \texttt{fst}l_0)^2 + (\texttt{snd}l_1 - \texttt{snd}l_0)^2}$.
Pour $i \in [\![0, n[\![$: Pour $j \in [\![0, n[\![$: Si $\sqrt{(\texttt{fst}l_i - \texttt{fst}l_j)^2 + (\texttt{snd}l_i - \texttt{snd}l_j)^2} < d$, alors
soit $d = \sqrt{(\texttt{fst}l_i - \texttt{fst}l_j)^2 + (\texttt{snd}l_i - \texttt{snd}l_j)^2}$.

I.2 Complexité

Ya deux `for` imbriqués parcourant n éléments, c'est $O(n^2)$ quoi c'est bon

I.3 Implémentation

```
let distance p1 p2 = sqrt ( ((fst p1) -. (fst p2))**2 +. ((snd p1) -. (snd p2))**2 )

let naive_min_distance points =
    let min_distance = ref (distance l.(0) l.(1)) in
    for i = 0 to (Array.length l) do
    for j = 0 to (Array.length l) do
        let d = distance l.(i) l.(j) in
        if d < !min_distance
        then min_distance := d
    !min_distance;;

let min_distance_nuage l =
    let rec min_1couple l min (a,b) = match l with
        |[] -> min
        |(c,d)::t -> if (abs_float(a-.c)**2. +. abs_float(b-.d)**2.) < min then
                            let min = abs_float(a-.c)**2. +. abs_float(b-.d)**2. in
                            min_1couple t min (a,b)
                        else
                            min_1couple t min (a,b)
        in

    let rec aux_nuage l min = match l with
        |[] -> min
        |(a,b)::t -> aux_nuage t (min_1couple t min (a,b))
        in

    let mini = abs_float(fst (List.hd l)-. fst (List.hd (List.tl l)))**2.
                +. abs_float(snd (List.hd l) -. snd (List.hd (List.tl l)))**2. in

    aux_nuage l mini ;;
```

C'est tellement plus clair.

II Cas de dimension 1

On ne regarde ques les distances deux à deux, avec un algorithme à la diviser pour régner.

— Cas de base : liste à deux éléments : c'est la distance elle-même :

— Récursion : Si la demi-liste à gauche a une distance minimum plus petite, on retourne celle-là, sinon on retourne l'autre.

III Diviser pour régner trop naïf

$$c_2 = 1$$
$$c_n = c_{\lfloor \frac{n}{2} \rfloor} + c_{\lceil \frac{n}{2} \rceil}$$

On a un $O(n)$, sauf que le tri est en $O(n \lg n)$.
Sauf qu'on a finalement $O(n^2 \lg n)$, si il y a trop de points à trier.

IV Stratégie optimale

IV.1 On a au plus 8 points

On ne prend plus une bande verticale mais un carré car on a fait le trie sur les x et sur les y. On regarde les points dans ce carré qui peuvent plus proche que delta. delta = min(d1,d2) bande = carrée de longueur 2*delta Si d1 = d2 =delta , on pourra avoir un point à gauche et un point à droite (trie des abcisses) avec un point en dessous et un point au dessus (trie des ordonnées). Tous les points sont possibles sauf le centre(car la distance 0 n'est pas possible), soit 9-1 points soit 8 points qui correspondent à "je suis situé à distance delta de mon voisin le plus proche"

IV.2 Algorithme

Bon...

IV.3 Complexité

$$c_n = 2c_{\lfloor \frac{n}{2} \rfloor} + O(n)$$
$$\Longrightarrow \Theta(n \lg n)$$

IV.4 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;;
```