TP 5 : Calcul de la paire de points les plus proches dans un nuage de points

On s'intéresse dans ce TP au calcul de la paire de points la plus proche dans un nuage de points du plan. Après un premier algorithme naïf, on rappelle la stratégie « diviser pour régner » permettant d'écrire un algorithme efficace. Remarque : cet algorithme est explicitement cité dans le programme comme un des exemples d'algorithmes « diviser pour régner ».

Annexe ! Télécharger l'annexe sur le site web, et commencez à coder à la suite ! Les fonctions données sont les suivantes :

— `rd_couple ()` et `rd_nuage n` renvoient respectivement un couple aléatoire de flottants dans $[0, 1000[^2$ et un tableau de n tels couples. Vous utiliserez ces fonctions une fois le programme terminé pour faire des tests.

— Les fonctions `round x`, `trace_nuage tab` et `coloration p q` sont également utiles pour les tests. `trace_nuage` permet de tracer le nuage de points à l'écran (ne pas fermer la fenêtre graphique qui s'ouvre automatiquement), et `coloration p q` permet de tracer deux points (couples de flottants) en rouge et bleu, un peu plus gros.

— Sont également définis quelques tableaux vous permettant de vérifier vos fonctions comme dans l'énoncé.

1 Mise en place et résolution du problème par un algorithme naïf

Question 1. Écrire une fonction `distance : float * float -> float * float -> float`, prenant en entrée deux couples de flottants et calculant la distance qui les sépare. (Rappel : `sqrt` fournit la racine carrée, les opérateurs arithmétiques sur les flottants comportent des « points », comme `+.` par exemple, et enfin les constantes flottantes sont des nombres à virgule).

```
# distance tab1.(0) tab1.(1) ;;
- : float = 614.463544765679217
```

Question 2. En déduire `plus_proches_naif : (float * float) array -> (float * float) * (float * float)`, une fonction permettant de rechercher le couple de points les plus proches parmi les points d'un nuage.

```
# plus_proche_naif tab1 ;;
- : (float * float) * (float * float) =
((137.952822280455479, 579.517066764289893),
 (136.342566575285474, 698.315452710874752))
```

La complexité de cet algorithme est en $O(n^2)$ (normalement [1]), la suite est dévolue à l'écriture de l'algorithme efficace en $O(n \log n)$.

2 Intermède : le tri fusion pour les tableaux

On écrit ici le tri fusion, pour les tableaux. Le principe est le même que celui pour les listes :

— si le tableau à trier possède au plus un élément, on le renvoie à l'identique ;

— sinon, on le coupe en deux parties de même taille à un élément près, on trie séparément les deux parties, et on fusionne le résultat en un tableau trié.

Question 3. Écrire une fonction `fission : 'a array -> 'a array * 'a array` prenant en entrée un tableau et renvoyant un couple de tableaux (on coupe en deux). Avec n la taille du tableau, les deux éléments du couple seront de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. Rappel : en Caml, `/` est la division entière, et `Array.sub t i k` renvoie le sous-tableau de `t` de taille k démarrant à l'indice i.

```
# fission tab_int ;;
- : int array * int array = ([|2; 50; 13; 8; 2; 12|], [|48; 16; 10; 46; 81; 21; 30|])
```

La fonction de fusion est un peu plus compliquée. On pourra supposer que l'on ne fusionne que des tableaux non vides (ce sera le cas en pratique). De plus, dans l'optique de trier notre nuage de points suivant les abscisses ou les ordonnées, notre fonction (et la fonction `tri_fusion`) prendra en paramètre une fonction de tri, de la forme `inf : 'a -> 'a -> bool`. On a `inf x y` lorsque $x \leq y$ pour l'ordre choisi.

1. Et pas $O(n!)$, $O(2^n)$ ou tout autre proposition très surévaluée !

Question 4. Écrire une fonction `fusion : 'a array -> 'a array -> ('a -> 'a -> bool) -> 'a array` prenant en entrée deux **tableaux** non vides, de tailles n_1 et n_2, supposés triés dans l'ordre croissant (pour la fonction de tri), et retournant un nouveau tableau, de taille $n_1 + n_2$, contenant les éléments de `t1` et `t2` triés dans l'ordre croissant. Attention à ne pas faire de dépassement d'indices, et à faire les choses correctement. Dans l'idée, on maintiendra deux références vers des entiers (indices dans `t1` et `t2`) pour comparer les éléments et récupérer le plus petit. Traiter par exemple le cas où l'un des deux tableaux a entièrement été recopié à part.

```
# fusion [|0;2;3;8|] [|1;5;7;9;11|] (fun x y -> x<=y) ;;
- : int array = [|0; 1; 2; 3; 5; 7; 8; 9; 11|]
# fusion [|10;4|] [|5;3;1|] (fun x y -> x>=y) ;;
- : int array = [|10; 5; 4; 3; 1|]
```

Question 5. En déduire une fonction `tri_fusion : 'a array -> ('a -> 'a -> bool) -> 'a array` permettant de trier un tableau dans l'ordre croissant pour la fonction de tri (en renvoyant un nouveau tableau).

```
# tri_fusion tab_int (fun x y -> x<=y) ;;
- : int array = [|2; 2; 8; 10; 12; 13; 16; 21; 30; 46; 48; 50; 81|]
# tri_fusion tab_int (fun x y -> x>=y) ;;
- : int array = [|81; 50; 48; 46; 30; 21; 16; 13; 12; 10; 8; 2; 2|]
```

Dans la suite, on aura à trier le nuage de points pour les deux ordres lexicographiques $\preceq_x$ et $\preceq_y$ suivants :

— $(a, b) \preceq_x (c, d)$ si et seulement si $a < c$ ou $(a = c$ et $b \leq d)$;

— $(a, b) \preceq_y (c, d)$ si et seulement si $b < d$ ou $(b = d$ et $a \leq c$;

Question 6. Définir deux fonctions de tri `inf_x` et `inf_y` associés à ces ordres. Vérifier que `tri_fusion tab1 inf_x` (resp. `tri_fusion tab1 inf_y`) est un tableau croissant sur la première (resp. deuxième) coordonnée. (Ce sont les tableaux `tab1x` et `tab1y` de l'annexe)

3 Implémentation de l'algorithme efficace

L'algorithme efficace consiste à faire un précalcul : on duplique le nuage de points `t` en deux tableaux `tx` et `ty`, triés pour les ordres précédents. L'algorithme récursif prend en entrée deux tels tableaux, contenant les mêmes points mais triés différemment.

— si les deux tableaux possèdent moins de 6 éléments (le 6 est assez arbitraire), on résout le problème à l'aide de l'algorithme naïf établi précédemment, appelé au choix sur `tx` ou `ty` ;

— sinon, on sépare les points en deux parties égales (à un élément près...) situées de part et d'autre d'une droite verticale d'équation $x = x_0$: les éléments situés à gauche sont placés dans deux tableaux `tgx` et `tgy`, triés respectivement pour les relations $\preceq_x$ et $\preceq_y$. Il en va de même pour les éléments situés à droite qu'on place dans deux tableaux `tdx` et `tdy`.

— on applique récursivement l'algorithme pour déterminer la distance minimale d_g (resp. d_d) dans la partie gauche (resp. droite), et les couples de points associés ;

— on détermine un couple de points situés de part et d'autre de la droite $x = x_0$, qui minimise la distance entre les points situés de part et d'autre de la droite $x = x_0$, si celle-ci est inférieure aux deux distances d_g et d_d. Il n'y a plus qu'à renvoyer le couple de points qui minimise la distance. Faire ceci efficacement est un peu subtil.

Dans la suite, on découpe le travail à l'aide de quelques fonctions.

Remarque. On ne travaille qu'avec des couples de flottants aléatoires, en pratique les nuages considérés sont constitués de points distincts (et même de points à coordonnées toutes distinctes).

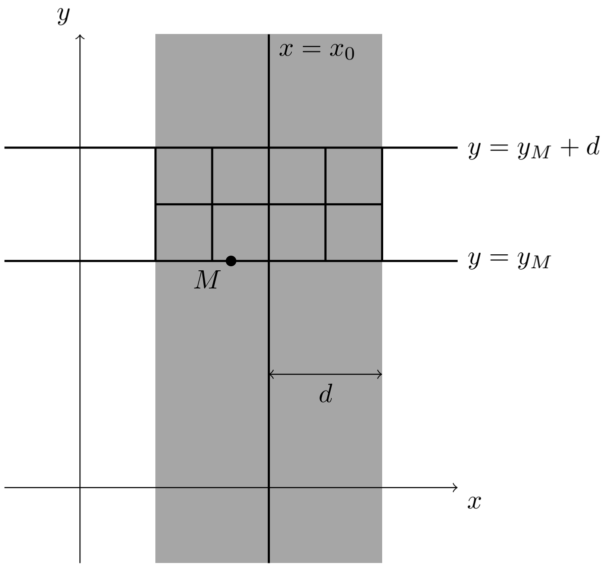
Question 7. *Découpage de `tx`.* Écrire une fonction `decoupage tx` prenant en entrée le tableau trié pour $\preceq_x$ et le scindant en deux (avec n la taille de `tx` les deux morceaux ont taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$). On utilisera `Array.sub`.

```
# decoupage tab1x ;;
- : (float * float) array * (float * float) array =
([|(136.342566575285474, 698.315452710874752);
   (137.952822280455479, 579.517066764289893);
   (186.89530977266773, 283.014495971758549);
   (345.004922664389085, 939.068336631267471)|],
 [|(466.113061566003125, 188.103956342865786);
```

```
   (745.24288848465, 673.134497490137164);
   (891.680623462061476, 287.663161181994212);
   (944.098271788395095, 410.935431876780797)|])
```

Question 8. *Répartition de* `ty`. En notant `txg` et `txd` les deux tableaux précédents, on considère c, le premier couple du tableau `txd`. Le but est maintenant d'obtenir les tableaux `tyg` et `tyd`, contenant les mêmes éléments que `txg` et `tyd`, mais triés pour l'ordre $\preceq_y$. Utiliser à nouveau le tri fusion ici est mauvais en terme de complexité : il suffit d'exploiter que `ty` est déja trié, pour répartir ses éléments en deux tableaux de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$: les éléments de `tyg` sont les éléments a vérifiant $a \prec_x c$, ceux de `tyd` vérifiant $a \succeq_x c$. Écrire une fonction `reparti ty c` prenant en entrée un tableau trié pour $\preceq_y$, et un couple c du tableau. En notant n la taille du tableau, on suppose que c est le couple d'indice $m = \lfloor n/2 \rfloor$ du tableau `tx` associé. Votre fonction devra avoir une complexité $O(n)$ et renverra un couple de deux tableaux `tgy` et `tdy`,

```
# reparti tab1y tab1x.(4) ;;
- : (float * float) array * (float * float) array =
([|(186.89530977266773, 283.014495971758549);
   (137.952822280455479, 579.517066764289893);
   (136.342566575285474, 698.315452710874752);
   (345.004922664389085, 939.068336631267471)|],
 [|(466.113061566003125, 188.103956342865786);
   (891.680623462061476, 287.663161181994212);
   (944.098271788395095, 410.935431876780797);
   (745.24288848465, 673.134497490137164)|])
```



À présent, on peut par un appel récursif calculer les distances d_g et d_d entre les points les plus proches dans les parties gauche et droite du nuage. En notant x_0 la moyenne des abscisses entre le dernier point de `tgx` et le premier de `tdx`, et $d = \min(d_g, d_d)$, on a vu en cours qu'on pouvait se restreindre à la bande verticale d'équation $x \in [x_0 - d, x_0 + d]$ pour chercher deux points du nuage à une distance strictement inférieure à d. (voir figure ci-dessus).

Question 9. Écrire `extrait_bande : (float * 'a) array -> float -> float -> (float * 'a) array` telle que `extrait_bande ty x0 d` extrait du tableau `ty` les points de la bande d'abscisse $x \in [x_0 - d, x_0 + d]$. Ils seront ainsi triés par ordonnée croissante. On pourra utiliser une référence vers une liste, ainsi que les fonctions `List.rev` (renversant une liste) et `Array.of_list` (convertissant une liste en tableau)

```
# extrait_bande tab1y 400. 100. ;;
- : (float * float) array =
[|(466.113061566003125, 188.103956342865786);
  (345.004922664389085, 939.068336631267471)|]
```

On a vu dans le cours qu'en notant `tb` le tableau renvoyé par la fonction précédente, il suffit pour un indice i donné d'examiner seulement les points d'indice dans $[\![i+1, i+7]\!]$ (voir figure : dans chaque petit carré de côté $d/2$ ne se trouve qu'au plus un point du nuage).

Question 10. Écrire `parcours_bande : (float * float) array -> float -> float * int * int` prenant en paramètre le tableau des points de la bande, et la distance $d = \min(d_g, d_d)$. La fonction renvoie un triplet (d_2, i, j) :

— s'il y a dans la bande deux points à distance $< d$, alors d_2 sera le minimum des distances entre deux points de la bande, et i et j les indices correspondant dans le tableau `tb`.

— sinon, on aura $d_2 \geq d$ et i et j sont arbitraires.

On prendra garde à ne pas faire de dépassement d'indice. En particulier, il se peut que la bande ait 0 ou 1 élément, il ne faut donc pas tenter d'accéder à un élément qui n'existe pas.

```
# parcours_bande tb1 1000. ;;
- : float * int * int = (760.667260877024887, 0, 1)
# parcours_bande [||] 100. ;;
- : float * int * int = (101., 0, 1)
```

On impose une complexité $O(n_b)$ avec n_b le nombre de points de la bande.

On a maintenant tout ce qu'il faut pour écrire l'algorithme final.

Question 11. Écrire une fonction `plus_proche_efficace t` prenant en entrée un nuage de points et renvoyant le couple de points du nuage minimisant la distance. On pourra par exemple utiliser une fonction auxiliaire interne, travaillant sur `tx` et `ty`. S'il vous reste peu de temps, la fonction est déja écrite dans la deuxième annexe (cf. site).

Question 12. Faites des tests avec les fonctions fournies dans la deuxième annexe (chez moi, pour 10000 points, l'algorithme naïf calcul la plus petite distance en un peu plus de vingt secondes, et l'algorithme efficace en un dixième de seconde).