

## Division en sous-problèmes

<b>I</b>	<b>Diviser pour régner</b>	<b>1</b>
I.1	Principe	1
I.2	Somme d'éléments dans un tableau	1
I.3	Tri fusion	2
I.4	Recherche dichotomique	4
I.5	Principe général d'analyse des récurrences	5
I.6	Nombre d'inversions	6
I.7	Points les plus proches	7
<b>II</b>	<b>Meet in the middle</b>	<b>10</b>
II.1	Principe	10
II.2	Sous-ensemble de somme donnée	10
<b>III</b>	<b>Dichotomie pour passer de décision à optimisation</b>	<b>10</b>
III.1	Principe	10
III.2	Couverture par des segments égaux	10
<b>IV</b>	<b>Problèmes</b>	<b>11</b>
IV.1	Multiplication d'entiers	11

Dans ce chapitre, on étudie des algorithmes ayant la particularité de reposer sur une division d'un problème en plusieurs sous-problèmes :

- soit pour les résoudre récursivement, on parle alors de la méthode *diviser pour régner*
- soit pour résoudre ces sous-problèmes, peut-être d'une manière différente, afin d'obtenir une solution plus efficace au problème principal, on parle de *rencontre au milieu* (ou *meet in the middle*)

### I Diviser pour régner

#### I.1 Principe

Le principe des algorithmes basé *Diviser pour régner* est de décomposer un problème en plusieurs sous-problèmes disjoints et de déduire des solutions de ces sous-problèmes une solution au problème de départ.

Le point clé pour ce principe est de pouvoir **fusionner** les solutions de sous-problèmes pour en faire une solution, et de pouvoir le faire dans un temps/espace raisonnable. On procède alors par récursivité en appliquant ce principe pour résoudre les sous-problèmes eux-mêmes jusqu'à tomber sur des sous-problèmes très simples.

Ainsi, si on a une entrée  $E$ , on va en déduire des entrées  $E_1, \dots, E_k$  pour des sous-problèmes tels que  $|E_i| \leq \left\lceil \frac{|E|}{p} \right\rceil$ , où  $p$  et  $k$  sont **indépendants** de  $E$ . Une fois les solutions  $S_1, \dots, S_k$  à ces sous-problèmes obtenus par des appels récursifs, on va en déduire par une **fusion** des solutions, la solution  $S$  au problème initial.

En terme de complexité temporelle, si  $T(n)$  est la complexité de résolution pour une solution de taille  $n$ , on aura :

$$T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

où  $f(n)$  est le coût de la fusion et en supposant, ce qui est très raisonnable, que  $T$  est croissante.

**Remarque** La croissance de  $T$  est naturellement déduite du fait que le problème sur une entrée de taille  $n$  est **plus facile** que le problème sur une entrée de taille  $p \geq n$ . On peut ainsi penser au tri d'un tableau où il suffit de rajouter des éléments factices au début ou à la fin pour se ramener à un problème sur une entrée de taille plus grande.

On ne donnera pas ici de théorème général permettant d'exprimer  $T(n)$  selon les différentes valeurs de  $f(n)$ , on se contentera de faire des preuves dans des cas particuliers.

## I.2 Somme d'éléments dans un tableau

On considère un tableau  $t$  de  $n$  nombres dont on cherche à calculer la somme  $S(t) = \sum_{i=0}^{n-1} t[i]$ . Il est possible de faire ce calcul très aisément avec une simple boucle :

**OCaml**

```
let somme t =
  let s = ref 0 in
  for i = 0 to Array.length t - 1 do
    s := !s + t.(i)
  done;
  !s
```

**C**

```
int somme(int *t, size_t nb)
{
  int s = 0;
  for(int i = 0; i < nb; i++)
  {
    s = s + t[i];
  }
  return s;
}
```

**Python**

```
def somme(t):
  s = 0
  for x in t:
    s = s + x
  return s
```

Il est possible, bien que cela ne soit pas naturel, de traiter ce problème avec un algorithme *diviser pour régner*. En effet, on peut couper le tableau en deux, calculer les deux sous-sommes  $S_1$  et  $S_2$ , puis les fusionner de manière triviale en calculant la somme  $S = S_1 + S_2$ .

**OCaml**

```
let rec somme_div t i j =
  if i = j
  then t.(i)
  else
    let m = i + (j-i)/2 in
    let s1 = somme_div t i m in
    let s2 = somme_div t (m+1) j in
    s1 + s2

let somme t = somme_div t 0 (Array.length t - 1)
```

Si  $T(n)$  est le coût d'une somme d'un tableau à  $n$  éléments, on a donc  $T(1) = 1$  et

$$\forall n \geq 2, T(n) \leq 2T(\lceil n/2 \rceil) + O(1)$$

**Remarque** On constate ici qu'on utilise  $\lceil x \rceil$  la partie entière supérieure liée à une majoration. En effet, si on coupe l'entrée en deux et qu'elle est de taille impaire  $2p + 1$ , une des deux moitiés sera de longueur  $p + 1$  qui est la partie entière supérieure.

On aura alors  $T(n) = T(p) + T(p + 1) + O(1) \leq 2T(p + 1) + O(1)$  par croissance de  $T$ .

Les résolutions de récurrence vont se faire en deux temps : un cas où toutes les divisions tombent justes, c'est-à-dire ici une puissance de deux, et un théorème où on s'y ramène pour un entier quelconque par encadrement et croissance de  $T$ .

**Lemme I.1**  $\forall n \in \mathbb{N}, T(2^n) = O(2^n)$

### ■ Preuve

On commence par expliciter le  $O(1)$  présent dans la récurrence : il existe  $M \geq 1$  tel que  $\forall n \in \mathbb{N}^*, T(n) \leq 2T(\lceil n/2 \rceil) + M$ .

Montrons par récurrence que  $\forall n \in \mathbb{N}, T(2^n) \leq M2^n$

- a) On a  $T(2^0) = 1 \leq M2^0$ .
- b) Si  $T(2^n) \leq M2^n$  alors  $T(2^{n+1}) \leq 2T(2^n) + M \leq M2^n + M \leq M2^{n+1}$ .  
On a ainsi  $T(2^n) = O(2^n)$ .

**Théorème I.2**  $\forall n \geq 1, T(n) = O(n)$ .

■ **Preuve**

Soit  $n \in \mathbb{N}^*$ , il existe  $p \in \mathbb{N}$  tel que  $2^p \leq n < 2^{p+1}$ . Par croissance de  $T$ , on a  $T(n) \leq T(2^{p+1})$ . Or, il existe  $M'$  tel que  $\forall k, T(2^k) \leq M'2^k$ .  $T(n) \leq M'2^{p+1} \leq 2M'n$ .

Donc,  $T(n) = O(n)$ .

## I.3 Tri fusion

L'algorithme du tri fusion est un des exemples les plus importants d'algorithmes *Diviser pour régner* :

- Étant donnée une liste  $l$  de taille  $n \geq 2$ , on va considérer les sous-listes  $l_p$  des valeurs d'indice pair et  $l_i$  des valeurs d'indice impair.
- On trie ensuite  $l_1$  et  $l_2$  pour obtenir  $l'_1$  et  $l'_2$ .
- On fusionne ces deux listes pour obtenir  $l' = \text{fusion}(l'_1, l'_2)$  liste triée déduite de  $l$ .

Comme expliqué dans le paragraphe précédent, les tris de  $l_1$  et  $l_2$  s'effectuent eux-aussi à l'aide d'un tri fusion.

■ **Note 1** TODO : dessin

Voici une implémentation en OCaml de cet algorithme :

OCaml

```

let rec separe_en_deux l =
  match l with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x::y::q -> let l1, l2 = separe_en_deux q in
    (x::l1, y::l2)

let rec fusionne l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x::q1, y::q2 ->
    if x < y
    then x :: (fusionne q1 l2)
    else y :: (fusionne l1 q2)

let rec tri_fusion l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ ->
    let l1, l2 = separe_en_deux l in
    let l1p = tri_fusion l1 in
    let l2p = tri_fusion l2 in
    fusionne l1p l2p

```

La correction et la terminaison de cet algorithme ne posant aucune difficulté, on va se concentrer sur le calcul de la complexité temporelle :

- `separe_en_deux` consiste en un parcours linéaire de la liste  $l$  donc  $O(|l|)$ .
- `fusionne` supprime un élément d'une des deux listes à chaque appel récursif, donc une complexité en  $O(|l_1| + |l_2|)$ .
- Pour `tri_fusion` la situation est plus complexe en raison du double appel récursif. On va d'abord traiter le cas des listes contenant  $2^k$  éléments.

Notons  $t_n$  la complexité temporelle pour  $|l| = n$ .

**Lemme I.3**  $t_{2^n} = O(2^n \log_2 2^n)$

■ **Preuve**

Par l'analyse de complexité des deux fonctions auxiliaires, on a pour  $n \in \mathbb{N}$

$$t_{2^{n+1}} = 2t_{2^n} + O(2^n) \leq 2t_{2^n} + M2^n$$

où on peut supposer que  $M \geq 1$ .

On va montrer par récurrence sur  $n \in \mathbb{N}^*$  que  $t_{2^n} \leq 2Mn2^n$ .

- Initialisation :  $t_{2^1} = 2t_1 + M2 = 2M + 2 \leq 4M = 2 \times 1 \times 2^1 M$ .
- Hérédité : si  $n \in \mathbb{N}^*$  et l'hypothèse est vérifiée pour  $t_{2^n}$ , alors  $t_{2^{n+1}} \leq 4n2^n M + M2^n = (4n + 1)M2^n \leq 4(n + 1)M2^n \leq 2M(n + 1)2^{n+1}$ .

Ainsi  $t_{2^n} = O(n2^n)$ . ■

**Théorème I.4**  $t_n = O(n \log_2 n)$

■ **Preuve**

Le lemme assure qu'il existe  $M'$  tel que  $\forall p \in \mathbb{N}^*, t_{2^p} \leq M'p2^p$ .

Soit  $n \in \mathbb{N}^*$  on sait qu'il existe  $p$  tel que  $2^{p-1} \leq n < 2^p$  et donc  $p - 1 \leq \log_2 n < p$ . Par croissance de  $t$  (direct car on a plus d'éléments à trier) on a  $t_n \leq t_{2^p} \leq M'p2^p \leq 2M'(1 + \log_2 n)n = O(n \log_2 n)$ .

Ainsi,  $t_n = O(n \log_2 n)$ . ■

**Remarque** On a utilisé implicitement la croissance de  $t_n$  ici : plus la liste est longue, plus on effectue d'opérations.

Le programme suivant présente une implémentation du tri fusion reposant sur des tableaux. Les sous-tableaux sont manipulés à l'aide de leurs indices de début et de fin comme pour la recherche dichotomique.

OCaml

```
let fusionne t1 t2 =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let i1 = ref 0 in
  let i2 = ref 0 in
  let i = ref 0 in
  let t = Array.make (n1+n2) t1.(0) in
  while !i1 < n1 || !i2 < n2 do
    if !i1 = n1 || (!i2 < n2 && t1.(i1) > t2.(i2))
    then begin
      t.(i) <- t2.(i2);
      incr i2
    end else begin
      t.(i) <- t1.(i1);
      incr i1
    end;
    incr i
  done;
  t

let rec tri_fusion_aux t i j =
  let n = j - i in
  if n >= 1
  then begin
    let m = i + n / 2 in
    let t1 = tri_fusion_aux t i m in
    let t2 = tri_fusion_aux t (m+1) j in
    print_t t1;
    print_t t2;
    fusionne t1 t2
  end
  else if n = 0
```

```

    then [|t.(i)|]
    else [|]|

let tri_fusion t =
  let n = Array.length t in
  tri_fusion_aux t 0 (n-1)

```

■ **Note 2** TODO : exercice tri avec un tableau et tri **en place**

## I.4 Recherche dichotomique

On a déjà vu la recherche dichotomique, ici, on coupe le tableau en deux mais on n'effectue qu'un seul appel récursif, la récurrence est donc en

$$T(n) = T(\lceil n/2 \rceil) + O(1)$$

et on obtient facilement  $T(n) = O(\log_2 n)$ .

## I.5 Principe général d'analyse des récurrences

On remarque ici qu'on retombe souvent sur la même méthode pour analyser des récurrence de la forme

$$\forall n \in \mathbb{N}^*, T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

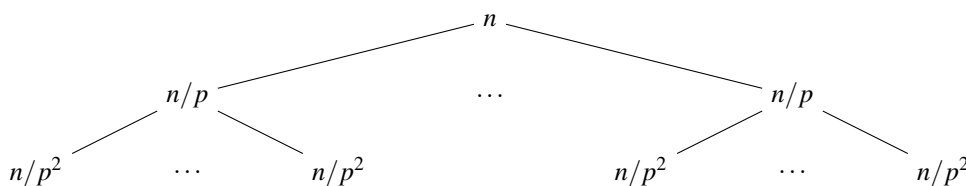
Comme on ne considère que des suites positives et croissantes, on peut se ramener à une égalité par majoration. En effet, si  $S$  vérifie  $S(0) = T(0)$  et  $\forall n \in \mathbb{N}^*, S(n) = kS\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$  alors on a  $\forall n \in \mathbb{N}, T(n) \leq S(n)$  par une récurrence immédiate et d'une majoration de  $S(n)$  on en déduira directement une majoration pour  $T(n)$ .

Cette remarque peut donc se résumer ainsi :

**Remarque** Il est toujours possible de se ramener à une relation de récurrence avec une égalité.

Que signifie cette relation de récurrence : que pour résoudre l'instance de taille  $n$  on va faire  $k$  appels récursifs à des instances de taille au plus  $\lceil n/p \rceil$  et le coup de la fusion des résultats sera  $f(n)$ .

On peut ainsi représenter les coûts de fusion cumulés qui sont associés à chaque niveau de l'arbre d'appels récursifs. Pour simplifier, on considère que  $n = p^l$  ce qui permet de n'avoir que des divisions entières tout au long de l'arbre. On remarque qu'on peut s'y ramener par croissance en majorant  $n$  par une puissance de  $p$  et on qu'on a alors  $l = O(\log_p n)$ .



On a alors les coûts de fusion

$$T(n) = f(n) + kf(n/p) + k^2f(n/p^2) + \dots = \sum_{i=0}^l k^i f(n/p^i)$$

On a trois cas standard pour cette somme :

- Poids sur la racine, c'est  $f(n)$  qui l'emporte sur les coûts et le reste de l'arbre n'a pas d'influence sur la complexité. Ici  $T(n) = f(n)$ .
- Poids sur les feuilles, le coût de fusion n'est pas important et c'est le nombre de feuilles qui compte, on a alors  $T(n) = k^l f(1) = O(k^l) = O(k^{\log_p n}) = O(n^{\log_p k})$ .
- Poids réparti uniformément, on a chaque  $k^i f(n/p^i) = O(f(n))$  et donc  $T(n) = O(f(n) \log n)$ .

**Exemple** • Dans le cas de la multiplication de Karatsuba (exercice) on a  $T(n) = 3T(\lceil n/2 \rceil) + O(n)$  donc les niveaux sont de plus en plus peuplés, on est dans le second cas et le niveau final va l'emporter. Ainsi,  $T(n) = O(n^{\log_2 3})$ .

• Dans le cas du tri fusion  $T(n) = 2T(\lceil n/2 \rceil) + O(n)$  on a un équilibre du travail total de fusion qui reste linéaire pour chaque niveau. Donc, on est dans le troisième cas et  $T(n) = O(n \log n)$ .

**Remarque** Pour retrouver des expressions rapidement, on peut étudier  $u_n = \frac{T(p^n)}{k^n}$ .

En effet, de la récurrence  $T(p^n) = kT(p^{n-1}) + f(p^n)$  on déduit  $u_n = u_{n-1} + \frac{f(p^n)}{k^n}$  et donc  $u_n = \sum_{i=0}^n \frac{f(p^i)}{k^i} = O(g(n))$  qui se simplifie en général. Pour obtenir ensuite  $T(p^n) = k^n u_n = O(k^n g(n))$  puis  $T(n) = O(n^{\log_p k} g(\log_p n))$ .

Par exemple, pour  $T(n) = 2T(\lceil n/2 \rceil) + O(n)$  on a  $u_n = T(2^n)/2^n = \sum_{i=0}^n \frac{O(2^i)}{2^i} = O(n)$  donc  $T(2^n) = 2^n O(n) = O(n 2^n)$  puis  $T(n) = O(n \log n)$ .

Autre exemple, pour  $T(n) = 2T(\lceil n/2 \rceil) + O(1)$ , on a  $u_n = \sum_{i=0}^n 2^{-i} O(1) = O(1) \frac{1 - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} = O(1)$  car  $\frac{1}{2^{n+1}} = o(1)$ . Ainsi  $T(2^n) = O(2^n)$  puis  $T(n) = O(n)$ .

## I.6 Nombre d'inversions

**Définition I.1** Soit  $t$  une structure séquentielle (tableau, liste, ...) contenant des valeurs comparables  $a_0, \dots, a_{n-1}$  et énumérées dans cet ordre au sein de  $t$ .

Une paire  $(i, j) \in \llbracket 0, n-1 \rrbracket^2$  où  $i < j$  est appelée une *inversion* de  $t$  lorsque  $a_i > a_j$ .

On note  $I(t)$  le nombre d'inversion de  $t$ .

**Remarque** • Le nombre d'inversions permet de mesurer à quel point  $t$  est non triée dans l'ordre croissant.

• Ce concept d'inversion est exactement celui utilisé pour les permutations en mathématiques : si  $\sigma \in \mathfrak{S}_n$ , il suffit de considérer  $(\sigma(1), \dots, \sigma(n))$ .

On cherche dans ce paragraphe à calculer  $I(t)$  efficacement. Remarquons tout d'abord qu'un algorithme naïf est en  $O(n^2)$  où  $|t| = n$  en explorant toutes les paires :

```
size_t inversions(int *t, size_t taille)
{
    size_t inv = 0;
    for (size_t i = 0; i < taille; i++)
    {
        for (size_t j = i+1; j < taille; j++)
        {
            if (t[i] > t[j]) inv++;
        }
    }
    return inv;
}
```

On va maintenant donner un algorithme type *Diviser pour régner* :

- On sépare  $t$  en deux moitiés  $t_1$  et  $t_2$ .
- On calcule  $I(t_1)$  et  $I(t_2)$  par des appels récursifs.
- On compte les inversions entre des éléments de  $t_1$  et des éléments de  $t_2$ 
  - ★ Cela ne dépend pas de leur position dans  $t_1$  ou dans  $t_2$ .
  - ★ On peut donc trier  $t_1$  en  $t'_1$  et  $t_2$  en  $t'_2$ .
  - ★ On compte  $N(t_1, t_2) = N(t'_1, t'_2)$  le nombre d'inversions entre  $t'_1$  et  $t'_2$  en  $O(n)$  par l'algorithme ci-dessous.
- On en déduit que  $I(t) = I(t_1) + I(t_2) + N(t_1, t_2)$ .

**Remarque** Pour calculer le nombre d'inversions entre deux tableaux triés  $t'_1$  et  $t'_2$ , on remarque que si  $t'_1[i] > t'_2[j]$  alors  $t'_1[i+1] \geq t'_1[i] > t'_2[j]$ . Cela signifie qu'on peut compter les inversions de manière cumulative en

avançant dans les deux tableaux en même temps :

```
OCaml
let inversions_croisees t1 t2 =
  let i = ref 0 in
  let j = ref 0 in
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let inv = ref 0 in
  let cumul = ref 0 in
  while !i < n1 && !j < n2 do
    if t1.(!i) > t2.(!j)
    then ( incr j; incr inv; incr cumul )
    else ( incr i; inv := !inv + !cumul )
  done;
  !inv + (!cumul * (n1 - !i - 1))
```

■ **Note 3** Rajouter un dessin illustrant l'aspect cumulatif. ■

Pour que cet algorithme fonctionne, il est nécessaire de trier à chaque étape les sous-tableaux, on obtient alors naïvement une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n \log n)$$

Qui se résout en  $T(n) = O(n(\log n)^2)$ . Il est possible de faire mieux en remarquant qu'on peut calculer le nombre d'inversions en décorant le tri fusion car celui fait naturellement calculer les sous-tableaux triés. On a donc une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$$

qui se résout en  $T(n) = O(n \log n)$ .

**Exercice 1** Écrire le programme réalisant cet algorithme.

■ **Preuve**

```
OCaml
let inversions_croisees t1 t2 =
  let i = ref 0 in
  let j = ref 0 in
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let inv = ref 0 in
  let cumul = ref 0 in
  while !i < n1 && !j < n2 do
    if t1.(!i) > t2.(!j)
    then ( incr j; incr inv; incr cumul )
    else ( incr i; inv := !inv + !cumul )
  done;
  !inv + (!cumul * (n1 - !i - 1))

let split t =
  let n = Array.length t in
  let m = n/2 in
  Array.sub t 0 m, Array.sub t m (n-m)

let fusion t1 t2 =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let t = Array.make (n1+n2) (t1.(0)) in
  let i = ref 0 in let j = ref 0 in
  for k = 0 to n1+n2-1 do
    if !j = n2 || (!i < n1 && t1.(!i) < t2.(!j))
    then ( t.(k) <- t1.(!i); incr i )
```

```

        else ( t.(k) <- t2.(!j); incr j )
    done;
    t

let rec inversions_div t =
  let n = Array.length t in
  if n = 1
  then t, 0
  else let t1, t2 = split t in
        let t1', n1 = inversions_div t1 in
        let t2', n2 = inversions_div t2 in
        let n12 = inversions_croisees t1' t2' in
        fusion t1' t2', n1+n2+n12

```

## I.7 Points les plus proches

On considère le problème

### Problème - PLUSPROCHEPAIRE

- Entrées :  
Un ensemble  $P$  de  $n$  points dans le plan.
- Sortie :  
Une paire  $\{p, p'\}$  de points telle que  $\text{dist}(p, p') = \|\vec{pp'}\|$  soit minimale.

Ce problème a déjà été étudié dans le chapitre Recherche par force brute où l'algorithme naïf en  $O(n^2)$  a été amélioré en un algorithme en  $O(n \log n)$ .

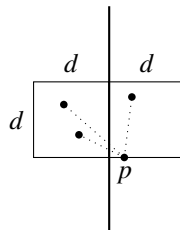
On présente ici un algorithme diviser pour régner.

Si  $n > 1$  l'idée est de séparer les points en fonctions de la médiane des abscisses. C'est-à-dire de déterminer tel  $x$ , par exemple en prenant la moitié entre l'abscisse médiane et l'abscisse suivante, tel que  $\lceil n/2 \rceil$  points soient d'abscisses  $< x$ , notons  $S_1$  leur ensemble, et  $\lfloor n/2 \rfloor$  soient d'abscisses  $> x$ , notons  $S_2$ . Ensuite, on applique l'algorithme récursivement pour obtenir  $d_1$  la plus petite distance dans  $S_1$  et  $d_2$  la plus petite distance dans  $S_2$ . On pose  $d = \min(d_1, d_2)$  et la question qui se pose est celle de calculer  $d' = \min\{\text{dist}(p_1, p_2) \mid p_1 \in S_1, p_2 \in S_2\}$ .

Comme on a déjà déterminé  $d$ , on peut se contenter de chercher à déterminer s'il existe  $d' < d$ . Pour cela, on peut se contenter de ne considérer dans  $S_1$  que les points d'abscisse dans  $]x - d; x]$  et pour  $S_2$  dans  $[x; x + d[$ . On a ainsi une bande à étudier. Comme il n'est pas possible d'exclure qu'un nombre important de points soit dans cette bande, on a *a priori* une fusion en  $O(n)$ .

On va considérer les points de bas en haut dans la bande, par exemple avec un tri initial selon les ordonnées. Ainsi, il sera inutile, quand on considère un point  $p$  d'ordonnée  $y$  de considérer des points d'ordonnées  $< y$ . D'un autre côté, comme on cherche l'existence d'une distance  $< d$ , il est inutile de considérer des points d'ordonnée  $> y + d$ .

On se retrouve alors à comparer les points présents dans l'union de deux carrés de côté  $d$  de part et d'autres de la séparation :

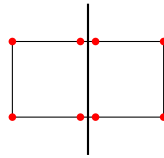


Cependant, on sait que chaque carré étant situé d'un même côté de la ligne médiane de séparation, les points qui s'y trouvent sont à distance  $\geq d$ . Il est facile de se convaincre qu'il n'est pas possible de placer plus que quatre points dans un carré de côté  $d$  de sorte qu'aucune paire soit à distance  $< d$ . Cette configuration correspond ainsi à placer les points aux sommets.

On en déduit donc qu'il y a maximum quatre points dans le carré gauche et quatre points dans le carré droit, dont le point  $p$  considéré : il n'est pas nécessaire d'explorer plus que les sept points suivants.

Sur le schéma suivant, on a volontairement séparé les points autour de la frontière pour marquer cette différence.





**Remarque** Cette analyse est très naïve, on peut démontrer qu'il suffit d'observer cinq autres points.

OCaml

```

let distance (x1,y1) (x2,y2) =
  let v = (x1-.x2)*.(x1-.x2)+.(y1-.y2)*.(y1-.y2) in
  sqrt v

let closest_brute points i j =
  let d = ref (distance points.(i) points.(i+1)) in
  for k = i to j do
    for l = k+1 to j do
      let dist_kl = distance points.(k) points.(l) in
      if !d > dist_kl
      then d := dist_kl
    done
  done;
  !d

let rec closest_rec p_x p_y i j =
  if j-i+1 <= 3
  then closest_brute p_x i j
  else begin
    let m = (i+j) / 2 in
    let d1 = closest_rec p_x p_y i m in
    let d2 = closest_rec p_x p_y (m+1) j in
    let d = min d1 d2 in
    let min_x = fst p_x.(m) -. d in
    let max_x = fst p_x.(m+1) +. d in

    let bande = array_filter
      (fun (x,_) -> min_x <= x && x <= max_x) p_y in
    let m_d = ref d in
    for k = 0 to Array.length bande - 1 do
      for l = k+1 to min (k+7) (Array.length bande - 1) do
        let d_kl = distance bande.(k) bande.(l) in
        if d_kl < !m_d
        then m_d := d_kl
      done
    done;
    !m_d
  end

let closest points =
  let p_y = Array.copy points in
  Array.sort Stdlib.compare points;
  Array.sort Stdlib.compare p_y;
  closest_rec points p_y 0 (Array.length points - 1)

```

**C** ERROR: src/algorithmique/../../snippets/algorithmique/closest\_div.c does not exist

Python

```

def distance(p1,p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt( (x1-x2)**2 + (y1-y2)**2 )

def closest_brute(points, i, j):
    m = distance(points[i], points[i+1])

```

```

    for k in range(i,j+1):
        for l in range(k+1,j+1):
            d = distance(points[k], points[l])
            if d < m:
                m = d
    return m

def closest_rec(points_x, points_y, i, j):
    if j-i+1 <= 3:
        return closest_brute(points_x, i, j)
    m = (i+j)//2
    d1 = closest_rec(points_x, points_y, i, m)
    d2 = closest_rec(points_x, points_y, m+1, j)
    d = min(d1, d2)
    min_x = points_x[m][0]-d
    max_x = points_x[m+1][0]+d
    bande = [ p for p in points_y if min_x <= p[0] <= max_x ]
    for k in range(len(bande)):
        for l in range(1,8):
            if k+l >= len(bande):
                break
            d_kl = distance(bande[k], bande[k+l])
            if d_kl < d:
                d = d_kl
    return d

def closest(points):
    n = len(points)
    points.sort(key=lambda p: p[0])
    points_y = points[:]
    points_y.sort(key=lambda p: p[1])
    return closest_rec(points, points_y, 0, n-1)

```

## II Meet in the middle

### II.1 Principe

### II.2 Sous-ensemble de somme donnée

## III Dichotomie pour passer de décision à optimisation

### III.1 Principe

### III.2 Couverture par des segments égaux

■ **Note 4** Il y a une confusion réel/entiers ici. À reprendre.

■

On considère ici  $n$  points sur la droite réelle. Le  $i$ -ème point est identifié par sa coordonnée  $x_i$ . On se pose alors, dans un premier temps, la question de savoir si on peut trouver  $k$  segments de longueur  $l$  tels que chaque point appartienne à au moins un de ces segments. Dans un second temps, on se posera la question de la longueur  $l$  minimale de ces segments.

#### III.2.i Couverture par des segments de longueur donnée

On va ici résoudre le premier problème :

##### Problème - EXISTENCECOUVERTURESEGMENT

- Entrées :
  - ★  $n$  points sur la droite réelle  $x_1, \dots, x_n$
  - ★ un entier  $k \geq 1$
  - ★ un réel  $l$
- Sortie :
 

existe-t-il  $k$  segments  $S_1, \dots, S_k$  de longueur  $l$  tels que  $\forall i \in \llbracket 1, n \rrbracket, \exists j \in \llbracket 1, k \rrbracket, x_i \in S_j$  ?

On remarque qu'un segment de longueur  $l$  est uniquement caractérisé par son extrémité gauche. Pour chaque  $x_i$ , il doit ainsi exister une extrémité gauche dans le segment  $S_i = [x_i - l, x_i]$ . On peut ainsi renverser le problème et en faire un problème de couverture de segments par des points : on cherche  $k$  points tel que chaque segment  $S_i$  contienne au moins un de ces points.

**Problème - ENSEMBLEINTERSECTANTLIGNE**

- Entrées :
  - ★  $n$  segments  $S_i = [l_i, r_i]$
  - ★ un entier  $k \geq 1$ .
- Sortie :  
Un ensemble  $P$  de  $k$  points tels que  $\forall p \in P, \exists i \in \llbracket 1, n \rrbracket, p \in S_i$  en cas de succès

Ce problème peut se résoudre par un algorithme glouton :

- on commence avec  $P = \emptyset$
  - on trie les segments par  $r_i$  croissant
  - pour chaque segment  $S_i = [l_i, r_i]$ , s'il ne contient aucun élément de  $P$ , on rajoute  $r_i$  à  $P$ .
  - on répond avec un succès si  $|P| \leq k$  (on peut alors compléter avec des points quelconques pour avoir exactement  $k$  points).
- Cet algorithme est en  $O(n \log n)$  en raison du tri initial.

### III.2.ii Longueur minimale

On considère maintenant le problème suivant :

**Problème - COUVERTURESEGMENTSMINIMALE**

- Entrées :
  - ★  $n$  points à coordonnées entières sur la droite réelle  $x_1, \dots, x_n$
  - ★ un entier  $k \geq 1$
- Sortie :  
la longueur  $l$  minimale pour laquelle il existe une couverture des points par  $k$  segments de longueur  $l$ .

**On peut résoudre ce problème par dichotomie à l'aide de l'algorithme précédent**

- on considère  $L = \frac{\max_{1 \leq i < j \leq n} |x_j - x_i|}{k} = \frac{D}{k}$  où le diamètre  $D$  se calcule en  $O(n)$  et correspond à répartir de manière uniforme les segments pour couvrir les points. Une telle couverture existe toujours.
- on peut considérer, sans avoir besoin de le calculer au préalable, le tableau  $V$  de booléen de longueur  $L + 1$  tel que  $V[l]$  indique s'il est possible de couvrir les points par  $k$  segments de longueur  $l$ . Pour obtenir la valeur  $V[l]$  il suffit d'appliquer l'algorithme précédent.
- on effectue alors une recherche dichotomique du plus petit indice  $l$  tel que  $V[l]$  soit vrai.

On obtient ainsi un algorithme en  $O(n \log n \log \frac{D}{k})$  qu'on peut considérer comme étant en  $O(n \log n)$  en supposant que  $D$  est une constante.

Cette dichotomie est en fait une instance d'un principe fondamental permettant de transformer un problème de décision (existe-t-il?) en un problème d'optimisation (quelle est ... minimal/maximal?).

### III.2.iii Implémentation

## IV Problèmes

### IV.1 Multiplication d'entiers

On considère ici le problème de la multiplication de deux entiers données en précision arbitraire par le tableau de leurs chiffres.

Ainsi, l'entier  $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$  sera représenté par le tableau  $[|3;2;1|]$  de telle sorte que le nombre représenté par  $t$  soit  $\sum_{k=0}^{n-1} t[k]10^k$  où  $n = |t|$  la taille de  $t$ . On suppose que  $t$  ne finit pas par des 0. Autrement dit : soit  $t$  est vide, soit la dernière valeur de  $t$  est non nulle.

On se convaincra aisément que les entiers naturels sont en bijection avec de telles représentations.

**Question IV.1** Écrire une fonction `simpl : int array -> int array` qui renvoie un nouveau tableau obtenu en supprimant tous les 0 en fin de tableau : `simpl [|1;2;3;0;0|] = [|1;2;3|]`.

**Question IV.2** Écrire une fonction `add : int array -> int array -> int array` qui calcule la somme de deux entiers en temps linéaire.

On va étudier ici plusieurs techniques de multiplication.

#### IV.1.i Multiplication naïve

On considère la multiplication posée comme étudiée à l'école primaire :

$$\begin{array}{r} 123 \\ \times 42 \\ \hline 246 \\ 492 \\ \hline 5166 \end{array}$$

Plus précisément, si  $y = \sum_{j=0}^m y_j 10^j$ , on a

$$x \times y = \sum_{j=0}^m (y_j \times x) 10^j$$

**Question IV.3** Écrire une fonction `mult_digit : int -> int array -> int array` telle que `mult_digit a x` renvoie  $a \times x$  où  $a$  est un chiffre et  $x$  un nombre.

**Question IV.4** Écrire une fonction `mult_dec : int -> int array -> int array` telle que `mult_dec p x` renvoie  $x 10^p$  où  $p$  est un entier et  $x$  un nombre.

**Question IV.5** Écrire une fonction `mult_naive : int array -> int array -> int array` réalisant cette multiplication.

**Question IV.6** Quelle est la complexité de cette méthode en fonction de la longueur des entrées ?

#### IV.1.ii Première approche diviser pour régner

On remarque que pour  $0 \leq a, b, c, d < 10^m$  :

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (ad + bc) + bd$$

On peut alors calculer récursivement les quatre produits  $ac, ad, bc$  et  $bd$ .

On en déduit donc une solution type **diviser pour régner** de la multiplication :

- $mul(x, y, 1) = x \times y$
  - pour  $n \geq 2$ ,  $mul(x, y, n) = 10^{2m} mul(a, c, m) + 10^m (mul(a, d, m) + mul(b, c, m)) + mul(b, d, m)$  où  $x = a 10^m + b$ ,  $y = c 10^m + d$  et  $m = \lceil n/2 \rceil$ .
- où  $x, y \leq 10^n$ .

**Question IV.7** Déterminer avec précision la complexité  $T(n)$  de  $mul(x, y, n)$ .  
Que peut-on en conclure ?

#### IV.1.iii Algorithme de Karatsuba

Face à ce problème, Karatsuba, alors âgé de 23 ans, est parti de la remarque suivante :

$$ac + bd - (a - b)(c - d) = ad + bc$$

qui permet de voir qu'en calculant  $ac, bd$  et le produit  $(a - b)(c - d)$  on peut en déduire le coefficient  $ad + bc$  de  $10^m$ , et ainsi économiser une multiplication.

On en déduit ainsi une nouvelle solution pour la multiplication :

- $kar(x, y, 1) = x \times y$
- pour  $n \geq 2$ ,  $kar(x, y, n) = 10^{2m}X + 10^m(X + Y - Z) + Y$  où  $x = a10^m + b$ ,  $y = c10^m + d$ ,  $m = \lceil n/2 \rceil$ ,  $X = kar(a, c, m)$ ,  $y = kar(b, d, m)$  et  $Z = kar(a - b, c - d, m)$ .

La présence de nombres négatifs ne complique pas la complexité car il suffit de rajouter un booléen indiquant le le signe à côté du tableau des chiffres d'un nombre.

**Question IV.8** Déterminer avec précision la complexité  $K(n)$  de  $kar(x, y, n)$ .