TD XXXIII

DEUX EXEMPLES DE 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.

Exercice XXXIII.1 p. 4

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

```
distance2 : 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.

Exercice XXXIII.2 p. 4

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 (ou en se référant au TP du début d'année sur le tri fusion).

```
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.

Exercice XXXIII.3 p. 5

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.

Exercice XXXIII.4 p. 6

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 $T(n) = O\left(n(\log n)^2\right)$.
4. Comment pourrait-on réduire la complexité à $O(n\log n)$?
5. **Si vous avez fini le reste du sujet.** Écrire une nouvelle version de `dmin_dc` ayant cette complexité.

2 Nombre d'inversions

Exercice XXXIII.5 – Nombre d'inversions

p. 8

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 $O(n \log n)$ permettant de résoudre ce problème.
   Remarque
   On pourra s'inspirer du tri fusion.
5. Implémenter cet algorithme en OCaml.

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

Solutions

Correction de l'exercice XXXIII.1 page I

1.

```
let distance a b = (a.x -. b.x) ** 2. +. (a.y -. b.y) ** 2.
```

2. On initialise à `infinity`, c'est le plus simple (et cela reste valable si $n < 2$).

```
let dmin_naif points =
  let n = Array.length points in
  let dmin = ref infinity in
  for i = 0 to n - 1 do
    for j = i + 1 to n - 1 do
      dmin := min !dmin (distance points.(i) points.(j))
    done
  done;
  !dmin
```

Correction de l'exercice XXXIII.2 page I

1. Pour $n = 0$ ou $n = 1$, il faut renvoyer `infinity` (ou lever une exception, mais cela rendrait l'écriture de la fonction nettement plus compliquée). Pour $n = 2$, on renvoie la distance entre les deux points (c'est le « vrai » cas de base).
2. On a écrit ici une version récursive terminale (qui nécessite un appel à `List.rev` à la fin) pour ne pas être limité par la pile d'appels sur de très grandes listes, mais ce n'est pas indispensable. On pourra regarder le dernier exercice de ce sujet pour une autre version.

```
let separe u =
  let rec separe_aux v k pris =
    match v with
    | [] -> List.rev pris, []
    | x :: xs ->
      if k = 0 then List.rev pris, v
      else separe_aux xs (k - 1) (x :: pris) in
  separe_aux u (List.length u / 2) []
```

Remarque

On peut en fait remplacer `List.rev pris` par `[]` dans le premier cas du `match`.

3.

```
let compare_x a b =
  if a.x -. b.x < 0. then -1
  else if a.x = b.x then 0
  else 1
```

4.

```
let trie_par_x = List.sort compare_x
```

5. On pourrait convertir les deux listes en tableaux, mais ce n'est pas nécessaire :

```
let rec dmin_gauche_droite u v =
  match u with
  | [] -> infinity
  | hd :: tl ->
    (* on calcule la distance minimale entre p et un point de v *)
    let rec aux = function
      | [] -> infinity
      | hd' :: tl' -> min (distance hd hd') (aux tl') in
    min (aux v) (dmin_gauche_droite tl v)
```

6. On traduit exactement l'algorithme donné, en évitant cependant de re-trier à chaque étape : puisque `separe` respecte l'ordre, l'argument `points` de la fonction `aux` restera toujours trié.

```
let dmin_dc_naif points =
  let rec aux points =
    match points with
    | [] | [_] -> infinity
    | [a; b] -> distance a b
    | _ ->
      let gauche, droite = separe points in
      let dg = aux gauche in
      let dd = aux droite in
      let d_gd = dmin_gauche_droite gauche droite in
      min d_gd (min dg dd) in
  let par_x = trie_par_x points in
  aux par_x
```

7. Il y a $n/2$ points (à un près) dans `gauche` et dans `droite`, donc l'appel `dmin_gauche_droite` se fera en temps proportionnel à $(n/2)^2$ et donc à n^2. La séparation peut être négligée puisqu'elle est en temps linéaire, et l'on obtient donc $T(n) = 2T(n/2) + An^2$. Il est alors immédiat que la complexité est au moins quadratique : on n'a donc rien gagné par rapport à la solution naïve.
On obtient en fait une complexité en $\Theta(n^2)$ (en appliquant la technique usuelle), et l'on n'a donc « rien perdu » non plus, sauf que l'algorithme est plus compliqué. . .

Correction de l'exercice XXXIII.3 page 2

1. Les seules distances que l'on souhaite considérer sont celles entre un point A situé à gauche et un point B situé à droite. Si par exemple A n'est pas dans la bande suggérée, on a $x_A < x_{med} - d$ et comme $x_B \geqslant x_{med}$ on en déduit $x_B - x_A > d$ et donc $distance(A, B) > d$.

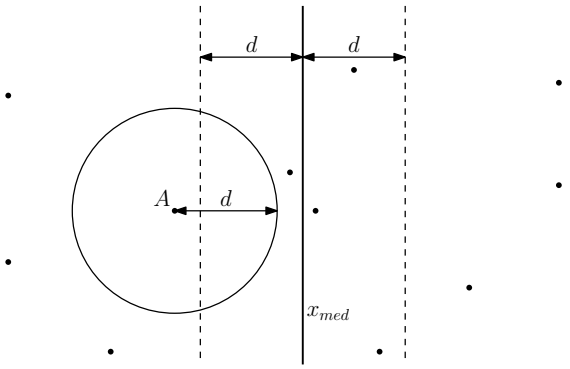

Figure XXXIII.3 – Un point de la moitié droite ne peut être à une distance inférieure à d de A.

2. Un bon dessin vaut mieux qu'un long discours :

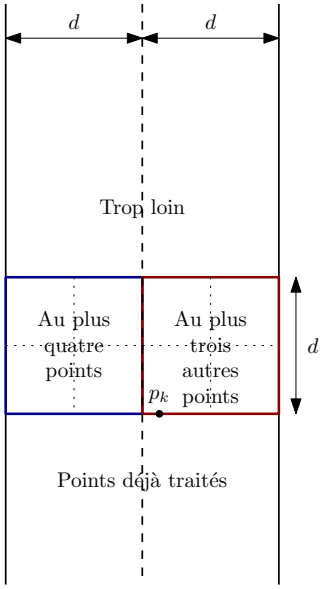

Figure XXXIII.4 – ei

Si l'on place 5 points dans le carré de gauche, on aura nécessairement deux points situés à une distance inférieure à d l'un de l'autre, ce qui est impossible. En effet, en divisant ce carré en quatre carrés de côté d/2, le principe des tiroirs qu'il y aurait deux points dans le même carré. Or la distance maximale entre deux points d'un tel carré est $\frac{d\sqrt{2}}{2} < d$ (longueur de la diagonale). Il y a donc au plus 4 points dans le carré de gauche, et 4 (dont le point actif) dans celui de droite : il suffit donc de s'intéresser aux 7 prochains points.

Correction de l'exercice XXXIII.4 page 2

1. Pour le calcul de la distance minimale avec l'un des sept prochains points, il est plus agréable d'utiliser un tableau :

```
let dmin_7 par_y =
  let dmin = ref infinity in
  let n = Array.length par_y in
  for i = 0 to n - 1 do
    for j = i + 1 to min (n - 1) (i + 7) do
      dmin := min !dmin (distance par_y.(i) par_y.(j))
    done
  done;
  !dmin
```

Ensuite, on traduit l'énoncé, avec deux remarques :

- l'appel à `List.hd` est légitime, puisqu'on est dans un cas où `liste_points` est de longueur au moins 3 (et donc `droite` de longueur au moins 1) ;
- `liste_points` reste trié par x croissants tout au long des appels (on la trie par y croissants quand on en a besoin).

```
let dmin_dc points =
  let rec aux liste_points =
    match liste_points with
    | [] | [_] -> infinity
    | [a; b] -> distance a b
    | _ ->
      let gauche, droite = separe liste_points in
      let x_median = (List.hd droite).x in
      let d_gauche = aux gauche in
      let d_droite = aux droite in
      let d = min d_gauche d_droite in
      let dans_bande a =
        x_median -. d <= a.x && a.x <= x_median +. d in
      let bande = List.filter dans_bande liste_points in
      let tab_trie_y = Array.of_list (trie_par_y bande) in
      min d (dmin_7 tab_trie_y) in
  aux (trie_par_x points)
```

2. La séparation se fait en temps $O(n)$. Pour la fusion :
   - le calcul de `bande` est en $O(n)$;
   - le calcul de `tab_trie_y` est en $O(p \log p)$ où $p = |\texttt{bande}|$;
   - le calcul de `dmin_7 tab_trie_y` est en $O(p)$.

   On a $p \leqslant n$, et c'est la meilleure majoration que l'on puisse obtenir ; on en déduit $T(n) \leqslant 2T(n/2) + An \log n$ avec A une constante.

3. On pose maintenant $T(n)$ le nombre maximal d'opérations élémentaires pour une instance de taille inférieure ou égale à n (ce qui assure la croissance). On a alors successivement :

$$\begin{aligned}
T(2^i) &\leqslant 2T(2^{i-1}) + Ai2^i \\
\frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} &\leqslant Ai \\
\frac{T(2^k)}{2^k} - B &\leqslant A\sum_{i=1}^{k} i && \text{B constante} \\
T(2^k) &\leqslant Ck^2 2^k && \text{C constante}
\end{aligned}$$

   Si $n = 2^k$, on a donc bien $T(n) = O(n \log^2 n)$. Dans le cas général, on a $n \leqslant 2^{\lceil \log_2 n \rceil}$, d'où $T(n) \leqslant C \cdot \left(\lceil \log_2 n \rceil\right)^2 2^{\lceil \log_2 \rceil} \leqslant C(1 + \log_2 n) \cdot 2n$, donc également $T(n) = O(n \log^2 n)$.

4. Pour passer à du $O(n \log n)$, il faut éliminer l'étape de tri, ou plutôt ne trier qu'une seule fois au début. C'est possible : il commencer par calculer la liste des points triés suivant x et la liste triée suivant y, puis filtrer ces listes à chaque appel récursif pour ne garder que ceux qui nous intéressent, sans modifier l'ordre (c'est ce que l'on fait déjà pour la liste triée suivant x).

   Remarque

   Si l'on savait trier une liste de flottants en temps linéaire (ce qui est possible en adaptant le tri radix), ce serait également une option valable.

5. Laissé en exercice... Cela ne pose aucun problème si l'on suppose que deux points distincts du nuage n'ont jamais la même abscisse ni la même ordonnée, il faut faire un peu plus attention si l'on ne fait plus cette hypothèse.

Correction de l'exercice XXXIII.5 page 3

1. $\sigma(x)$ est clairement majoré par le nombre de couples (i,j) vérifiant $0 \leqslant i < j < n$, c'est-à-dire $\frac{n(n-1)}{2}$. Inversement, tous les couples considérés sont en inversion si la séquence est strictement décroissante : ce majorant est donc un maximum.
2. Naïvement, on parcourt tous les couples (i,j) tels que $0 \leqslant i < j < n$ en testant à chaque fois si les éléments sont inversés. Comme dit plus haut, il y a de l'ordre n^2 tels couples, et la complexité serait donc en $\Theta(n^2)$.
3. C'est un peu plus facile à écrire avec un tableau qu'avec une liste, mais on va éviter de convertir : c'est un bon entraînement.

```
let rec nb_inv_naif u =
  match u with
  | [] -> 0
  | x :: xs ->
    (* on compte le nombre d'éléments de xs qui sont < x *)
    let rec aux = function
      | [] -> 0
      | y :: ys -> if x > y then 1 + aux ys else aux ys in
    aux xs + nb_inv_naif xs
```

4. On va simultanément trier la séquence par ordre croissant (à l'aide d'un tri fusion) et compter le nombre d'inversions. Le point crucial est que, si `s` et `t` sont deux séquences triées et que l'on considère `u = s @ t`, alors il est possible de déterminer le nombre d'inversions dans `u` en temps linéaire. En effet :
   - il n'y a pas d'inversion entre éléments de `s` ou entre éléments de `t`, et il suffit donc de compter le nombre d'inversions entre un élément de `s` et un élément de `t` (que l'on notera $inv(s,t)$) ;
   - si `s` ou `t` est vide, c'est trivial ;
   - sinon, on a `s = x :: xs` et `t = y :: ys` et l'on distingue deux cas :
     — si $x \leqslant y$, alors x n'est en inversion avec *aucun* élément de `t` (puisqu'elle est croissante), donc $inv(s,t) = inv(xs,t)$;
     — sinon, y est en inversion avec *tous* les éléments de s (puisqu'elle est croissante), et donc $inv(s,t) = |s| + inv(s,ys)$.

   Ensuite, on remarque que si `u = s @ t` (sans hypothèse sur `s` et `t`), on a en notant `s'`, `t'` les versions triées de `s` et `t` :

   $$\sigma(u) = \sigma(s) + \sigma(t) + inv(s',t')$$

   On va donc écrire une fonction (disons `aux`) qui prend en entrée une liste u et renvoie le couple p, u' où $p = \sigma(u)$ et u' est la version triée de u. On procède comme suit (si l'on n'est pas dans un cas de base) :
   - on coupe la liste en deux (contrairement au tri fusion classique, il est indispensable de couper en première moitié, deuxième moitié) ;
   - on appelle récursivement `aux` sur chacune des moitiés, on récupère $\sigma(s), s'$ et $\sigma(t), t'$;
   - on fusionne s' et t' suivant la variante décrite plus haut pour obtenir $inv(s',s')$ et u' ;
   - on renvoie le couple $(\sigma(s) + \sigma(t) + inv(s',u'), u')$.

   En prenant le soin de ne pas recalculer inutilement la longueur à chaque étape de fusion, on obtiendra la complexité souhaitée.
5.

```
let separe u =
  let rec debut_fin u n =
    match n, u with
    | 0, _ -> [], u
    | _, x :: xs -> let a, b = debut_fin xs (n - 1) in (x :: a, b)
    | _ -> failwith "erreur" in
  debut_fin u ((List.length u) / 2)

let fusionne u v =
  let rec fus_aux u v n =
    match u, v with
    | [], _ -> (0, v)
    | _, [] -> (0, u)
    | x :: xs, y :: ys when x <= y ->
      let p, w = fus_aux xs v (n - 1) in
      (p, x :: w)
    | x :: xs, y :: ys ->
      let p, w = fus_aux u ys n in
      (p + n, y :: w)
  in
  let n = List.length u in
  fus_aux u v n


let nb_inv u =
  let rec aux u =
    match u with
    | [] | [_] -> (0, u)
    | _ -> let v, w = separe u in
      let n, a = aux v in
      let p, b = aux w in
      let q, c = fusionne a b in
      ((n + p + q), c) in
  let sigma, _ = aux u in sigma
```

`separe` est en $O(|u|)$ et `fusionne` en $O(|u| + |v|)$ (on ajoute un et un seul parcours à chaque fois pour calculer la longueur de la liste au début). On obtient donc la relation $T(n) \leqslant 2T(n/2) + An$ pour la complexité de `nb_inv`. C'est exactement la même relation que pour le tri fusion, et la complexité est donc identique : $O(n \log n)$.