Diviser pour régner

1 Paire la plus proche

On considère un ensemble de n points du plan et l'on souhaite déterminer la distance minimale entre deux de ces points. On représente un point par le type suivant :

```
type point = {x : float; y : float}
```

Un ensemble (ou *nuage*) de points sera représenté par une liste ou un tableau de points suivant les cas. On pourra éventuellement utiliser les fonctions `Array.of_list` et `Array.to_list` pour passer d'une représentation à l'autre quand l'une est plus appropriée.

Remarque

⇨ On pourra utiliser la valeur spéciale `infinity` (de type `float`) pour simplifier certaines fonctions.

1. Écrire une fonction `distance` calculant la distance euclidienne entre deux points.

   ```
   distance : point -> point -> float
   ```

2. Écrire une fonction `dmin_naif` résolvant le problème de la manière la plus simple possible.

   ```
   dmin_naif : point array -> float
   ```

On se propose de chercher une solution « diviser pour régner » à ce problème. Pour ce faire, on va tenter d'exploiter l'idée suivante :

— séparer le nuage de point en deux, suivant la valeur de x (la moitié de points la plus à gauche d'une part, la moitié la plus à droite d'autre part) ;
— calculer d_g (respectivement d_d), la distance minimale entre deux points situés tous à gauche (respectivement deux points situés à droite) ;
— en déduire d, la distance minimale entre deux points du nuage.

1. Que faut-il renvoyer dans les cas où le nuage contient zéro, un ou deux points ? Attention, ce sont bien sûr des cas différents !
2. Écrire une fonction `separe_moitie` qui prend en argument une liste u de longueur n et renvoie un couple v, w de listes telles que `u = v @ w`, $|v| = \lfloor n/2 \rfloor$ et $|w| = n - |v|$.

   ```
   separe_moitie : 'a list -> ('a list * 'a list)
   ```

3. Écrire une fonction `compare_x` telle que l'appel `compare_x a b` renvoie :
   — `-1` si `a.x < b.x` ;
   — `0` si `a.x = b.x` ;
   — `1` sinon.

   ```
   compare_x : point -> point -> int
   ```

4. Écrire une fonction `tri_par_x` qui trie une liste de points par coordonnée `x` croissante. On pourra utiliser la fonction `List.sort` de la bibliothèque standard, en cherchant sa documentation.

   ```
   tri_par_x : point list -> point list
   ```

5. Écrire une fonction `dmin_gauche_droite` qui prend en entrée deux listes de points et renvoie la distance minimale entre un point de la première liste et un point de la deuxième liste.

   ```
   dmin_gauche_droite : point list -> point list -> float
   ```

6. En déduire une fonction `dmin_dc_naif` qui calcule la distance minimale entre deux points d'un nuage à l'aide d'une stratégie « diviser pour régner ».

```
dmin_dc_naif : point list -> float
```

7. Donner la relation de récurrence vérifiée par la complexité de `dmin_dc_naif`. Que peut-on en conclure ?

Pour obtenir une complexité satisfaisante, nous allons améliorer l'étape de fusion.

On suppose ici que l'on a séparé notre ensemble de points en deux suivant l'axe des `x` et l'on note x_{med} une abscisse médiane (par exemple l'abscisse du point le plus à gauche de la partie de droite). On note également d_g (respectivement d_d) les distances minimales entre deux points de la partie de gauche (respectivement droite), que l'on suppose calculées. On note $d = \min(d_g, d_d)$, et l'on souhaite calculer d_{min} (distance minimale entre deux points du nuage).

1. Justifier que l'on peut se limiter aux points dont l'abscisse vérifie $x_{med} - d \leqslant x \leqslant x_{med} + d$.
2. On suppose désormais que l'on dispose des points de cette bande, triés par *ordonnée* croissante. Justifier que l'on peut se contenter de calculer la distance minimale entre chaque point de cette liste et les 7 points suivants.

1. Écrire une fonction `dmin_dc` implémentant la stratégie exposée ci-dessus.

```
dmin_dc : point list -> float
```

2. Montrer que la complexité temporelle de cette fonction vérifie $T(n) \leqslant 2T(n/2) + An\log n$.
3. En déduire qu'on a $T(n) = \mathrm{O}\left(n(\log n)^2\right)$.
4. Comment pourrait-on réduire la complexité à $\mathrm{O}(n\log n)$?
5. **Si vous avez fini le reste du sujet.** Écrire une nouvelle version de `dmin_dc` ayant cette complexité.

2 Nombre d'inversions

On définit le nombre d'inversions $\sigma(x)$ d'une séquence $x = x_0, \dots x_{n-1}$ d'entiers comme le nombre de couples (i, j) tels que $1 \leqslant i < j \leqslant n$ et $x_i > x_j$.
On représentera ici les séquences par des listes.

1. Au maximum, combien vaut $\sigma(x)$?
2. Quelle est la complexité de l'algorithme naïf pour calculer $\sigma(x)$?
3. Écrire une fonction `nb_inv_naif`.

```
nb_inv_naif : 'a list -> int
```

4. En utilisant une stratégie « diviser pour régner », trouver un algorithme de complexité $\mathrm{O}(n\log n)$ permettant de résoudre ce problème. *On pourra s'inspirer du tri fusion.*
5. Implémenter cet algorithme en OCaml.

```
nb_inv : 'a list -> int
```