Algorithmique et complexité des problèmes

Xavier Goaoc & Antoine Meyer

Laboratoire d'informatique Gaspard Monge

Algorithme, modèle de calcul, complexité

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre* x.
- *Sortie :* 1 *si* x *est dans le tableau,* 0 *sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72
x	41										

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie :* 1 *si x est dans le tableau,* 0 *sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

1	2	3	4	5	6	7	8	9	10	11

x 41 deb fin

mil

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 1 fin 11

mil

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 1 fin 11

mil 6

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x = 41

1	2	3	4	5	6	7	8	9	10	11
					27					

x = 41 deb = 1 fin = 11

mil = 6

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
    $mil = (deb + fin)/2$
    si $x < tab[mil]$
        $fin \leftarrow mil$
    sinon
        $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x = 41

1	2	3	4	5	6	7	8	9	10	11
					27					

x = 41 deb = 6 fin = 11

mil = 6

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
    $mil = (deb + fin)/2$
    si $x < tab[mil]$
        $fin \leftarrow mil$
    sinon
        $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
  $mil = (deb + fin)/2$
  si $x < tab[mil]$
    $fin \leftarrow mil$
  sinon
    $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 6 fin 11

mil 6

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 6 fin 11

mil 8

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
    $mil = (deb + fin)/2$
    si $x < tab[mil]$
        $fin \leftarrow mil$
    sinon
        $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x : 41

1	2	3	4	5	6	7	8	9	10	11
							35			

x : 41 deb : 6 fin : 11

mil : 8

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
  $mil = (deb + fin)/2$
  si $x < tab[mil]$
    $fin \leftarrow mil$
  sinon
    $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
    $mil = (deb + fin)/2$
    si $x < tab[mil]$
        $fin \leftarrow mil$
    sinon
        $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 8 fin 11

mil 8

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

1	2	3	4	5	6	7	8	9	10	11

x 41 deb 8 fin 11

mil 9

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x = 41

1	2	3	4	5	6	7	8	9	10	11
								42		

x = 41 deb = 8 fin = 11

mil = 9

$deb = 1$
$fin = 11$
tant que $(fin - deb) > 1$
  $mil = (deb + fin)/2$
  si $x < tab[mil]$
    $fin \leftarrow mil$
  sinon
    $debut \leftarrow mil$

si $tab[mil] = x$ répondre 1
sinon répondre 0

Un exemple de problème algorithmique : la **recherche dichotomique**.

- *Entrée : un tableau de nombres, triés (dans l'ordre croissant) et un nombre x.*
- *Sortie : 1 si x est dans le tableau, 0 sinon.*

i	1	2	3	4	5	6	7	8	9	10	11
tab[i]	5	9	12	23	24	27	29	35	42	51	72

x 41

1	2	3	4	5	6	7	8	9	10	11
								42		

x 41 deb 8 fin 9

mil 9

```
deb = 1
fin = 11
tant que (fin − deb) > 1
    mil = (deb + fin)/2
    si x < tab[mil]
        fin ← mil
    sinon
        debut ← mil

si tab[mil] = x répondre 1
sinon répondre 0
```

Comment mesurer l'**efficacité** de la recherche dichotomique ?

Est-ce que toute opération a le même coût ?
Mesure indépendante du matériel ?
Dépendance à l'entrée ?

Comment mesurer l'**efficacité** de la recherche dichotomique ?

Est-ce que toute opération a le même coût ?
Mesure indépendante du matériel ?
Dépendance à l'entrée ?

L'analyse de complexité d'un algorithme propose généralement
de considérer le traitement d'une entrée de taille n,
de s'intéresser au comportement asymptotique pour $n \to \infty$,
de la quantité maximum de ressources utilisées pour son traitement.

Comment mesurer l'**efficacité** de la recherche dichotomique ?

Est-ce que toute opération a le même coût ?
Mesure indépendante du matériel ?
Dépendance à l'entrée ?

L'analyse de complexité d'un algorithme propose généralement
de considérer le traitement d'une entrée de taille n,
de s'intéresser au comportement asymptotique pour $n \to \infty$,
de la quantité maximum de ressources utilisées pour son traitement.

Une recherche dichotomique dans un tableau trié de taille n prend $O(\log_2 n)$ opérations.

*Chaque itération de la boucle **tant que** divise par 2 la taille*
de la partie du tableau restant à examiner.
Les erreurs d'arrondis disparaissent dans le $O()$.

Comment mesurer l'**efficacité** de la recherche dichotomique ?

Est-ce que toute opération a le même coût ?
Mesure indépendante du matériel ?
Dépendance à l'entrée ?

L'analyse de complexité d'un algorithme propose généralement
de considérer le traitement d'une entrée de taille n,
de s'intéresser au comportement asymptotique pour $n \to \infty$,
de la quantité maximum de ressources utilisées pour son traitement.

Une recherche dichotomique dans un tableau trié de taille n prend $O(\log_2 n)$ opérations.

Chaque itération de la boucle ***tant que*** *divise par* 2 *la taille*
de la partie du tableau restant à examiner.
Les erreurs d'arrondis disparaissent dans le $O()$.

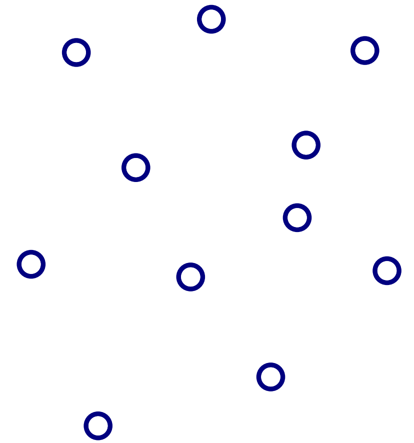
Requiert de formaliser un **modèle de calcul** et la ressource considérée.

Ex. : au plus deux cartes visibles, nombre de retournements.
Machine de Turing, modèle RAM, automates...
Instructions, espace mémoire, violations de cache...

Pourquoi réfléchir ?

C’est fatigant, les machines sont de plus en plus rapides, etc.

Parce que certains algorithmes passent à l’échelle, d’autre non.



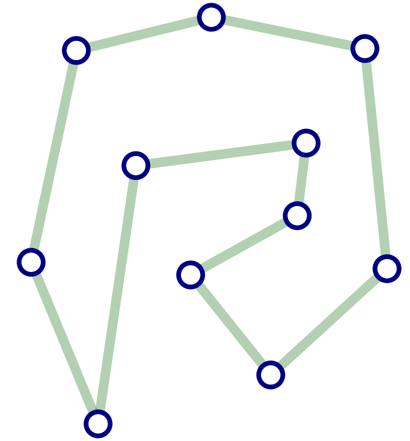
Pourquoi réfléchir ?

C'est fatigant, les machines sont de plus en plus rapides, etc.

Parce que certains algorithmes passent à l'échelle, d'autre non.

Exemple sur le problème du **voyageur de commerce**.

- *Entrée : une liste de n villes et des distances entre elles.*
- *Sortie : un circuit visitant chaque ville une fois et minimisant le trajet total.*



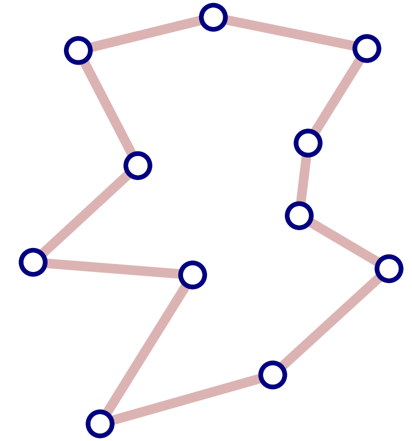
Pourquoi réfléchir ?

C'est fatigant, les machines sont de plus en plus rapides, etc.

Parce que certains algorithmes passent à l'échelle, d'autre non.

Exemple sur le problème du **voyageur de commerce**.

- *Entrée : une liste de n villes et des distances entre elles.*
- *Sortie : un circuit visitant chaque ville une fois et minimisant le trajet total.*



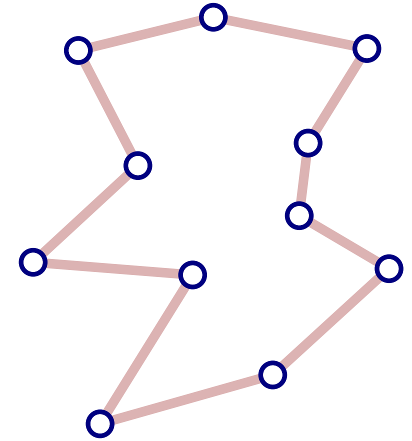
Pourquoi réfléchir ?

C’est fatigant, les machines sont de plus en plus rapides, etc.

Parce que certains algorithmes passent à l’échelle, d’autre non.

Exemple sur le problème du **voyageur de commerce**.

- *Entrée : une liste de n villes et des distances entre elles.*
- *Sortie : un circuit visitant chaque ville une fois et minimisant le trajet total.*



Pourquoi réfléchir ?

C'est fatigant, les machines sont de plus en plus rapides, etc.

Parce que certains algorithmes passent à l'échelle, d'autre non.

Exemple sur le problème du **voyageur de commerce**.

- *Entrée : une liste de n villes et des distances entre elles.*
- *Sortie : un circuit visitant chaque ville une fois et minimisant le trajet total.*

On peut faire cela "facilement" en examinant chacun des $n!$ ordres à tour de rôle...

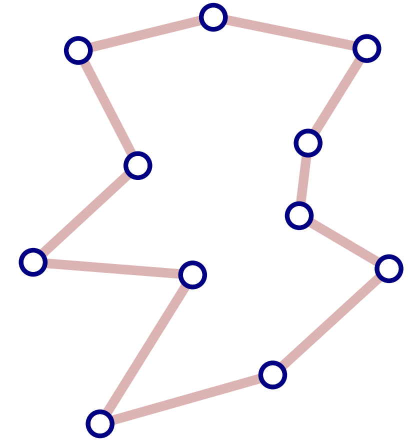
... mais une machine traitant 10^{15} ordres par seconde...

génération de la permutation, calcul de longueur, comparaison...

une hypothèse très optimiste...

Pourquoi réfléchir ?

C'est fatigant, les machines sont de plus en plus rapides, etc.



Parce que certains algorithmes passent à l'échelle, d'autre non.

Exemple sur le problème du **voyageur de commerce**.

- *Entrée : une liste de n villes et des distances entre elles.*
- *Sortie : un circuit visitant chaque ville une fois et minimisant le trajet total.*

On peut faire cela "facilement" en examinant chacun des $n!$ ordres à tour de rôle...

... mais une machine traitant 10^{15} ordres par seconde...

génération de la permutation, calcul de longueur, comparaison...

une hypothèse très optimiste...

... mettra...

n	20	22	24	25	30
temps	1h	2 semaines	20 ans	500 ans	trop$^{(*)}$

$(*)$ après

Le problème **3-SUM**

Problème **3-SUM** :

- *Entrée :* n *entiers relatifs* $a_1, a_2, \ldots, a_n \in \mathbb{Z}$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* $a_i + a_j + a_k = 0$.

$1, 5, -2, 8, 3, -4, -1, 12$ *réponse :* 1

$5, -2, 9, -6, 4, 3, -35, 42$ *réponse :* 0

Problème **3-SUM** :

- *Entrée :* n *entiers relatifs* $a_1, a_2, \ldots, a_n \in \mathbb{Z}$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* $a_i + a_j + a_k = 0$.

$1, 5, -2, 8, 3, -4, -1, 12$ *réponse :* 1

$5, -2, 9, -6, 4, 3, -35, 42$ *réponse :* 0

Algorithme naïf : examiner chacun des triplets à tour de rôle.

Pour chaque triplet, calculer sa somme.

Si une des sommes est nulle, répondre 1*, sinon répondre* 0*.*

Problème **3-SUM** :

- *Entrée :* n *entiers relatifs* $a_1, a_2, \ldots, a_n \in \mathbb{Z}$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* $a_i + a_j + a_k = 0$.

$1, 5, -2, 8, 3, -4, -1, 12$ *réponse :* 1

$5, -2, 9, -6, 4, 3, -35, 42$ *réponse :* 0

Algorithme naïf : examiner chacun des triplets à tour de rôle.

Pour chaque triplet, calculer sa somme.

Si une des sommes est nulle, répondre 1*, sinon répondre* 0.

La **complexité en temps** de cet algorithme est $\Theta(n^3)$.

Spécifier l'algorithme signifie décrire l'ordre d'examen des triplets.

Examiner un triplet prend un temps constant, examiner les $\binom{n}{3}$ *triplets suffit.*

Pour tout ordre d'examen, il existe une entrée pour laquelle il faut examiner les $\binom{n}{3}$ *triplets.*

Problème **3-SUM** :

- *Entrée :* n *entiers relatifs* $a_1, a_2, \ldots, a_n \in \mathbb{Z}$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* $a_i + a_j + a_k = 0$.

$1, 5, -2, 8, 3, -4, -1, 12$ *réponse :* 1

$5, -2, 9, -6, 4, 3, -35, 42$ *réponse :* 0

Algorithme naïf : examiner chacun des triplets à tour de rôle.

Pour chaque triplet, calculer sa somme.

Si une des sommes est nulle, répondre 1*, sinon répondre* 0.

La **complexité en temps** de cet algorithme est $\Theta(n^3)$.

Spécifier l'algorithme signifie décrire l'ordre d'examen des triplets.

Examiner un triplet prend un temps constant, examiner les $\binom{n}{3}$ *triplets suffit.*

Pour tout ordre d'examen, il existe une entrée pour laquelle il faut examiner les $\binom{n}{3}$ *triplets.*

Peut-on faire mieux ?

Les propriétés de l’**ordre** peuvent permettre d’économiser des comparaisons.

Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

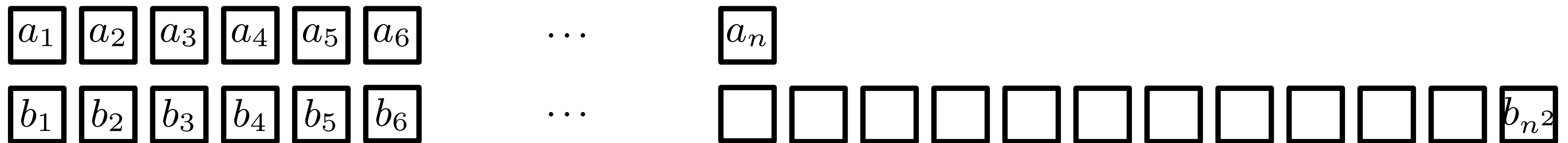
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

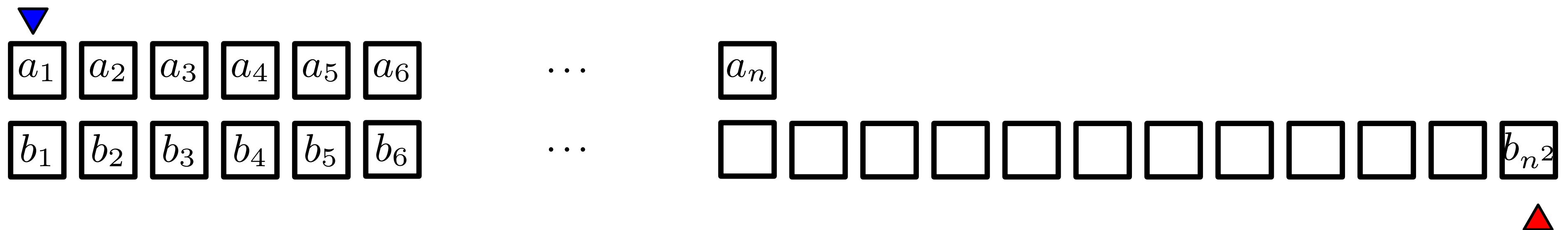
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

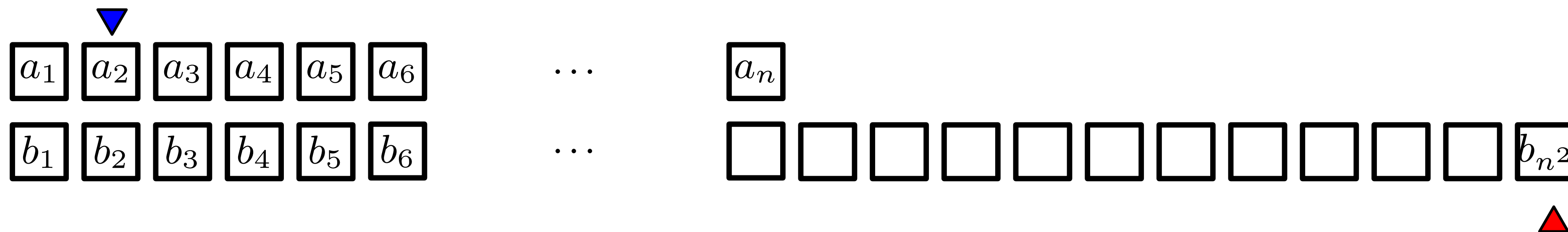
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

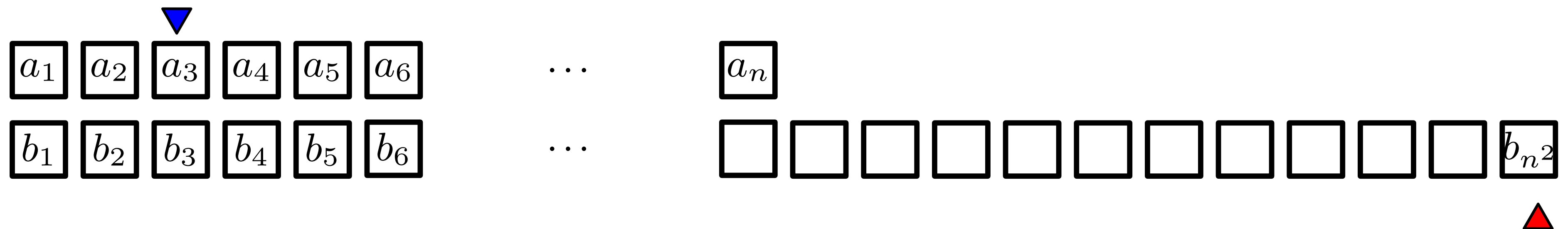
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

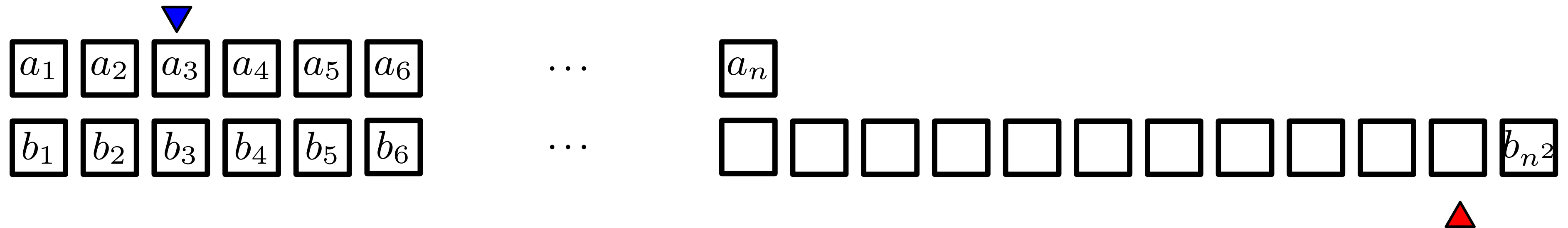
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



Les propriétés de l'**ordre** peuvent permettre d'économiser des comparaisons.

Supposons $a_{i'} > a_i$ *et* $a_{j'} < a_j$

Si $a_i + a_j + a_k > 0$ *alors* $a_{i'} + a_j + a_k > 0$

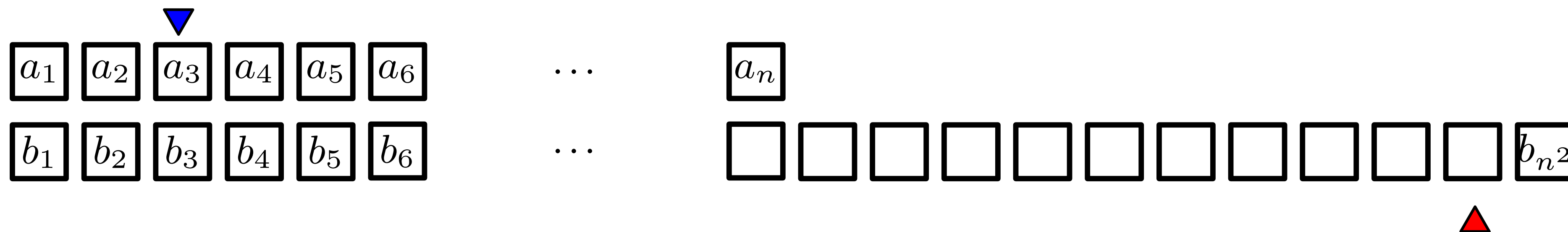
Si $a_i + a_j + a_k < 0$ *alors* $a_i + a_{j'} + a_k < 0$

Nouvel algorithme :

Pour $1 \leq i, j \leq n$ *on définit* $b_{n(i-1)+j} = a_i + a_j$.

On trie chacune des listes $a_1, a_2, \ldots, a_n$ *et* $b_1, b_2, \ldots, b_{n^2}$ *dans l'ordre croissant.*

On marche simultanément sur les deux listes pour chercher i *et* j *tels que* $a_i + b_j = 0$.



La **complexité en temps** de cet algorithme est $O(n^2 \log n)$.

Trier k *entiers peut se faire en* $O(k \log k)$.

Chaque pas de la marche se fait au moyen d'un nombre constant d'opérations.

Le nombre total de pas est au plus $n^2 + n$.

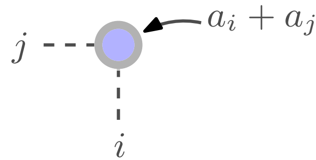
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

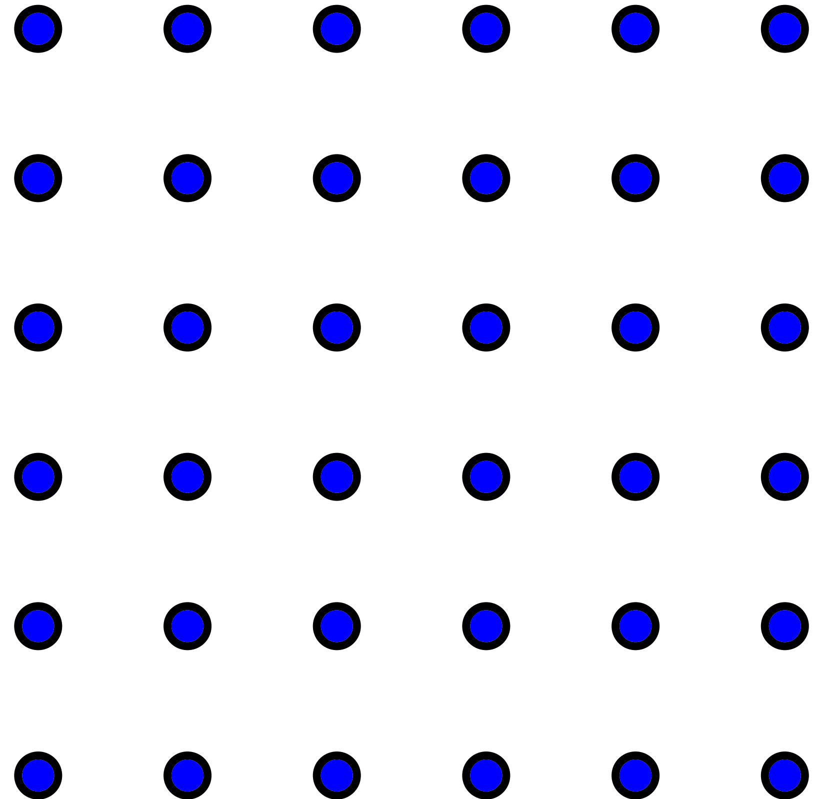
On trie $a_1, a_2, \ldots, a_n$ *par ordre croissant.*

Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i,j] = a_i + a_j$.

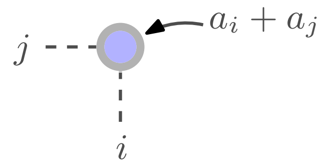




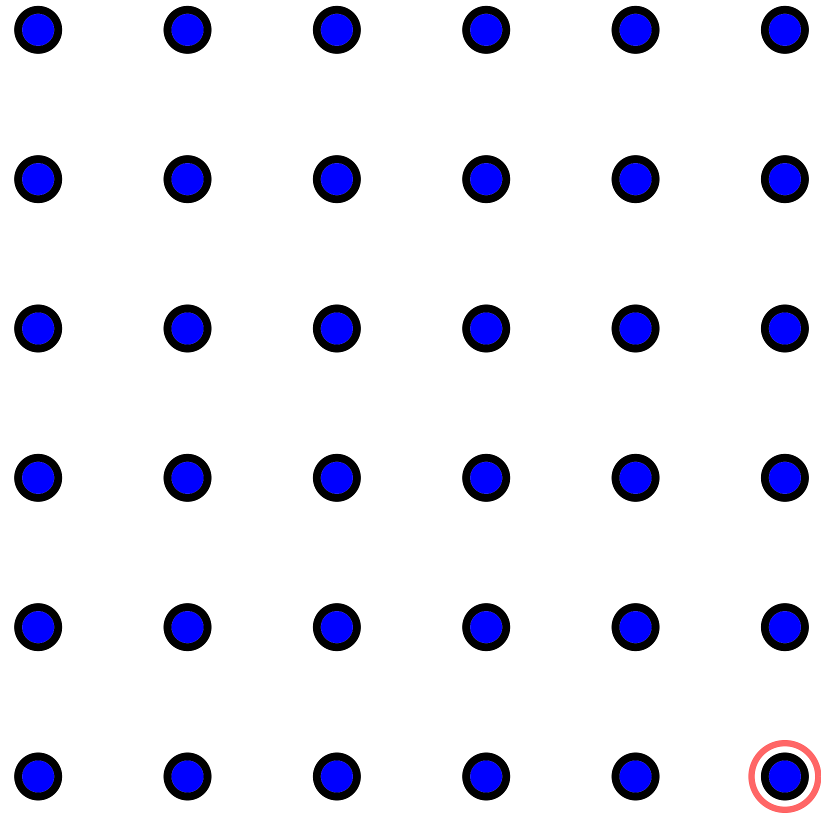
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i, j] = a_i + a_j$.



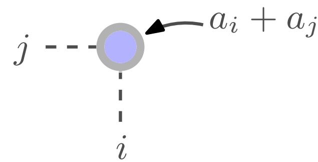
On décide en temps $O(n)$ si le tableau contient une valeur x par marche.



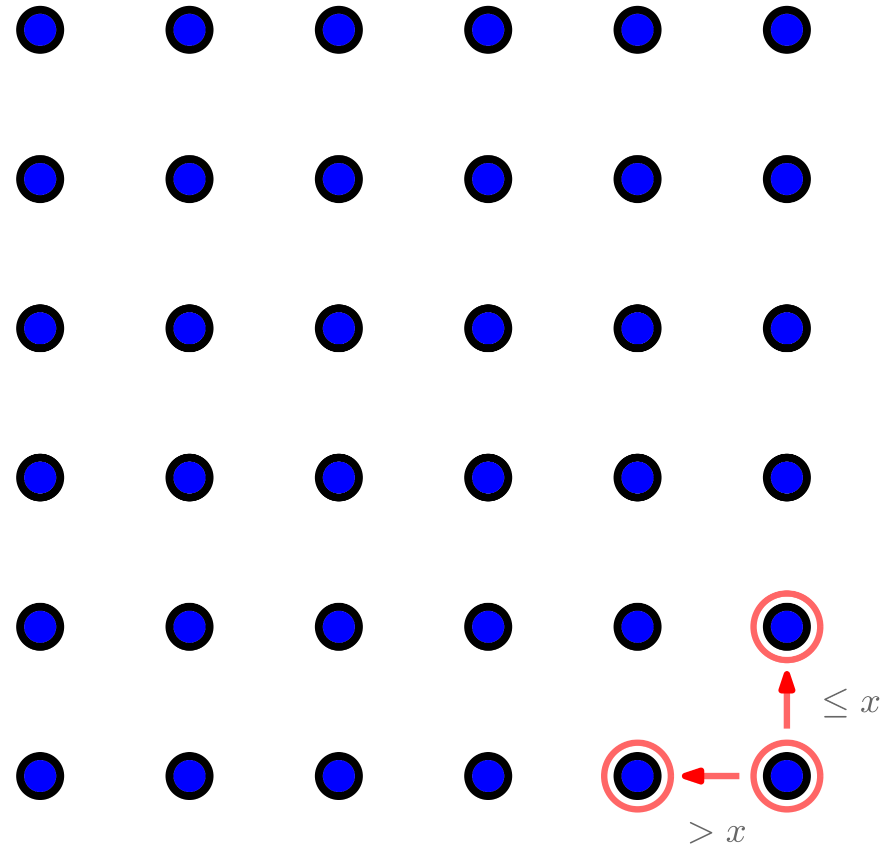
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i,j] = a_i + a_j$.



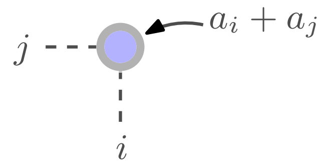
On décide en temps $O(n)$ si le tableau contient une valeur x par marche.



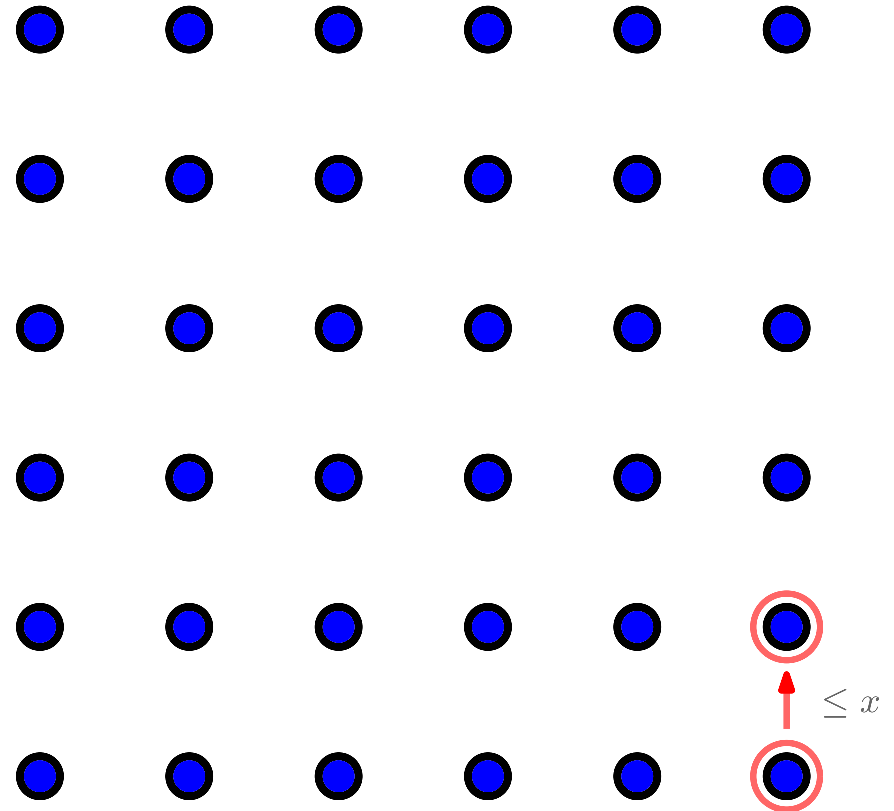
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i,j] = a_i + a_j$.



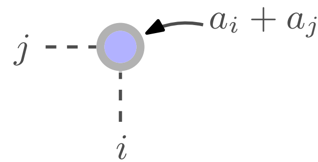
On décide en temps $O(n)$ si le tableau contient une valeur x par marche.



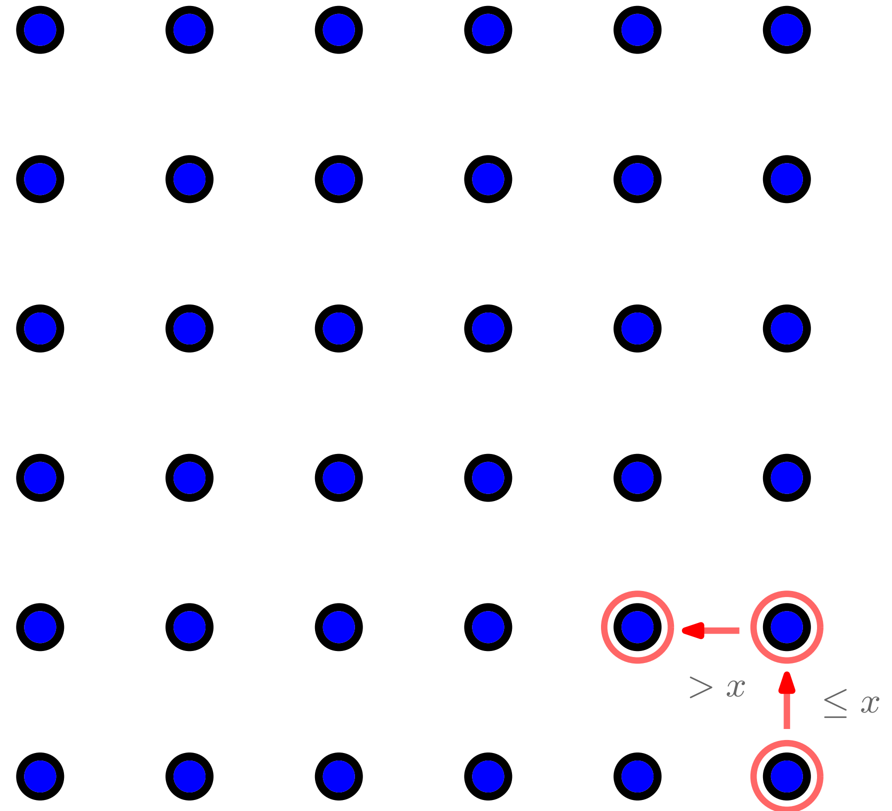
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i, j] = a_i + a_j$.



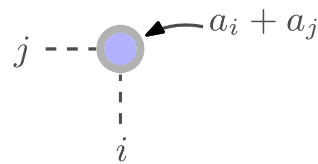
On décide en temps $O(n)$ si le tableau contient une valeur x par marche.



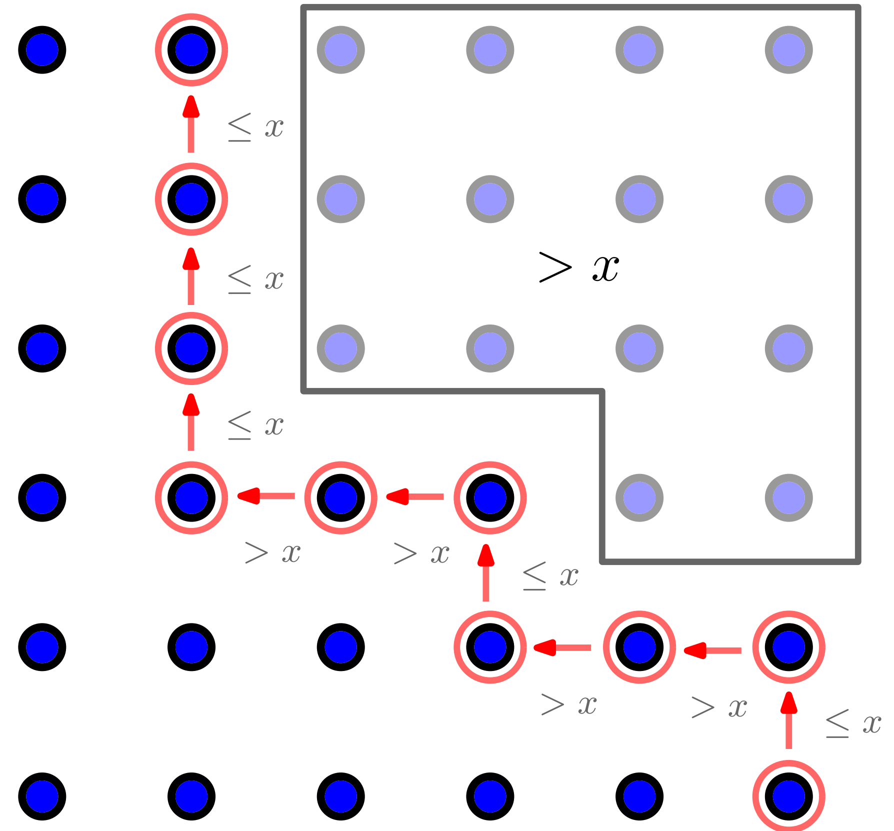
Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i,j] = a_i + a_j$.



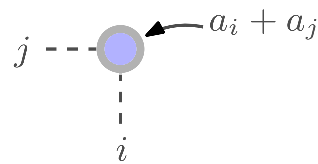
On décide en temps $O(n)$ si le tableau contient une valeur x par marche.



Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

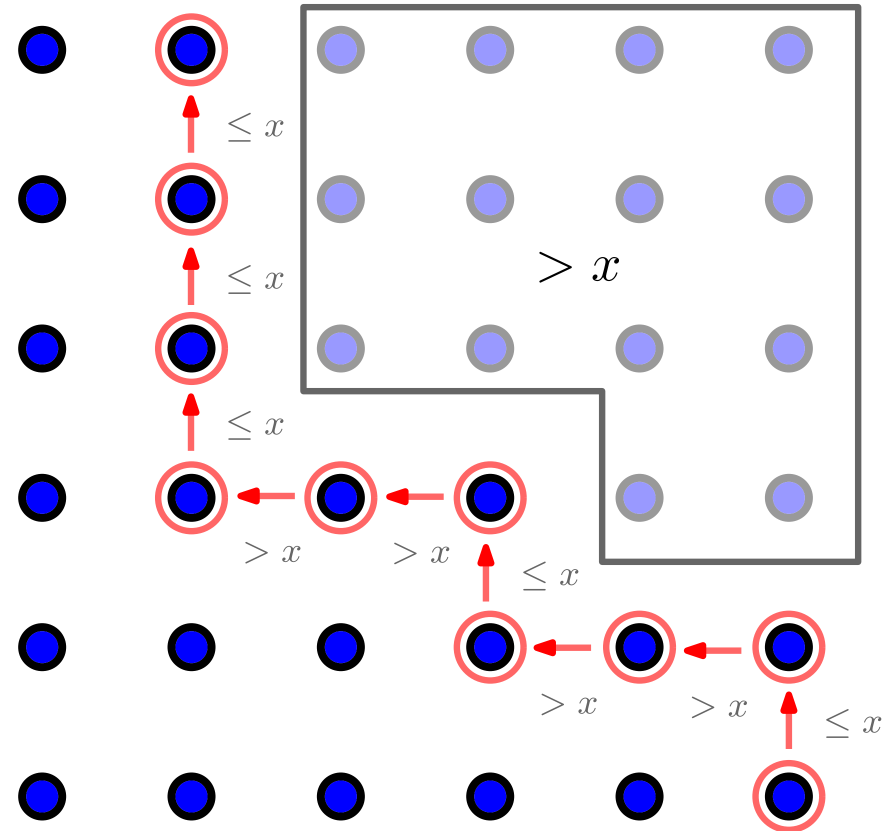
On construit un tableau $c[i,j] = a_i + a_j$.



On décide en temps $O(n)$ si le tableau contient une valeur x par marche.

Pour résoudre 3-SUM il suffit de chercher

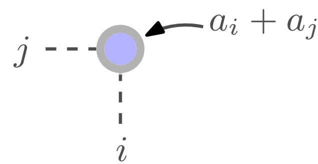
$$x = -a_1, x = -a_2, \ldots, x = -a_n$$

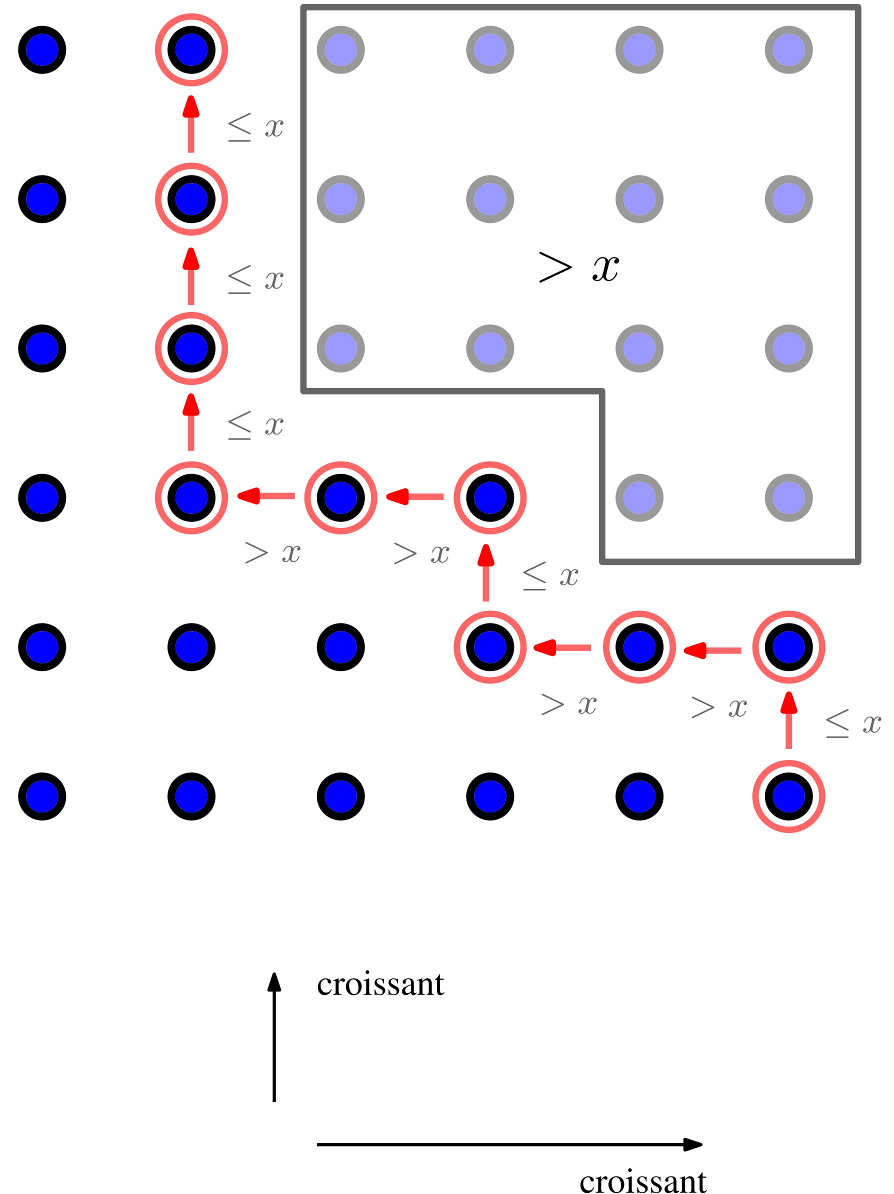


Raffiner cette idée conduit à un algorithme de complexité $O(n^2)$.

On trie $a_1, a_2, \ldots, a_n$ par ordre croissant.

On construit un tableau $c[i, j] = a_i + a_j$.





On décide en temps $O(n)$ si le tableau contient une valeur x par marche.

Pour résoudre 3-SUM il suffit de chercher

$$x = -a_1, x = -a_2, \ldots, x = -a_n$$

Complexité :

Le tri initial prends un temps $O(n \log n)$.

Construire le tableau prends un temps $O(n^2)$.

Chaque marche prends un temps $O(n)$.

L'ensemble des marches prends un temps $O(n^2)$.

On a vu trois algorithmes pour 3-SUM de complexités $O(n^3)$, $O(n^2 \log n)$ et $O(n^2)$.

On a vu trois algorithmes pour 3-SUM de complexités $O(n^3)$, $O(n^2 \log n)$ et $O(n^2)$.

$$\text{Complexité d'un algorithme} \quad \rightsquigarrow \quad \text{complexité d'un problème}$$

Quelle est la complexité **minimale** d'un algorithme résolvant 3-SUM ?

Peut on envisager $O(n \log n)$ *?* $O(n\sqrt{n})$ *?* $O(\sqrt{n})$ *?* $O(1)$ *?*

On a vu trois algorithmes pour 3-SUM de complexités $O(n^3)$, $O(n^2 \log n)$ et $O(n^2)$.

Complexité d'un algorithme $\rightsquigarrow$ complexité d'un problème

Quelle est la complexité **minimale** d'un algorithme résolvant 3-SUM ?

Peut on envisager $O(n \log n)$ *?* $O(n\sqrt{n})$ *?* $O(\sqrt{n})$ *?* $O(1)$ *?*

Conjecture (1995) : Tout algorithme pour 3-SUM a complexité $\Omega(n^2)$.

On a vu trois algorithmes pour 3-SUM de complexités $O(n^3)$, $O(n^2 \log n)$ et $O(n^2)$.

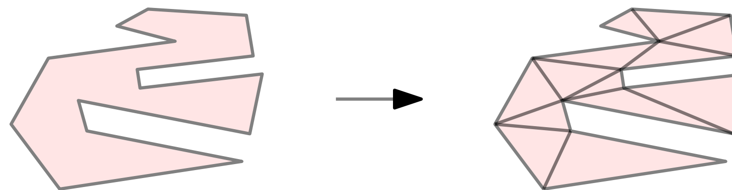
Complexité d'un algorithme $\rightsquigarrow$ complexité d'un problème

Quelle est la complexité **minimale** d'un algorithme résolvant 3-SUM ?

Peut on envisager $O(n \log n)$ *?* $O(n\sqrt{n})$ *?* $O(\sqrt{n})$ *?* $O(1)$ *?*

Conjecture (1995) : Tout algorithme pour 3-SUM a complexité $\Omega(n^2)$.

Les bornes supérieures sont généralement obtenues par des algorithmes explicites.



$O(n \log n)$ *en 1978*
$O(n \log \log n)$ *en 1986*
$O(n \log^* n)$ *en 1989*
$\Theta(n)$ *en 1991*

On a vu trois algorithmes pour 3-SUM de complexités $O(n^3)$, $O(n^2 \log n)$ et $O(n^2)$.

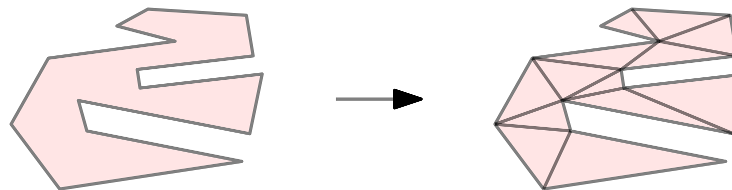
Complexité d'un algorithme $\rightsquigarrow$ complexité d'un problème

Quelle est la complexité **minimale** d'un algorithme résolvant 3-SUM ?

Peut on envisager $O(n \log n)$? $O(n\sqrt{n})$? $O(\sqrt{n})$? $O(1)$?

Conjecture (1995) : Tout algorithme pour 3-SUM a complexité $\Omega(n^2)$.

Les bornes supérieures sont généralement obtenues par des algorithmes explicites.



$O(n \log n)$ *en 1978*
$O(n \log \log n)$ *en 1986*
$O(n \log^* n)$ *en 1989*
$\Theta(n)$ *en 1991*

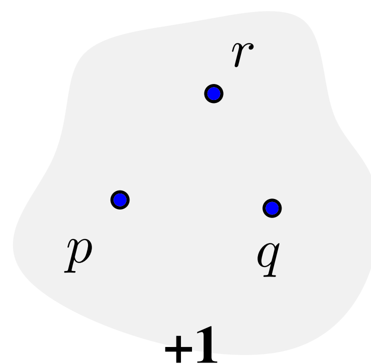
Comment établir une borne inférieure ?

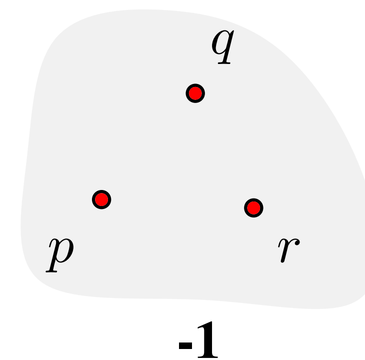
Intermède géométrique

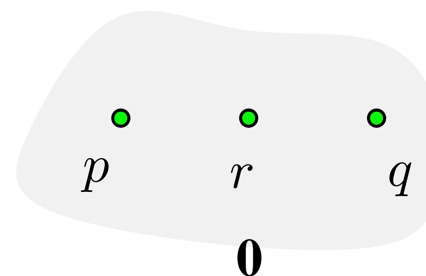
On considère le plan euclidien muni d'un repère orthonormé.

L'orientation d'un triplet ordonné de points du plan est...

$$\text{Orientation}(p, q, r) = \text{sign} \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}.$$



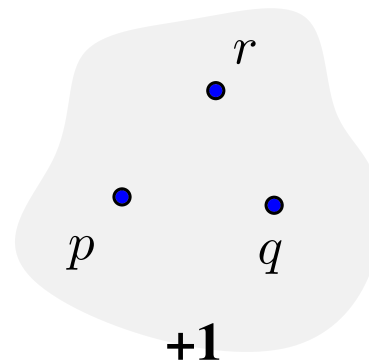


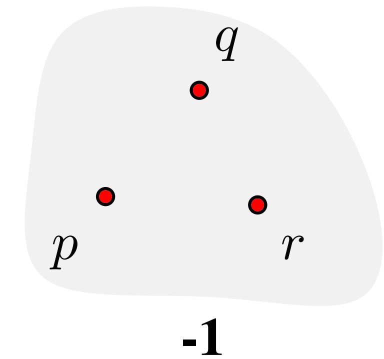


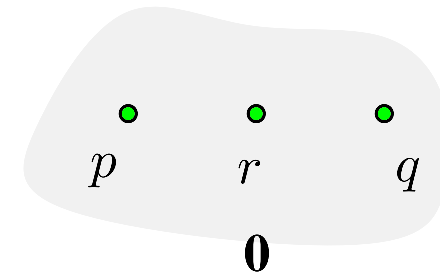
On considère le plan euclidien muni d'un repère orthonormé.

L'orientation d'un triplet ordonné de points du plan est...

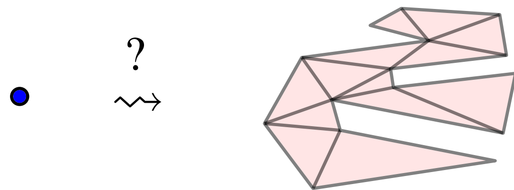
$$\text{Orientation}(p, q, r) = \text{sign} \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}.$$

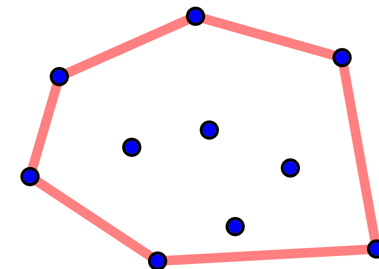






Prédicat géométrique élémentaire à la base de nombreux algorithmes.



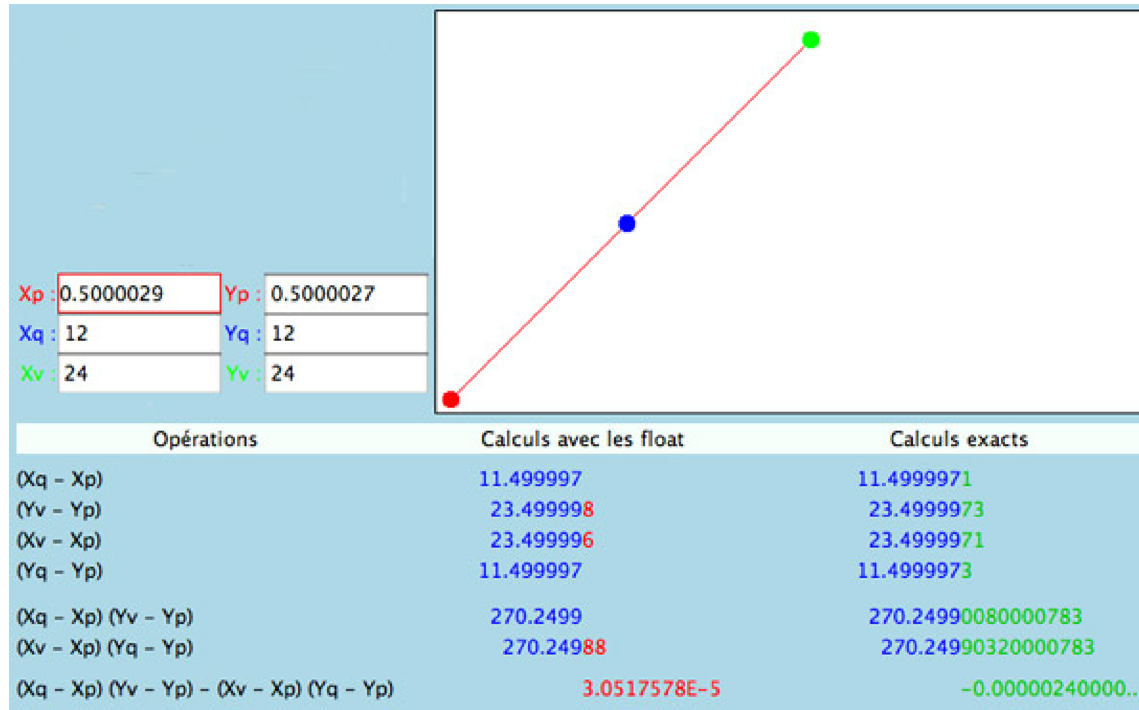


Le calcul sur machine se fait à précision fixée et des erreurs sont possibles.

```
Float xp,yp,xq,yq,xr,yr;
Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

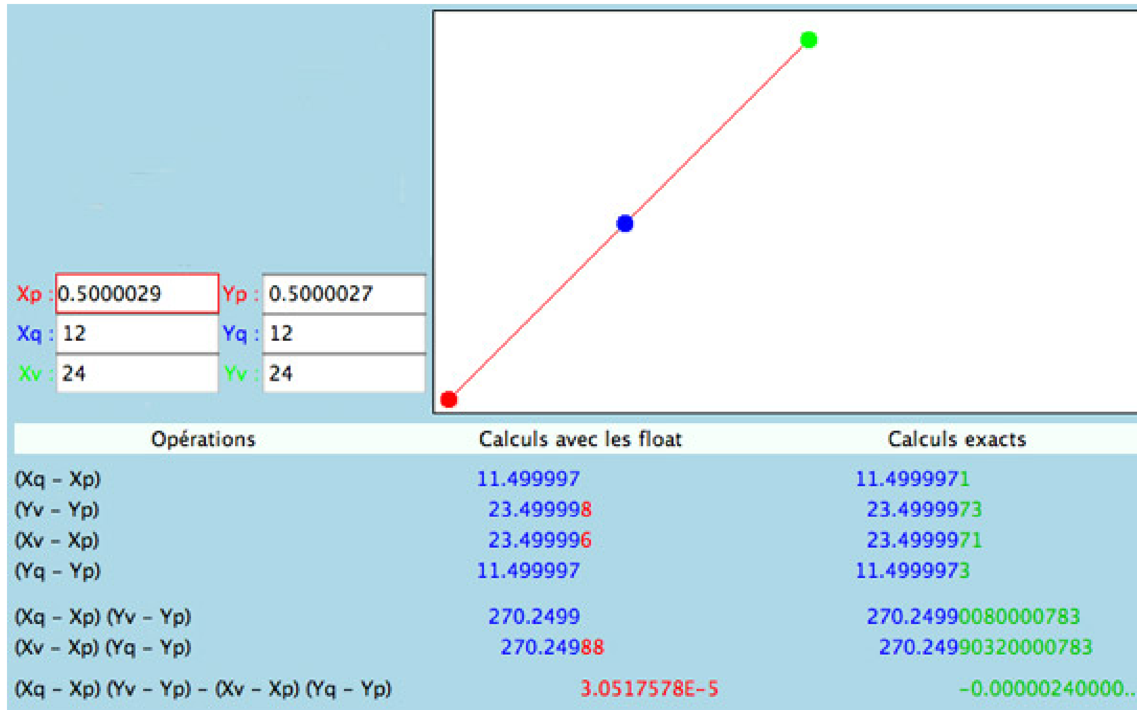
Le calcul sur machine se fait à précision fixée et des erreurs sont possibles.

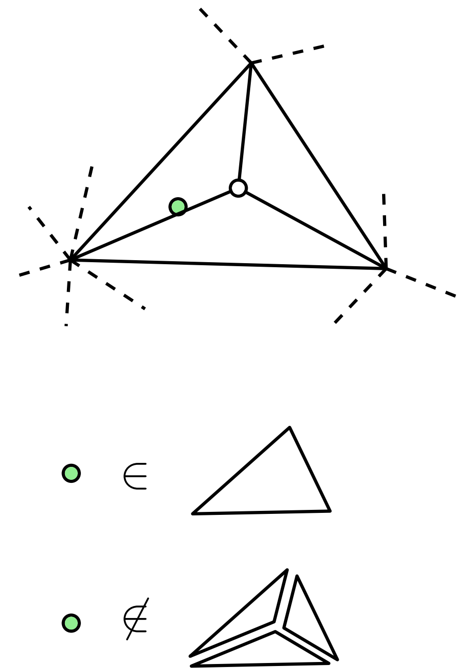
```
Float xp,yp,xq,yq,xr,yr;
Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```



Le calcul sur machine se fait à précision fixée et des erreurs sont possibles.

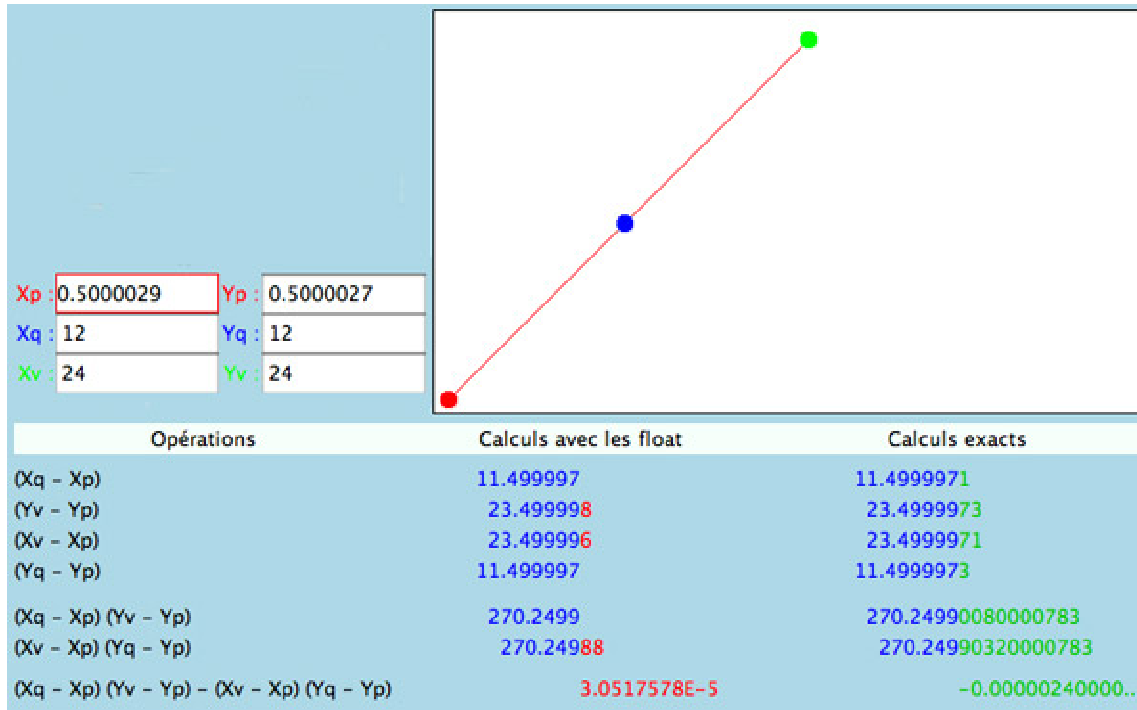
```
Float xp,yp,xq,yq,xr,yr;
Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

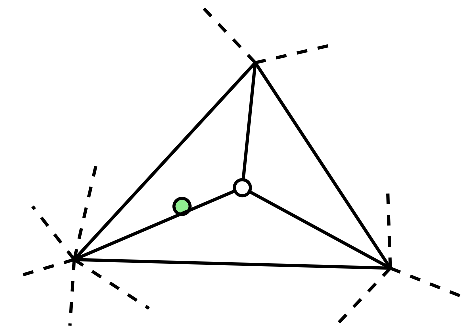


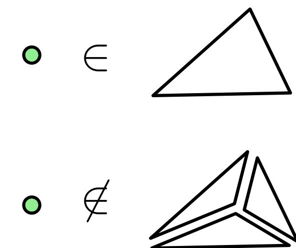


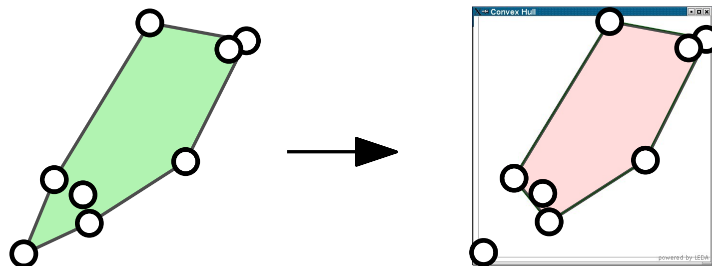
Le calcul sur machine se fait à précision fixée et des erreurs sont possibles.

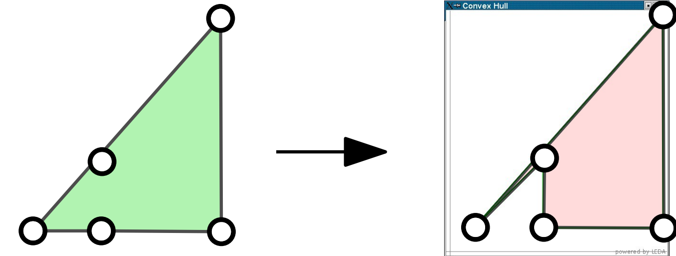
```
Float xp,yp,xq,yq,xr,yr;
Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```











Problème de **détection d'alignement** :

- *Entrée :* n *points du plan* $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* p_i*,* p_j *et* p_k *sont alignés.*

Un algorithme de complexité $O(n^3)$ est simple à concevoir.

Problème de **détection d'alignement** :

- *Entrée :* n *points du plan* $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* p_i, p_j *et* p_k *sont alignés.*

Un algorithme de complexité $O(n^3)$ est simple à concevoir.

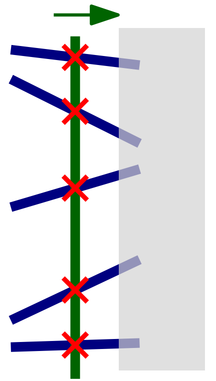
Tester les triplets un par un.

Problème de **détection d'alignement** :

- *Entrée :* n *points du plan* $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* p_i, p_j *et* p_k *sont alignés.*

Un algorithme de complexité $O(n^3)$ est simple à concevoir.

Tester les triplets un par un.



Un algorithme de complexité $O(n^2 \log n)$ peut s'expliquer en cours.

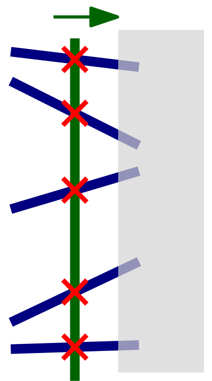
Dualité point-droite, algorithme de balayage.

Problème de **détection d'alignement** :

- *Entrée :* n *points du plan* $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* p_i*,* p_j *et* p_k *sont alignés.*

Un algorithme de complexité $O(n^3)$ est simple à concevoir.

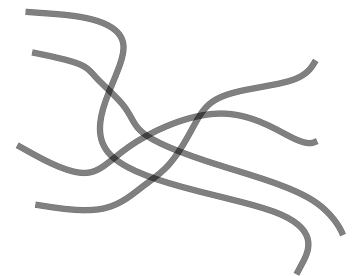
Tester les triplets un par un.



Un algorithme de complexité $O(n^2 \log n)$ peut s'expliquer en cours.

Dualité point-droite, algorithme de balayage.

Un algorithme de complexité $O(n^2)$ peut s'expliquer en quelques cours.



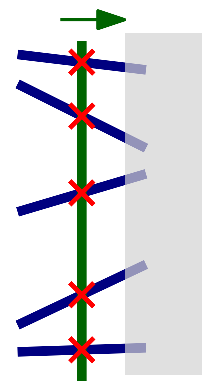
Pseudodroites, plans topologiques, tri topologique, queue de priorité en temps linéaire, dualité point-droite , algorithme de balayage.

Problème de **détection d'alignement** :

- *Entrée :* n *points du plan* $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$.
- *Sortie :* 1 *s'il existe trois indices* i, j, k *tels que* p_i*,* p_j *et* p_k *sont alignés.*

Un algorithme de complexité $O(n^3)$ est simple à concevoir.

Tester les triplets un par un.

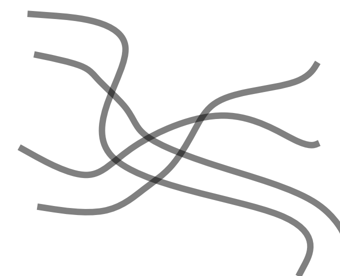


Un algorithme de complexité $O(n^2 \log n)$ peut s'expliquer en cours.

Dualité point-droite, algorithme de balayage.

Un algorithme de complexité $O(n^2)$ peut s'expliquer en quelques cours.

Pseudodroites, plans topologiques, tri topologique, queue de priorité en temps linéaire, dualité point-droite , algorithme de balayage.



On va montrer que

Algorithme sous-quadratique pour la détection d'alignement $\Rightarrow$ Algorithme sous-quadratique pour 3-SUM

Réductions

“pas plus de réponses, mais moins de questions”

Une **réduction** d'un problème A à un problème B consiste à :

Définir deux fonctions encodage $\mathbf{e}$ *et décodage* $\mathbf{d}$ *telles que*

pour toute entrée x *du problème* A, $\mathbf{e}(x)$ *est une entrée du problème* B

si y *est la réponse au problème* B *sur l'entrée* $\mathbf{e}(x)$
alors $\mathbf{d}(y)$ *est la réponse au problème* A *sur l'entrée* x.

Une **réduction** d'un problème A à un problème B consiste à :

Définir deux fonctions encodage $\mathbf{e}$ *et décodage* $\mathbf{d}$ *telles que*

pour toute entrée x *du problème* A, $\mathbf{e}(x)$ *est une entrée du problème* B

si y *est la réponse au problème* B *sur l'entrée* $\mathbf{e}(x)$
alors $\mathbf{d}(y)$ *est la réponse au problème* A *sur l'entrée* x.

Démarche courante en mathématiques pour résoudre A **via** B.

Intersection de droites du plan $\rightsquigarrow$ *résolution de systèmes linéaires.*

On va s'en servir pour transférer une borne inférieure de complexité de A vers B.

"S'il existe un algorithme rapide pour B*, il en existe un pour* A*."*

Cela nous amène à examiner les réductions de manière **quantitative**.

La complexité du calcul des encodage/décodage doit être controlée.

Illustrons cela sur un exemple...

On réduit

A = **3-SUM**

- *Entrée : n entiers*
- *Sortie : 1 si trois entiers distincts de somme nulle, 0 sinon.*

à

B = **détection d'alignement**

- *Entrée : n points du plan.*
- *Sortie : 1 s'il existe trois points alignés, 0 sinon.*

On réduit

A = 3-SUM

- *Entrée : n entiers*
- *Sortie : 1 si trois entiers distincts de somme nulle, 0 sinon.*

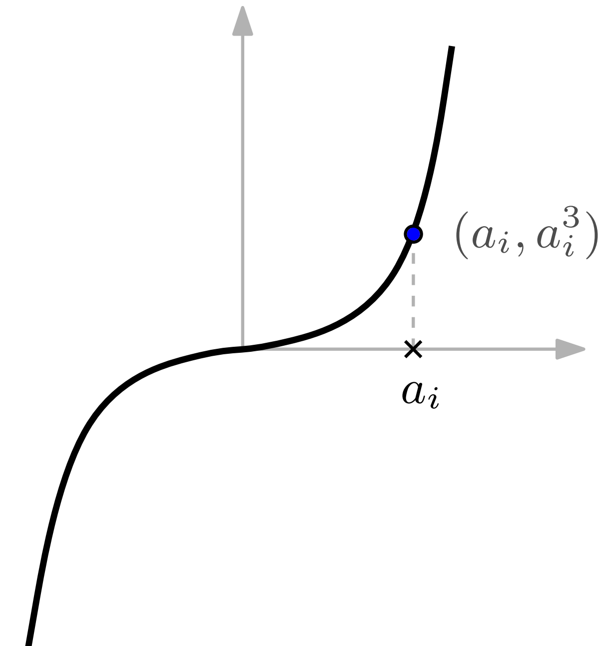
à

B = détection d'alignement

- *Entrée : n points du plan.*
- *Sortie : 1 s'il existe trois points alignés, 0 sinon.*

Encodage *entiers* $a_1, a_2, \ldots, a_n \quad \mapsto \quad$ *points* $(a_1, a_1^3), (a_2, a_2^3), \ldots, (a_n, a_n^3)$

Décodage $0 \mapsto 0$ *et* $1 \mapsto 1$.



On réduit

A = 3-SUM

- *Entrée : n entiers*
- *Sortie : 1 si trois entiers distincts de somme nulle, 0 sinon.*

à

B = détection d'alignement

- *Entrée : n points du plan.*
- *Sortie : 1 s'il existe trois points alignés, 0 sinon.*

Encodage *entiers* $a_1, a_2, \ldots, a_n \quad \mapsto \quad$ *points* $(a_1, a_1^3), (a_2, a_2^3), \ldots, (a_n, a_n^3)$

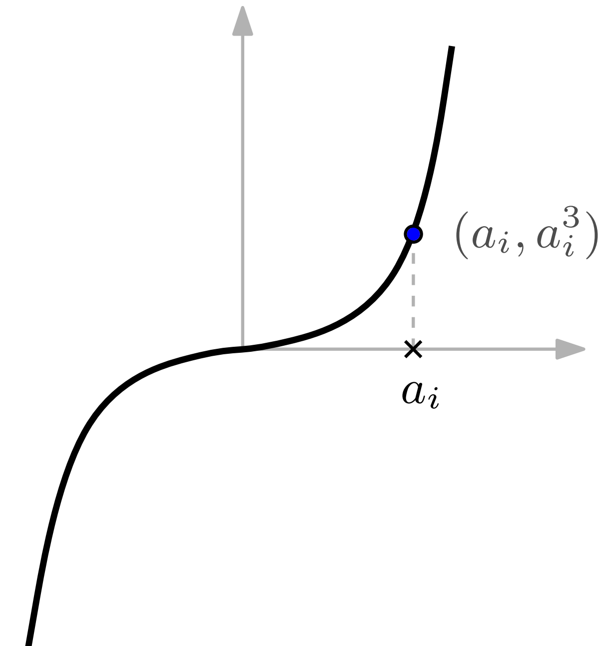
Décodage $0 \mapsto 0$ *et* $1 \mapsto 1$.

$a_i + a_j + a_k = 0 \Leftrightarrow (a_i, a_i^3)$, (a_j, a_j^3) et (a_k, a_k^3) sont alignés.

Preuve :

$$p, q, r \text{ alignés} \Leftrightarrow \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix} = 0.$$

$$\begin{vmatrix} t_i & t_j & t_k \\ t_i^3 & t_j^3 & t_k^3 \\ 1 & 1 & 1 \end{vmatrix} = (t_j - t_i)(t_k - t_i)(t_k - t_j)(t_i + t_j + t_k). \qquad \square$$



On réduit

A = 3-SUM

- *Entrée : n entiers*
- *Sortie : 1 si trois entiers distincts de somme nulle, 0 sinon.*

à

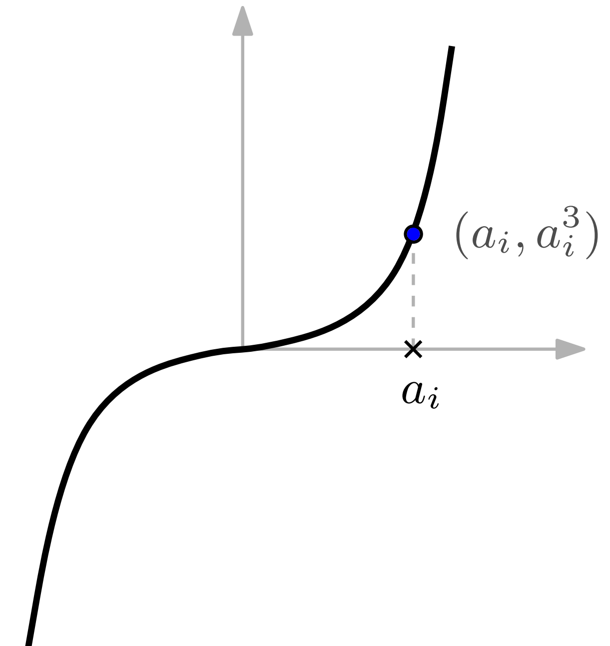
B = détection d'alignement

- *Entrée : n points du plan.*
- *Sortie : 1 s'il existe trois points alignés, 0 sinon.*

Encodage *entiers* $a_1, a_2, \ldots, a_n \quad \mapsto \quad$ *points* $(a_1, a_1^3), (a_2, a_2^3), \ldots, (a_n, a_n^3)$

Décodage $0 \mapsto 0$ *et* $1 \mapsto 1$.

$a_i + a_j + a_k = 0 \Leftrightarrow (a_i, a_i^3)$, (a_j, a_j^3) et (a_k, a_k^3) sont alignés.



Preuve :

$$p, q, r \text{ alignés} \Leftrightarrow \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix} = 0.$$

$$\begin{vmatrix} t_i & t_j & t_k \\ t_i^3 & t_j^3 & t_k^3 \\ 1 & 1 & 1 \end{vmatrix} = (t_j - t_i)(t_k - t_i)(t_k - t_j)(t_i + t_j + t_k). \qquad \square$$

Encodage en temps $O(n)$, entrée encodée de taille $O(n)$, décodage en temps $O(1)$.

Si **detection d'alignement** admet un algo. sous-quadratique alors **3-SUM** aussi...

3-SUM et **Détection d'alignement** peuvent être résolus en temps $O(n^2)$.

Les structures mises en jeu ont des degrés de sophistication différentes.

ordre sur $\mathbb{Z}$ VS *Pseudodroites, plans topologiques, tri topologique, queue de priorité en temps linéaire, dualité point-droite , algorithme de balayage.*

On conjecture que 3**-SUM** n'a pas d'algorithme sous-quadratique.

La réduction transfère cette conjecture à la **détection d'alignement**.

3-SUM et **Détection d'alignement** peuvent être résolus en temps $O(n^2)$.

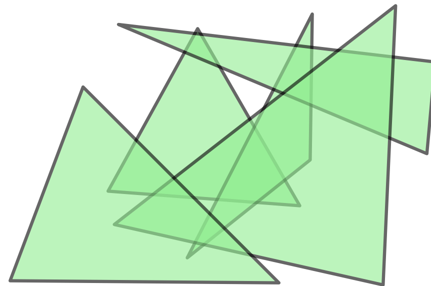
Les structures mises en jeu ont des degrés de sophistication différentes.

ordre sur $\mathbb{Z}$ VS *Pseudodroites, plans topologiques, tri topologique, queue de priorité en temps linéaire, dualité point-droite , algorithme de balayage.*

On conjecture que 3**-SUM** n'a pas d'algorithme sous-quadratique.

La réduction transfère cette conjecture à la **détection d'alignement**.

Depuis 1995, des dizaines de problèmes 3-SUM difficiles.



Décider si une union de triangles a un trou, ie si le complémentaire est connexe.

3**-SUM** identifie un verrou dans notre compréhension de l'algorithmique.

Pour conclure

La description et l’analyse d’un algorithme se fait dans un **modèle de calcul**.

Souvent laborieux et “bas niveau”.

Nécessaire à la formulation d’algorithmes car décrit les opérations élémentaires.

Le modèle de “cartes à retourner” est simple et induit des contraintes suffisantes.

La description et l'analyse d'un algorithme se fait dans un **modèle de calcul**.

Souvent laborieux et "bas niveau".

Nécessaire à la formulation d'algorithmes car décrit les opérations élémentaires.

Le modèle de "cartes à retourner" est simple et induit des contraintes suffisantes.

Notion de **complexité d'un problème**, souvent délicate à déterminer.

Produit de matrices $n \times n$ *:* $O(n^3) \to O(n^{2.807}) \to O(n^{2.376}) \to O(n^{2.373}) \to O(n^{2.372})$

On connaît peu de bornes inférieures **absolues** sur la complexité de problèmes.

La description et l'analyse d'un algorithme se fait dans un **modèle de calcul**.

Souvent laborieux et "bas niveau".

Nécessaire à la formulation d'algorithmes car décrit les opérations élémentaires.

Le modèle de "cartes à retourner" est simple et induit des contraintes suffisantes.

Notion de **complexité d'un problème**, souvent délicate à déterminer.

Produit de matrices $n \times n$: $O(n^3) \to O(n^{2.807}) \to O(n^{2.376}) \to O(n^{2.373}) \to O(n^{2.372})$

On connaît peu de bornes inférieures **absolues** sur la complexité de problèmes.

Une **réduction** est une modélisation, avec une finalité inversée.

"aussi difficile que" plutôt que "aussi facile que".

Les réductions permettent d'identifier des problèmes **épurés** contenant des verrous.

Les réductions sont un outil central en **théorie de la complexité** depuis les années 70.

$P \stackrel{?}{=} NP$ confronte résoudre (P) et vérifier une solution (NP).

	3		5					
9		7			4			
	8	2		3				4
4			9				6	
		5		7		8		
	9				8			7
		3		8		6	7	
			6			5		2
					5		1	

VS

1	3	4	5	9	6	7	2	8
9	5	7	8	2	4	1	3	6
6	8	2	1	3	7	4	9	5
4	7	8	9	5	2	3	6	1
2	6	5	3	7	1	8	4	9
3	9	1	4	6	8	2	5	7
5	1	3	2	8	9	6	7	4
7	4	9	6	1	3	5	8	2
8	2	6	7	4	5	9	1	3

Un problème est NP-difficile si tout problème NP s'y réduit en temps polynomial.

Sudoku en $n^2 \times n^2$, tetris, coloration de graphe...

On conjecture qu'aucun problème NP-difficile n'a d'algorithme polynomial.

Au mieux des bornes inférieures quadratiques !