Chapitre 5

Diviser pour régner

Option informatique MPSI1 & MPSI2

L'informatique est souvent utilisée pour faire des calculs répétés ou portant sur des ensembles d'objets. Les boucles consistent à travailler sur une valeur ou un objet à la fois.
La récursivité appliquée à une liste peut être un peu différente : on considère la liste comme découpée en une tête et une queue, on traite la queue (récursivement) et on combine le résultat avec la tête.

Nous allons, dans ce chapitre, généraliser cette méthode en découpant l'ensemble des objets à traiter en deux parties (ou davantage), en traitant chaque partie puis en assemblant les résultats. C'est la méthode "*diviser pour régner*".

Ce chapitre contient divers exemple d'application de cette méthode.
On l'appliquera dans le chapitre suivant pour l'écriture d'algorithmes de tris qui seront plus efficaces que les tris vus dans le chapitre précédent.

Dans ce chapitre, la division entière, a/b dans OCaml, sera notée $a \div b$ en mathématiques, c'est la notation de l'école primaire. $a \div b = \lfloor \frac{a}{b} \rfloor$.

I Exponentiation rapide

La fonction récursive classique peut s'écrire

```
let rec puissance a n =
  if n <= 0
  then 1
  else a*(puissance a (n-1));;
```

Le nombre de multiplication pour calculer a^n est n.
On va prouver que l'on peut calculer plus efficacement les puissances en divisant n en 2 parties égales ou de différence 1.
Si n est pair, $n = 2p$, on peut écrire $a^{2p} = a^p.a^p$, on calcule a^p, **que l'on garde en mémoire**, puis on le multiplie par lui-même : on ajoute une seule multiplication au calcul de a^p.
De même, si $n = 2p+1$, $a^{2p+1} = a^p.a^p.a$ permet de calculer a^n en n'ajoutant que 2 multiplications après avoir calculé a^p.
On va utiliser la récursivité pour poursuivre cette idée.

```
let rec expRapide a n =
  match n with
  |0 -> 1
  |n -> let b = expRapide a (n/2) in
        if n mod 2 = 0
        then b*b
        else a*b*b;;
```

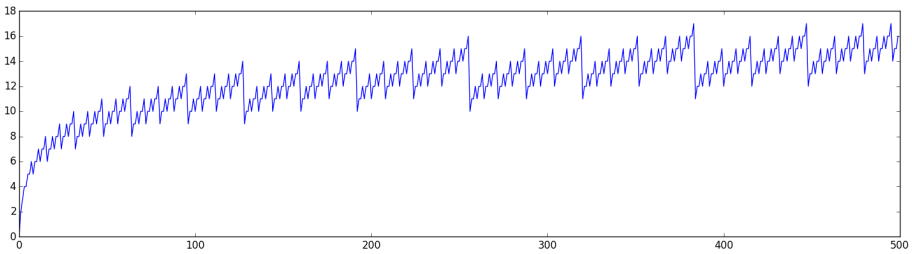
L'analyse est relativement simple

Terminaison Pour $n > 0$ on a $n \div 2 < n$ donc la variable n est un variant.

Preuve On prouve par récurrence sur n que le résultat est bien a^n.

Complexité Si on note $C(n)$ le nombre de multiplications effectuées lors de l'appel de `expRapide a n` on a $C(0) = 0$, $C(2p) = C(p) + 1$ et $C(2p+1) = C(p) + 2$.
On a donc $C(n) \leqslant C(n \div 2) + 2$ pour tout $n \geqslant 1$.
Si on a $2^p \leqslant n < 2^{p+1}$ alors $2^{p-1} \leqslant \lfloor \frac{n}{2} \rfloor < 2^p$ donc, par récurrence on a
$C(n) \leqslant C(1) + 2p$ pour $2^p \leqslant n < 2^{p+1}$ car 1 est le seul entier tel que $2^0 \leqslant n < 2^1$.
On en déduit que $C(n) \leqslant 2(p+1)$ pour $2^p \leqslant n < 2^{p+1}$ donc $p \leqslant \log_2(n)$.
Ainsi la complexité est majorée par $2\log_2(n) + 2$, c'est un $\mathcal{O}\big(\log_2(n)\big)$.
Le gain de rapidité est net : pour $n = 10^6$ on passe de 10^6 multiplications à moins de cinquante.

La valeur de a^n tend vers l'infini très rapidement, il ne semble pas utile d'avoir un algorithme rapide pour les grandes valeurs de n. En fait l'exponentiation est utilisée pour de très grandes valeurs de n en faisant les calculs modulo un entier N ; dans ce cas l'algorithme ci-dessus est indispensable.
Remarque : la complexité n'est pas une fonction croissante de n.



II Recherche dans un tableau

La recherche par dichotomie a déjà été étudiée avec Python.
Nous allons l'écrire de manière récursive, avec un découpage un peu différent.

```
let cherche x tab =
  let n = Array.length tab in
  let rec auxCh a b =
  if a = b
  then tab.(a) = x
  else let c = (a + b)/2 in
       if  tabl.(c) < x
       then auxCh (c+1) b
       else auxCh a c in
  aux 0 (n-1);;
```

II.1 Analyse de la recherche

La fonction `cherche` ne fait qu'appeler la fonction auxiliaire ; nous allons étudier la fonction récursive `auxCh`.

Terminaison

On montre par récurrence sur k la propriété
$\mathcal{P}(k)$: l'appel de `auxCh a b` termine pour $a \leqslant b \leqslant a + k$.
Pour $k = 0$, on a $a = b$ et `auxCh` renvoie le résultat de la comparaison
`tab.(a)= x` donc termine : $\mathcal{P}(0)$ est vérifiée.

On suppose que $\mathcal{P}(k)$ est valide avec $k \geqslant 0$.

On suppose qu'on a $a \leqslant b \leqslant a+k+1$alors $c = (a+b) \div 2 \geqslant a$.
De plus $c = (a+b) \div 2 \leqslant (2a+k+1) \div 2 = a + (k+1) \div 2 < a+k+1$ car $k+1 \geqslant 1$.
`auxCh a b` fait alors appel

- soit à `auxCh a c` avec $a \leqslant c \leqslant a+k$
- soit à `auxCh (c+1)b` avec $c+1 \leqslant b = a+k+1 \leqslant c+k+1 = (c+1)+k$.

Dans les deux cas les appels terminent d'après $\mathcal{P}(k)$ donc l'appel de `auxCh a b` termine : $\mathcal{P}(k+1)$ est vérifié. On a prouvé la récurrence.

Preuve

On montre par récurrence sur k la propriété
$\mathcal{Q}(k)$: l'appel de `auxCh a b` pour $a \leqslant b \leqslant a+k$ renvoie `true` si et seulement si il existe un indice i tel que $a \leqslant i \leqslant b$ et `tab.(i)` vaut x.
(*Par abus de langage, on dira dans ce cas que x est dans le tableau entre a et b.*)
Pour $k = 0$, on a $a = b$ et `auxCh` renvoie le résultat de la comparaison
`tab.(min)= x` : on teste le seul élément possible donc $\mathcal{Q}(0)$ est vérifiée.

On suppose que $\mathcal{Q}(k)$ est valide avec $k \geqslant 0$.
On suppose qu'on a $a \leqslant b \leqslant a+k+1$.
On a vu qu'on faisait alors appel à `auxCh a1 b1` avec $a_1 = a$ et $b_1 = c$ ou $a_1 = c+1$ et $b_1 = b$ avec, dans les deux cas, $a_1 \leqslant b_1 \leqslant a_1 + k + 1$.

- Si x n'est pas dans le tableau entre a et b alors il n'est pas dans le tableau entre a_1 et b_1 donc, d'après $\mathcal{Q}(k)$, l'appel récursif renvoie `false`.
- Si x est dans le tableau avec `tab.(c)< x` alors x n'est pas dans le tableau entre a et c donc il est dans le tableau entre $c+1$ et b et, d'après $\mathcal{Q}(k)$, l'appel de `auxCh (c+1)b` renvoie `true`.
- Si x est dans le tableau avec `tab.(c)= x` alors x est dans le tableau entre a et c donc, d'après $\mathcal{Q}(k)$, l'appel de `auxCh a c` renvoie `true`.
- Si x est dans le tableau avec `tab.(c)> x` alors x n'est pas dans le tableau entre c et b donc il est dans le tableau entre a et $c-1$ et, d'après $\mathcal{Q}(k)$, l'appel de `auxCh a c` renvoie `true`.

Dans tous les cas l'appel de `auxCh a b` renvoie la bonne réponse donc $\mathcal{Q}(k+1)$ est vérifiée.
On a prouvé la récurrence.

Complexité

On va évaluer la complexité en comptant le nombre de comparaisons entre la valeur recherchée et les valeurs du tableau.
Lors de chaque appel de la fonction auxiliaire on effectue 1 comparaison.

- Si le nombre d'indices parmi lesquels chercher est pair, $n = 2m$, on a $b = a+2m-1$ et $c = a+m-1$ donc on cherchera parmi m indices dans l'appel récursif entre a et c ou entre $c+1$ et b.
  Si on avait $n \leqslant 2^p$ alors on cherche parmi 2^{p-1} indices au plus dans l'appel récursif.
- Si le nombre d'indices est impair, $n = 2m+1$, on a $b = a+2m$ donc $c = a+m$. Entre a et c il y a donc $m+1$ indices parmi lesquels chercher
  et m indices entre $c+1$ et b.
  Si on avait $n \leqslant 2^p$ alors $n \leqslant 2^p - 1$ car n est impair donc $m = \frac{m-1}{2} \leqslant 2^{p-1} - 1$ donc on cherche parmi 2^{p-1} indices au plus dans l'appel récursif.
- Ainsi, si $n \leqslant 2^p$, on aboutit, au plus tard après p appels, à chercher parmi 2^0 élément donc le programme s'arrête après au plus $p+1$ comparaisons.
- Si p est tel que $2^{p-1} < n \leqslant 2^p$ on a $p \leqslant \log_2(n) + 1$ donc on effectue au plus $\log_2(n) + 2$ comparaisons : la complexité est un $\mathcal{O}\big(\ln(n)\big)$.

III Produit de polynômes

On s'intéresse aux opérations sur les polynômes dont les degrés peuvent être grands.
Le polynôme $P = a_0 + a_1 X + \cdots + a_n X^n$ sera représenté par le tableau

```
polP = [|a0; a1; ...; an|]
```

III.1 Opérations de base

Somme

La somme de deux polynômes est assez simple

```
let somme polP polQ =
  let n = Array.length polP in
  let m = Array.length polQ in
  let a = min n m in
  let b = max n m in
  let polS = Array.make b 0.0 in
  for i = 0 to (a-1) do polS.(i) <- polP.(i) +. polQ.(i) done;
  for i = a to (b-1) do
    polS.(i) <- if n < m then polQ.(i) else polP.(i) done;
  polS;;
```

La complexité est linéaire en la taille des tableaux, on effectue $\min(n, m)$ additions.

Somme

Le produit peut s'écrire assez simplement aussi

```
let produit polP polQ =
  let n = Array.length polP in
  let m = Array.length polQ in
  let prod = Array.make (n+m-1) 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (m-1) do
      prod.(i+j) <- prod.(i+j) +. polP.(i) *. polQ.(i) done done;
  prod;;
```

On effectue $n.m$ additions et $n.m$ multiplications .

III.2 Un produit plus rapide

Une méthode de type diviser pour régner permet d'améliorer la complexité du produit.
Pour simplifier les fonctions, on supposera que les deux polynômes à multiplier sont représentés par deux tableaux de même taille et que cette taille est une puissance de 2.
Les polynômes seront donc considérés comme appartenant à $\mathbb{R}_{2^n-1}[X]$.
Pour simplifier l'écriture on posera $p = 2^{n-1}$.
On peut écrire un polynôme de $\mathbb{R}_{2p-1}[X]$ sous la forme

$$P = \sum_{k=0}^{2p-1} a_k X^k = \sum_{k=0}^{p-1} a_k X^k + X^p \sum_{k=0}^{p-1} a_{p+k} X^k = P_1 + X^p P_2,\ P_1, P_2 \in \mathbb{R}_{p-1}[X]$$

De même on écrit $Q = Q_1 + X^p Q_2$ pour Q de degré $2p-1$.
On peut écrire le découpage d'un polynôme par la séparation d'un tableau, supposé de longueur paire.

```
let coupe tab =
  let m = (Array.length tab)/2 in
  let tab1 = Array.make m 0.0 in
  let tab2 = Array.make m 0.0 in
  for i = 0 to (m-1) do
    tab1.(i) <- tab.(i);
    tab2.(i) <- tab.(i+m) done;
  tab1, tab2;;
```

Il y 4 produits de polynômes de degré $p-1$ pour $P.Q = P_1.Q_1 + X^p(P1.Q_2 + P_2.Q_1) + X^{2p}P_2.Q_2$.
Un astuce difficile à trouver mais facile à prouver est que

$$P_1.Q_2 + P_2.Q_1 = (P_1 + P_2)(Q_1 + Q_2) - P_1.Q_1 - P_2.Q_2$$

On peut donc calculer les 3 termes avec seulement 3 produits de polynômes de degré $p-1$.
On va utiliser ce résultat de manière récursive :

Produit rapide de deux polynômes

```
let rec produit2 polP polQ =
  let m = (Array.length polP) / 2 in
  if m = 1
  then [|polP.(0) *. polQ.(0); 0.0|]
  else begin
    let p1, p2 = coupe polP in
    let q1, q2 = coupe polQ in
    let r1 = produit2 p1 q1 in
    let r2 = produit2 p2 q2 in
    let r3 = produit2 (somme p1 p2) (somme q1 q2) in
    let prod = Array.make (4*m) 0.0 in
    for i = 0 to (m-1) do
      prod.(i) <- r1.(i);
      prod.(i+m) <- r1.(i+m) +. r3.(i) -. r1.(i) -. r2.(i);
      prod.(i+2*m) <- r3.(i+m) -. r1.(i+m) -. r2.(i+m) +. r2.(i);
      prod.(i+3*m) <- r2.(i+m) done;
    prod end;;
```

On a besoin d'ajouter un 0 dans le cas de polynômes constants car on suppose que la longueur est doublée à chaque étape.

Complexité

Si $s(p)$ et $m(p)$ sont les nombres respectivement d'additions et de multiplications dans la fonction `produit` appliquée à deux tableaux de taille 2^p, on a $s(0) = 0$, $m(0) = 1$, $s(p+1) = 3s(p) + 6$ et $m(p+1) = 3m(p)$ donc $s(p) = 3^{p+1} - 3$ et $m(p) = 3^p$.
Dans le cas classique on avait $(2^p)^2 = 4^p$ opérations de chaque type.

III.3 Exercices

III.3.a Produit rapide de polynômes

Exercice 1 - Produit par un monôme
Écrire une fonction qui calcule le produit d'un polynôme par un terme de la forme $a_k X^k$ qui sera présenté sous la forme d'un réel, a_k, et d'un entier, k.

```
produitM float -> int -> float array -> float array
```

On veut utiliser le produit rapide pour des polynômes sans condition de degré. Pour cela

- On calcule la plus petite puissance de 2, n, supérieure ou égale aux longueurs des tableaux qui représente les polynômes
- On recopie les coefficients des polynômes dans des tableaux de longueur n.
- On calcule le produit rapide.
- On peut éliminer les 0 qu'on a placé en trop.

Exercice 2 - Produit rapide
En déduire une fonction qui calcule le produit rapide de deux polynômes.

Exercice 3 - Complexité
Quelle est la complexité de ce produit en fonction de n, maximum des longueurs des tableaux qui représente les polynômes ?
Comparer avec la complexité du produit classique, en $\mathcal{O}(n^2)$.

III.3.b Produit de matrices

On s'intéresse au produit de deux matrices carrées de taille n.

L'algorithme classique suit la définition mathématique : $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$.

Les matrices sont représentées par des tableaux doubles de flottants : `float array array`.

Exercice 4 - Produit classique
Écrire une fonction qui calcule ainsi le produit de deux matrices.

La complexité, en nombre d'opérations, est $2.n^3$ car il y a une addition et une multiplication par valeur des indices i, j et k.
En particulier le produit de deux matrices 2×2 demande 8 multiplications :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} \begin{pmatrix} a'+bc' & ab'+bd' \\ ca'+dc' & cb'+dd' \end{pmatrix}$$

Strassen, en 1969, a proposé une méthode pour calculer ce produit avec 7 multiplications.
On calcule les 7 produits (formules P) :

- $p_1 = (a+d).(a'+d')$
- $p_2 = (a-c).(a'+b')$
- $p_3 = (b-d).(c'+d')$
- $p_4 = a.(b'-d')$
- $p_5 = (a+b).d'$

- $p_6 = (c+d).a'$
- $p_7 = d.(a'-c')$

On les combine avec des additions ou des soustractions (formules C) :

- $aa' + bc' = p_1 + p_3 - p_5 - p_7$
- $ab' + bd' = p_4 + p_5$
- $ca' + dc' = p_6 - p_7$
- $cb' + dd' = p_1 - p_2 + p_4 - p_6$

On remarquera qu'on a effectué 18 additions et soustractions.
Une méthode de type diviser pour régner va utiliser le résultat ci-dessus pour les matrices d'ordre 2^r. En effet on peut écrire ces matrices par blocs où les blocs ont une taille $2^{r-1} \times 2^{r-1}$. On calculera le produit des blocs avec la méthode de Strassen pour ne faire que que 7 produits, ceux-ci étant calculés récursivement.

Exercice 5 - Fonctions auxiliaires 1 : sommations
Écrire des fonctions `somme` et `diff` qui calculent respectivement la somme et la différence de deux matrices carrées de même taille.

Exercice 6 - Fonctions auxiliaires 2 : blocs
Écrire une fonctions `decoupe` qui calcule les 4 blocs carrés d'une matrice carrée d'ordre pair et une fonctions `assemblage` qui reconstitue une matrice à partir de 4 blocs.

Exercice 7 - Produit rapide de matrices
Écrire une fonction de produit de matrices qui utilise la méthode de Strassen.

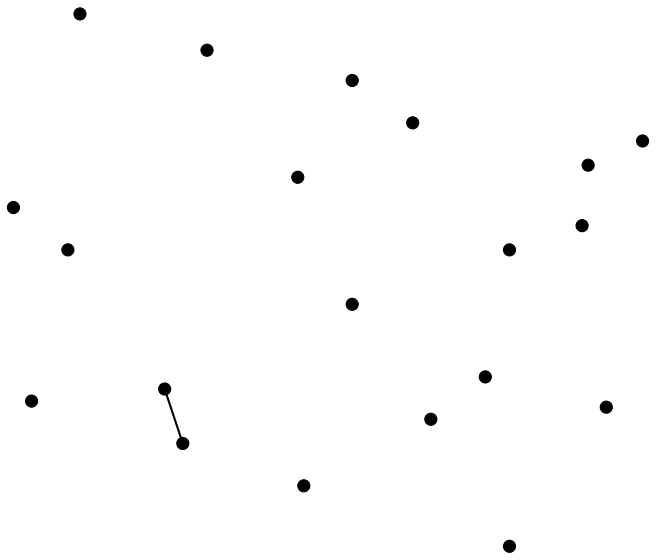
Exercice 8 - Premiers calculs de complexité
Calculer $m(r)$, le nombre de multiplications, et $a(r)$, le nombre d'additions et soustractions, effectuées dans le produit de deux matrices carrées de taille 2^r.
Pour quelles valeurs de r, le calcul est-il plus efficace que la méthode classique .

Exercice 9 - Cas général
Écrire une fonction de produit rapide pour des matrices de taille quelconque ; on pourra border les matrices par des 0.

Exercice 10 - Complexité
Quelle est la complexité de ce produit en fonction de n, maximum des tailles des matrices ?
Comparer avec la complexité du produit classique, en $\mathcal{O}(n^3)$.

IV Problème – Points proches

Le but est de trouver la paire (P, Q) dans un nuage de points du plan telle que la distance de P à Q est minimale parmi toutes les distance entre points.
Les points **distincts** seront donnés sous la forme d'une liste de paires de flottants.



Par exemple les points dessinés ci-dessus sont codés sous la forme

```
let pts = [(10.3, 5.4); ( 1.2, 2.5); ( 3.4, 2.7); (10.7, 2.4);
           ( 5.7, 1.1); ( 0.9, 5.7); ( 6.5, 7.8); ( 6.5, 4.1);
           (11.3, 6.8); ( 8.7, 2.9); ( 5.6, 6.2); ( 9.1, 5.0);
           ( 2.0, 8.9); ( 7.8, 2.2); ( 9.1, 0.1); ( 1.8, 5.0);
           ( 3.7, 1.8); ( 7.5, 7.1); (10.4, 6.4); ( 4.1, 8.3)];;
```

La distance entre deux points est calculée par la fonction

```
let distance p q =
  let x, y = p in
  let x', y' = q in
  sqrt ((x'-. x)**2.0 +. (y' -. y)**2.0);;
```

Exercice 11 - Premier algorithme
Écrire une fonction `points_proches` qui prend une liste de points comme paramètre et qui renvoie une paire de points **distincts** de la liste qui réalise le minimum de distance entre points ainsi que la distance entre ces points.
Combien de distances entre points sont-elles calculées ?

IV.1 Méthode diviser pour régner naïve

L'algorithme donné a une complexité quadratique.
On va utiliser une méthode diviser pour régner afin d'obtenir un algorithme de complexité quasi-linéaire.
Pour écrire cet algorithme on aura besoin de trier les points selon leur abscisse ou leur ordonnée.
On suppose donnée une fonction de tri de complexité quasi-linéaire, telle que le tri-fusion qui sera étudié dans le chapitre suivant.
Ce tri est implémenté en OCaml par la fonction

```
List.sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Le premier argument est une fonction de comparaison qui renvoie 0 si les deux éléments sont égaux, 1 si le premier est strictement supérieur au second et -1 sinon.

```
let compare_x p q =
   let x , y  = p in
   let x', y' = q in
   if x = x'
   then 0
   else if x > x'
        then 1
        else -1;;
```

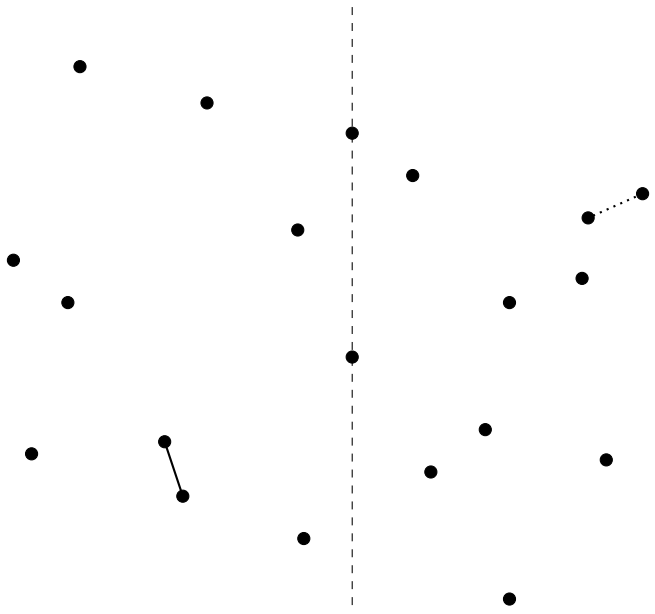
```
let compare_y p q =
   let x , y  = p in
   let x', y' = q in
   if y = y'
   then 0
   else if y > y'
        then 1
        else -1;;
```

Pour déterminer la paire de points de distance minimale on propose l'algorithme suivant :

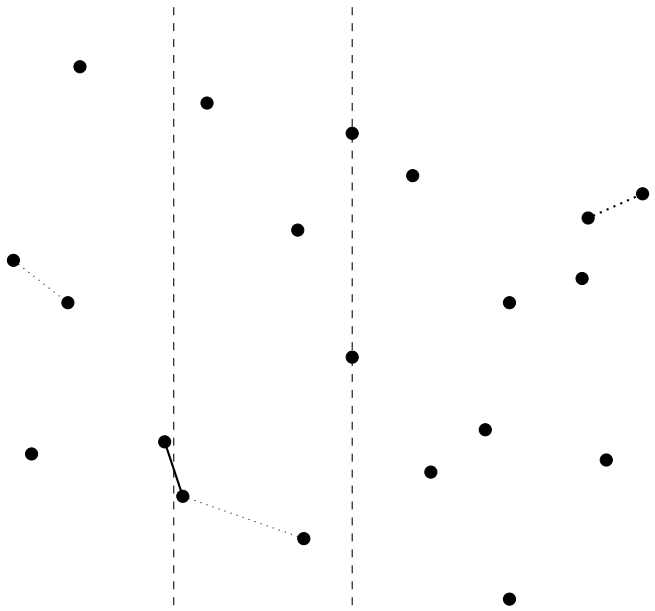
- on trie la liste des points selon les valeurs de x, la première coordonnée,
- on sépare en deux partie égales ou de cardinaux qui ne diffèrent que de 1,
- on calcule récursivement la paire de points de distance minimale dans chaque partie,
- on sélectionne laquelle des paires de points est la plus proche.

Exercice 12
Écrire une fonction `moities liste n` qui reçoit une liste (de taille n) et qui renvoie deux liste `l1` et `l2` telles que `liste = l1 @ l2`, `l1` est de taille $n \div 2$ et `l2` de taille $n - n \div 2$.

Voici une représentation de l'étape finale pour l'algorithme ci-dessus.
On a calculé les minimums de chaque coté et on trouve bien le minimum global.

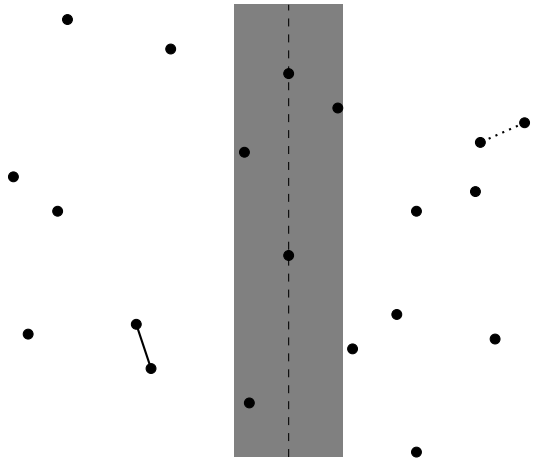


Le problème est qu'en agissant de la sorte on n'a pas toujours le bon résultat : en effet la distance minimale peut être atteinte pour deux points de chaque coté de la séparation. C'est le cas de l'exemple lors du second appel récursif de la partie gauche.
Le minimum de gauche n'est pas calculé, on aurait eu comme résultat le minimum des deux distances pointillées à gauche.
Il faut donc rassembler les résultats plus finement.



IV.2 Bande centrale

On note x_0 une valeur de séparation : les points de gauche ont une abscisse inférieure ou égale à x_0 et les points de droite ont une abscisse supérieure ou égale à x_0. Dans la suite on prendra pour valeur de x_0 l'abscisse du premier point, p_0, de la partie de droite : `p0 = List.hd l2` si `l1, l2` est le résultat de `moities liste`.
On note δ le minimum de deux distances minimales entre points de la partie gauche et entre points de la partie droite. C'est un candidat à la distance minimale mais il peut exister une distance plus petite entre points de chaque coté de la frontière entre les deux parties.
On peut remarquer que, pour un tel couple de points,
les abscisses appartiennent à $[x_0 - \delta; x_0 + \delta]$.
On va donc extraire les points de la bande $[x_0 - \delta; x_0 + \delta] \times \mathbb{R}$ et on cherche la distance minimale. Il faut cependant éviter de retrouver une complexité quadratique : tous les points pourraient se retrouver dans cette bande.



Pour chercher les distances minimales à partir d'un point (x, y) on cherche les distance avec les successeurs (x', y') , c'est-à-dire $y' \geqslant y$, en s'arrêtant lorsque $y' \geqslant y + \delta$ car on ne trouvera plus de points à une distance plus proche de (x, y) que δ.
On peut donc écrire l'algorithme.
On commence par trier, une fois pour toute, la liste selon les abscisses.

On applique alors, récursivement, le calcul des points proches d'une liste

1. On sépare la liste.
2. On calcule les points proches de chaque partie, avec la distance.
3. On calcule le minimum des deux distances, δ, et on détermine la bande autour du point de séparation, de largeur δ.
4. On trie la bande selon l'ordonnée.
5. On recherche la paire de points les plus proches de la bande en ne gardant que les distances inférieures à δ, il peut ne pas en exister.
6. On conclut en comparant les résultats.

Exercice 13
Écrire une fonction `bande liste p0 delta` qui reçoit une liste de points, un point `p0` et un flottant `delta` et qui renvoie la liste des éléments de cette liste qui vérifient que leur abscisse appartient à l'intervalle $[x_0 - \delta; x_0 + \delta]$ où x_0 est l'abscisse de `p0`.

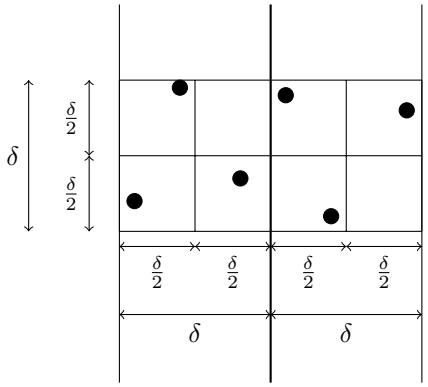
Exercice 14
Écrire une fonction `points_proches_bande liste delta` qui prend une liste de points et un flottant comme paramètres et qui renvoie `Some p q d` si p et q sont des points de la bande à distance minimale d avec $d \leqslant \delta$ ou `None` si un tel triplet n'existe pas.

Exercice 15
Écrire une fonction `points_proches liste` qui prend une liste de points comme paramètre et qui renvoie une paire de points **distincts** de la liste qui réalise le minimum de distance entre points ainsi que la distance entre ces points.

IV.3 Complexité

On va maintenant prouver que l'on a bien amélioré la complexité.

La clé est de constater que la fonction `points_proches_bande` ne fait pas beaucoup de tests. On a séparé les points en deux parties et on a calculé la valeur de δ.
On cloisonne la bande en carrés de cotés $\frac{\delta}{2}$; la distance minimale entre deux points d'un même carré est la diagonale, $\frac{\sqrt{2}\delta}{2} < \delta$.



Exercice 16 - Nombre de calculs
Prouver que chaque carré de cotés $\frac{\delta}{2}$ inclus dans une partie contient au plus un point.
En déduire que la fonction `point_proche_bande` effectue au plus 8 recherches de points.

Exercice 17 - Complexité de la recherche dans la bande
Prouver que la complexité de `points_proches_bande` en nombre d'appels à la fonction `ecart` est linéaire par rapport à la taille de la liste.

On note $C(n)$ le nombre maximal d'appels à la fonction `distance` dans l'exécution de `points_proches liste` pour des listes de taille n.

Exercice 18 -
Prouver que $C(n) \leqslant C(n \div 2) + C(n - n \div 2) + K \cdot n$ pour $n \geqslant 2$.
En déduire qu'on a $C(n) \leqslant 18.p.2^p$; écrire$C(n)$ sous la forme d'un $\mathcal{O}\big(f(n)\big)$.

Malgré le gros coefficient l'algorithme est plus rapide que l'algorithme naïf.
Voici les temps obtenus

n	algorithme basique	diviser pour régner
1000	0,1 s	0,004 s
2000	0,5 s	0,01 s
5000	3 s	0,05 s
10000	14 s	0,08 s
20000	60 s	0,2 s
50000	500 s	0,004 s

Le calcul ci-dessus est trompeur : en effet on a estimé le nombre de calculs de distances mais le tri de la bande pourrait introduire de nombreuses comparaisons qui n'ont pas été prises en compte.
On note $K(n)$ le nombre maximal d'opérations et de comparaisons dans l'exécution de la fonction auxiliaire pour des listes de taille n.

Exercice 19
Prouver que $K(n)$ est un $\mathcal{O}\big(n \log_2^2(n)\big)$.

Exercice 20
Comment peut-on assurer une complexité en $\mathcal{O}\big(n \cdot \log_2(n)\big)$?
On pourra utiliser la fusion du tri fusion.

Solutions

Solution de l'exercice 1 - Produit par un monôme

```
let produitM a k  pol =
  let n = Array.length pol in
  let prod = Array.make (n+k) 0.0 in
  for i = 0 to (n-1) do
    prod.(i+k) = a *. p.(i) done;
  prod;;
```

On effectue n multiplications.

Solution de l'exercice 2 - Produit rapide

```
let produit p1 p2 =
  let n1 = Array.length p1 in
  let n2 = Array.length p2 in
  let m = max n1 n2 in
  let n = ref 1 in
  while !n < m do n := 2 * !n done;
  let q1 = Array.make !n 0.0 in
  let q2 = Array.make !n 0.0 in
  for i = 0 to n1 - 1 do
    q1.(i) <- p1.(i) done;
  for i = 0 to n2 - 1 do
    q2.(i) <- p2.(i) done;
  let p = produit2 q1 q2 in
  let d = n1 + n2 -1 in
  let prod = Array.make d 0.0 in
  for i = 0 to (d-1) do prod.(i) <- p.(i) done;
  prod;;
```

Solution de l'exercice 3 - Complexité

On a cherché 2^p tel que $2^{p-1} < n \leqslant 2^p$.
On a vu qu'alors le nombre d'opérations est $3^{p+1} - 3 + 3^p = 4.3^p - 3 \leqslant 4.3^p$.
Or on a $p \leqslant \log_2(n) + 1$ donc le nombre d'opérations est majoré par $8.3^{\log_2(n)} = 8.n^{\log_2(3)}$, c'est donc un $\mathcal{O}\left(n^{\log_2(3)}\right)$ avec $\log_2(3) \simeq 1,58...$. La complexité asymptotique est meilleure.

Solution de l'exercice 4 - Produit classique

```
let produit_matrices a b =
  (* Les matrices doivent avoir la même taille *)
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      let s = ref 0.0 in
      for k = 0 to (n-1) do
        s:= !s +. a.(i).(k) *. b.(k).(j) done;
      c.(i).(j) <- !s done done;
  c;;
```

Solution de l'exercice 5 - Fonctions auxiliaires 1 : sommations

```
let somme a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) +. b.(i).(j) done done;
  c;;
```

```
let diff a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) -. b.(i).(j) done done;
  c;;
```

Solution de l'exercice 6 - Fonctions auxiliaires 2 : blocs

```
let decoupe m =
  let n = (Array.length m)/2 in
  let m1 = Array.make_matrix n n 0.0 in
  let m2 = Array.make_matrix n n 0.0 in
  let m3 = Array.make_matrix n n 0.0 in
  let m4 = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m1.(i).(j) <- m.(i).(j);
      m2.(i).(j) <- m.(i).(j+n);
      m3.(i).(j) <- m.(i+n).(j);
      m4.(i).(j) <- m.(i+n).(j+n) done done;
  m1, m2, m3, m4;;
```

```
let assemblage m1 m2 m3 m4 =
  let n = Array.length m1 in
  let m = Array.make_matrix (2*n) (2*n) 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m.(i).(j) <- m1.(i).(j);
      m.(i).(j+n) <- m2.(i).(j);
      m.(i+n).(j) <- m3.(i).(j);
      m.(i+n).(j+n) <- m4.(i).(j) done done;
  m;;
```

Solution de l'exercice 7 - Produit rapide de matrices

```
let rec prod a b =
  let n = Array.length a in
  if n = 1
  then [|[|a.(0).(0) *. b.(0).(0)|]|]
  else let a1, a2, a3, a4 = decoupe a in
       let b1, b2, b3, b4 = decoupe b in
       let p1 = prod (somme a1 a4) (somme b1 b4) in
       let p2 = prod (diff a1 a3) (somme b1 b2) in
       let p3 = prod (diff a2 a4) (somme b3 b4) in
       let p4 = prod a1 (diff b2 b4) in
       let p5 = prod (somme a1 a2) b4 in
       let p6 = prod (somme a3 a4) b1 in
       let p7 = prod a4 (diff b1 b3) in
       let c1 = diff (somme p1 p3) (somme p5 p7) in
       let c2 = somme p4 p5 in
       let c3 = somme p6 p7 in
       let c4 = somme (diff p1 p2) (diff p4 p6) in
       assemblage c1 c2 c3 c4;;
```

Solution de l'exercice 8 - Premiers calculs de complexité

On a $m(0) = 1$ et $m(r+1) = 7m(r)$ donc $m(r) = 7^r$.
On a $a(0) = 0$ et $a(r+1) = 7a(r) + 18 \cdot (2^r)^2$ car on calcule 18 fois la somme ou le produit de matrices carrées d'ordre 2^r. La partie $a(r+1) = 7a(r)$ permet de poser $a(r) = 7^r.u_r$.
On obtient alors $u_{r+1} = u_r + \frac{18}{7} \cdot \left(\frac{4}{7}\right)^r$ d'où

$$u_r = u_0 + \frac{18}{7} \cdot \sum_{k=0}^{r-1} \left(\frac{4}{7}\right)^k = \frac{18}{7} \cdot \frac{1 - \left(\frac{4}{7}\right)^r}{1 - \frac{4}{7}} = 6 \cdot \left(1 - \left(\frac{4}{7}\right)^r\right).$$

Ainsi $a(r) = 6 \cdot (7^r - 4^r)$.

Au total on effectue donc $7^{r+1} - 6.4^r$ opérations alors que la méthode classique en demande $2 \cdot (2^r)^3 = 2^{3r+1}$. La méthode classique est plus efficace pour $r \leqslant 9$; pour $r = 10$, $n = 1024$, le nombre d'opérations est de l'ordre de $2 \cdot 10^9$.

Solution de l'exercice 9 - Cas général

```
let borde mat n =
  let p = Array.length mat in
  let q = Array.length mat.(0) in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (p-1) do
    for j = 0 to (q-1) do
        c.(i).(j) <- mat.(i).(j) done done;
  c;;
```

```
let extraction mat p q =
  let c = Array.make_matrix p q 0.0 in
  for i = 0 to (p-1) do
    for j = 0 to (q-1) do
        c.(i).(j) <- mat.(i).(j) done done;
  c;;
```

```
let produit_matrice_rapide a b =
  let p1 = Array.length a in
  let q1 = Array.length a.(0) in
  let p2 = Array.length b in
  let q2 = Array.length b.(0) in
  if q1 <> p2
  then failwith "Les matrices ne peuvent pas être multipliées"
  else let m = max (max p1 q1) q2 in
  let n = ref 1 in
  while !n < m do m := 2 * !m done:
  let a1 = borde a n in
  let b1 = borde b n in
  extraction (prod a1 b1) p1 q2;;
```

Solution de l'exercice 10 - Complexité
On a cherché 2^p tel que $2^{p-1} < n \leqslant 2^p$.
On a vu qu'alors le nombre d'opérations est $7^{p+1} - 6.4^p \leqslant 7 \cdot 7^p$.
Or on a $p \leqslant \log_2(n) + 1$ donc le nombre d'opérations est majoré par $14 \cdot 7^{\log_2(n)} = 17. \cdot n^{\log_2(7)}$, c'est donc un $\mathcal{O}\left(n^{\log_2(7)}\right)$ avec $\log_2(7) \simeq 2,807....$ La complexité asymptotique est meilleure.

Solution de l'exercice 11 - Premier algorithme
On commence par calculer le point le plus proche d'un point dans une liste.

```
let rec point_proche p liste =
   match liste with
   |[] -> failwith "point-proche : la liste est vide"
   |[q] -> q, (distance p q)
   |q::reste -> let m, d = point_proche p reste in
                let d1 = distance p q in
                if d1 < d
                then q, d1 else m, d;;
```

On calcule une distance par point de la liste.

```
let rec points_proches liste =
  match liste with
  |[] -> failwith "points_proches : la liste est vide"
  |[p] -> failwith "points_proches : il n'y a qu'un seul point"
  |[p; q] -> p, q, (distance p q)
  |p::reste -> let q, r, d = points_proches reste in
               let p1, d1 = point_proche p reste in
               if d1 < d
               then p, p1, d1 else q, r, d ;;
```

On note $C(n)$ le nombre de calculs de distance lors de l'appel de `points_proches` avec une liste de taille n. On a $C(2) = 1$
Pour une liste de taille n on invoque la fonction `point_proche` avec le reste de la liste : on effectue alors $n-1$ calculs de distance.
On a donc $C(n) = C(n-1) + (n-1)$ d'où $C(n) = \frac{n(n-1)}{2}$.

Solution de l'exercice 12

```
let rec decoupage k liste =
   if k = 0
   then [], liste
   else match liste with
        |[] -> failwith "La liste est trop courte"
        |t::q -> let l1, l2 = decoupage (k-1) q in
                 (t::l1), l2;;

let moities liste =
   let n = List.length liste in
   decoupage (n/2) liste;;
```

Solution de l'exercice 13

```
let rec bande liste p0 delta =
   let x0, y0 = p0 in
   match liste with
   |[] -> []
   |(x, y):: q when x < x0 -. delta || x > x0 +. delta -> bande q
      p0 delta
   |(x, y):: q -> (x, y) :: (bande q p0 delta);;
```

Solution de l'exercice 14

On commence par chercher le point proche au-dessus.

```
let rec point p liste delta =
   let (a, b) = p in
   match liste with
   |[] -> None
   |(x, y)::reste when x > a +. delta -> None
   |q::reste when distance p q > delta -> point p reste delta
   |q::reste -> begin
                  let d = distance (a, b) q in
                  match point p reste delta with
                   |Some (q1,d1) when d1 < d -> Some (q1, d1)
                  |_ -> Some (q, d)
                end;;
```

On va comparer plusieurs fois des résultats sous forme de triplets optionnels.

```
let meilleur triplet1 triplet2 =
   match triplet1, triplet2 with
   |None, _ -> triplet2
   |_, None -> triplet1
   |Some (p1, q1, d1), Some (p2, q2, d2) -> if d1 < d2 then
      triplet1 else triplet2;;
```

```
let rec points_bande liste delta =
   match liste with
   |[] -> None
   |p::reste -> begin
                  let triplet = points_bande reste delta in
                  match point p reste delta with
                  |None ->  triplet
                  |Some (q, d) -> meilleur (Some (p, q, d))
                     triplet
                end;;
```

Solution de l'exercice 15

```
let delta triplet =
   match triplet with
   |None -> max_float
   |Some(p, q, d) -> d;;
```

```
let points_proches points  =
   let liste_x = List.sort compare_x points in
   let n0 = List.length points in
   let rec aux liste n =
      match liste with
      |[] -> None
      |[p] -> None
      |[p; q] -> Some (p, q, distance p q)
      |p::reste -> let gauche, droite = moities liste n in
                   let p0 = List.hd droite in
                   let triplet_g = aux gauche (n/2) in
                   let triplet_d = aux droite (n - n/2)in
                   let triplet_e =  meilleur triplet_g triplet_d
                      in
                   let d = delta triplet_e in
                   let b1 = bande liste p0 d in
                   let b = List.sort compare_y b1 in
                   let triplet_c = points_bande b d in
                   meilleur triplet_e triplet_c
  in aux liste_x n0;;
```

Solution de l'exercice 16 - Nombre de calculs

Comme chaque carré est inclus dans une partie (droite ou gauche) les distances entre deux points est au moins δ. Si deux points étaient dans un même carré leur distance serait majorée par la diagonale qui $\frac{\delta}{\sqrt{2}} < \delta$: c'est impossible.

Quand on calcule le résultat de `point_proche_bande` pour un point (a, b) on est limité au rectangle $[x_0 - \delta; x_0 + \delta] \times [b; b + \delta]$. Ce rectangle contient 4 carrés de coté $\frac{\delta}{2}$: chacun ne peut contenir qu'un point. Il y a 8 points au maximum dans le rectangle dont le point p dont on cherche les voisins. Au pire on calculera les distances de ces 8 points avant de voir un point dont l'abscisse est trop éloignée.

En fait une analyse géométrique plus précise permet de remplacer 8 par 6.

Solution de l'exercice 17 - Complexité de la recherche dans la bande

Lors de l'appel de `points_proches_bande` avec une liste de taille $n \geqslant 1$ on rappelle `points_proches_bande` avec une liste de taille $n - 1$, on appelle `point_proche_bande` donc on calcule au plus 16 écarts et on compare deux calculs d'écarts. Ainsi on effectue au plus 18 calculs d'écarts en plus de l'appel

récursif. Comme on ne calcule aucun écart lorsque la liste est de taille 0 ou 1, la complexité est majorée par $18(n-1)$ lorsque la liste est de taille n.

Solution de l'exercice 18 -
On a $C(n) \leqslant C(n_1) + C(n_2) + K \cdot m$ où n_1 et n_2 sont les nombres de points à gauche et à droite :$n_1 = n \div 2$ et $n_2 = n - n_1$ et m est le nombre de points dans la bande, $m \leqslant n$ (la bande peut contenir les n points).
On a $C(2) = 1$ et, pour $n \leqslant 2^p$, on a $n_1 \leqslant 2^{p-1}$ et $n_2 \leqslant 2^p$.
On prouve donc classiquement que $C(n) \leqslant K \cdot p \cdot 2^p$ pour $n \leqslant 2^p$ ce qui donne une complexité en $\mathcal{O}\big(n.\ln(n)\big)$ en choisissant p tel que $2^{p-1} < n \leqslant 2^p$.

Solution de l'exercice 19
On aboutit à $K(n) \leqslant K(n_1) + K(n_2) + A \cdot n + B \cdot n \log_2(n) leK(n_1) + K(n_2) + C \cdot n \log_2(n)$.
Si on note k_p un majorant de $K(n)$ pour $2^{p-1} \leqslant n \leqslant 2^p$ on a $2^{p-2} \leqslant n_1 = n \div 2 \leqslant 2^{p-1}$ et $2^{p-2} \leqslant n_2 = n - n \div 2 \leqslant 2^{p-1}$ d'où $k_p \leqslant 2k_{p-1} + C\dot{2}^p\dot{p}$.
Si on pose $u_p = 2^{-p}k_p$ on a $u_p \leqslant u_{p-1} + C \cdot p$; $u_0 = 0$ donne alors $u_p \leqslant C \cdot \frac{p(p+1)}{2} \leqslant C' \cdot p^2$ d'où $K(n) \leqslant 2^p C' \cdot p^2 = \mathcal{O}\big(n \cdot \log_2^2(n)\big)$.
Ajouter le tri initial, en $\mathcal{O}\big(n.\ln(n)\big)$, ne change pas l'ordre de complexité.

Solution de l'exercice 20
Il suffit de renvoyer la liste triée selon les ordonnées à chaque appel de la fonction auxiliaire et de remplacer le tri par la fusion des deux listes triées. La complexité ajoutée reste linéaire et on aboutit au résultat souhaité.

```
let rec fusion l1 l2 =
   match l1, l2 with
   |[], _ -> l2
   |_, [] -> l1
   |t1::q1, t2::q2 when compare_y t1 t2 = 1
                    -> t2 :: (fusion l1 q2)
   |t1::q1, _ -> t1 :: (fusion q1 l2);;
```

```
let points_proches2 points  =
   let liste_x = List.sort compare_x points in
   let n0 = List.length points in
   let rec aux liste n =
      match liste with
      |[] -> None, []
      |[p] -> None, [p]
      |[p; q] when compare_y p q = 1 -> Some (p, q, distance p q),
          [q; p]
      |[p; q]-> Some (p, q, distance p q), [p; q]
      |p::reste -> let gauche, droite = moities liste n in
                   let p0 = List.hd droite in
                   let triplet_g, tri_g = aux gauche (n/2) in
                   let triplet_d, tri_d = aux droite (n - n/2)in
                   let triplet_e =  meilleur triplet_g triplet_d
                       in
                   let d = delta triplet_e in
                   let liste_t = fusion tri_g tri_d in
                   let b = bande liste_t p0 d in
                   let triplet_c = points_bande b d in
                   meilleur triplet_e triplet_c, liste_t
  in fst (aux liste_x n0);;
```