Diviser pour Régner Divide and Conquer ACT

Sophie Tison Master Informatique S1

2025-2026

Le principe

RAPPELS: BILAN

- Quand on parle de la complexité sans préciser, c'est souvent de la complexité temporelle dans le pire des cas
- ▶ Praticable = polynomial: un algorithme "en 2^{n} " qui s'exécute en 1 seconde pour une donnée de taille n = 10, nécessite plus d'une semaine pour n = 30
- L'analyse asymptotique de la complexité ne révèle pas ce qui se passe sur des "petites" données et n'exempte pas de tester et d'expérimenter
- Attention à la taille de la donnée!

SALLES

- ► G1 : TD en M5 A8 TP en M5 A16 vendredi 13h15
- ► G2: TD en B09 Uranus TP en M5 A13
- ► G3: TD en B09 Saturne TP en M5 A14
- ► G4 : TD en M5 A3 TP en M5 A15 vendredi 13h15
- ► G5 : TD en B09 Jupiter TP en M5 A16
- ► G6: TD in B09 Mars TP en M5 A11

Le principe

PLAN

- Partie 1. Quelques schémas ou paradigmes d'algorithmes
- Partie 2. Un peu de complexité de problèmes
- Partie 3. Heuristiques et méta-heuristiques

PARADIGME?

DILBERT COMICS







Ici paradigme = approche pour résoudre un problème

Le principe

DIVISER POUR RÉGNER RÉSOUDRE

DIVIDE AND CONQUER / DIVIDE UT REGNES / DIVIDE ET IMPERA



Philippe II de Macédoine

- cas de base: savoir résoudre les "petits" problèmes.
- ▶ Diviser le problème en sous-problèmes de taille "vraiment" plus petite: typiquement la taille est divisée par une constante.
- ▶ Résoudre les sous-problèmes par induction
- ► Combiner les solutions des sous-problèmes

Simplifier et déléguer

Le principe

PARADIGMES/SCHÉMAS D'ALGORITHMES

ALGORITHMIC DESIGN PATTERN

- ► Diviser pour régner
- ► Programmation Dynamique
- ► Algorithmes Gloutons
- Backtracking
- Branch and Bound
- **.**..

Le principe o● Exemples

DIVISER POUR RÉGNER: LE SCHÉMA

```
DivReg (Probleme P)

if P est "petit" then

| CasBase(P);

else

| Diviser P en P1, ..Pk "vraiment" plus petits;

| CombinerSolution(DivReg(P1), DivReg(P2), ...DivReg(Pk));

end

Algorithm 1: Le schéma diviser pour régner
```

UN PREMIER EXEMPLE: RECHERCHE DICHOTOMIQUE PRINCIPE

Problème: chercher une valeur x dans un tableau trié Principe: on compare à la valeur du milieu :

- ▶ si elle est égale à x, on l'a trouvé
- si elle est plus petite, on cherche à droite,
- sinon, on cherche à gauche!

Donc on recherche dans une tranche de taille environ la moitié de la taille initiale.

Le principe

UN PREMIER EXEMPLE: RECHERCHE DICHOTOMIQUE

Exemples?

```
VERSION RÉCURSIVE
   Input: T trié croissant, x, g \le d
   Output: retourne Vrai Ssi x présent dans T[g..d]
   boolean rechDico(int x, int g, int d)
   if (g == d) then
       return (T[g] == x);
    else
       int m = (g + d)/2;
       if (T[m] < x) then
          return rechDicho(x, m + 1, d);
       else
          return rechDicho(x, g, m);
       end
    end
```

Arrêt: d - g décroît strictement à chaque appel, on a toujours $g \leq d$.

Correction: par induction

Le principe

RECHERCHE DICHOTOMIQUE

VERSION ITÉRATIVE

```
Input: T trié croissant non vide, x,
Output: retourne Vrai Ssi x présent dans T
boolean rechDico(int x)
int g = 0, d = T.length - 1;
while (g! = d) do
   int \overline{m} = (g + d)/2;
   if (T[m] < x) then
      g = m + 1;
   else
      d=m:
   end
end
return (T[g]==x);
```

Le principe

RECHERCHE DICHOTOMIQUE

CORRECTION?

```
Input: T trié croissant non vide, x

Output: retourne Vrai Ssi x présent dans T boolean rechDico(int x)

int g = 0, d = T.length - 1;

while (g! = d) do

// Invariant: ??

int m = (g + d)/2;

if (T[m] < x) then

| g = m + 1;

else

| d = m;

end

end

return (T[g] == x);
```

Le principe

RECHERCHE DICHOTOMIQUE

CORRECTION?

Il existe de nombreuses versions correctes de recherche dichotomique ... et aussi de nombreuses façons de se "planter"...- algorithme qui boucle, qui n'est pas en $O(\log n)$, qui oublie certaines valeurs...



RECHERCHE DICHOTOMIQUE

```
CORRECTION?
```

```
Input: T non vide trié croissant, x
Output: retourne Vrai Ssi x présent dans T
boolean rechDico(int x)
int g = 0, d = T.length - 1;
while (g! = d) do
   // Invariant: T trié, g \le d, si x est dans T, x est dans
   T[g..d]
   int m = (g + d)/2;
   if (T[m] < x) then
      g=m+1;
   else
      d=m;
   end
end
return (T[g]==x);
Arrêt: d - g décroît strictement.
```

Le principe

RECHERCHE DICHOTOMIQUE

COMPLEXITÉ?

```
Input: T non vide trié croissant, x

Output: retourne Vrai Ssi x présent dans T

boolean rechDico(int x)

int g = 0, d = T.length -1;

while (g! = d) do

int m = (g + d)/2; /* nb de boucles???? */

if T[m] < x then

g = m + 1;

else

d = m

end

end

return (T[g] = x);
```

RECHERCHE DICHOTOMIQUE

COMPLEXITÉ?

```
Soit n = T.length;

Input: T non vide trié croissant, x

Output: retourne Vrai Ssi x présent dans T

boolean rechDico(int x)

int g = 0, d = T.length - 1;

while (g! = d) do

int m = (g + d)/2; /* \Theta(\log n) boucles */

if (T[m] < x) then

g = m + 1;

else

d = m

end

end

return (T[g] = x);
```

Le principe

RECHERCHE DICHOTOMIQUE

Complexité en fonction de n = T.length?

```
Input: T non vide trié croissant, x

Output: retourne Vrai Ssi x présent dans T boolean rechDico(int x) int g = 0, d = T.length - 1; while (g! = d) do int m = (g + d)/2; /* \Theta(\log n) boucles en \Theta(1) */

if (T[m] < x) then g = m + 1; else d = m end end return (T[g] = x);

L'algorithme est en \Theta(\log n).
```

RECHERCHE DICHOTOMIQUE

COMPLEXITÉ?

```
Soit n = T.length;

Input: T non vide trié croissant, x

Output: retourne Vrai Ssi x présent dans T

boolean rechDico(int x)

int g = 0, d = T.length - 1;

while (g! = d) do

int m = (g + d)/2; /* coût d'une boucle??? */

if (T[m] < x) then

g = m + 1;

else

d = m

end

end

return (T[g] = x);
```

Le principe

LA COMPLEXITÉ DE LA VERSION RÉCURSIVE?

```
boolean rechDico(int x, int g, int d)

if (g == d) then

return (T[g] == x);

else

int m = (g + d)/2;

if (T[m] < x) then

return rechDicho(x, m + 1, d);

else

return rechDicho(x, g, m);

end

end
```

Soit A(n) est le nombre d'appels récursifs faits pour un tableau de taille n.

```
A(1) = 1, A(n) = 1 + A(n/2)
```

L'algorithme est en $O(\log n)$

Le principe

Un autre exemple?

Le principe

LE TRI FUSION

```
Input: L Liste

Output: L triée

Liste triF(L)

if (L.length \le 1) then

| return L

else

| mil = L.length/2;

return fusion(triF(L[0..mil - 1]), triF(T[mil..L.length - 1]))
end
```

Avec fusion qui fusionne deux listes triées en une liste triée.

Preuve:par induction

Arrêt: longueur de la liste à trier décroît strictement à chaque appel

UN AUTRE EXEMPLE: LE TRI FUSION- MERGE SORT

Le principe

COMPLEXITÉ?

Comptons le nombre de comparaisons d'éléments:

La fusion de deux listes triées de k éléments nécessite au plus 2k-1 comparaisons d'éléments.

Soit Comp(n) le nombre de comparaisons faites pour trier n éléments. Supposons n pair:

$$Comp(n) \le 2 * Comp(n/2) + n$$

 $Comp(1) = 0$

ANALYSE DE LA COMPLEXITÉ?

$$Comp(n) \le 2 * Comp(n/2) + n$$

 $Comp(1) = 0$

Le principe

Analyse de la Complexité: Méthode 2

Supposons
$$n = 2^k$$
.
 $Comp(n) <= 2 * Comp(n/2) + n$
 $2 * Comp(n/2) <= 4 * Comp(n/4) + n$ remplacer n par $n/2$...
 \vdots
 $2^i * Comp(n/2^i) <= 2^{i+1} * Comp(n/2^{i+1}) + n$
 \vdots
 $2^k * Comp(n/2^k) = 0$
Soit en sommant et en simplifiant:
 $Comp(n) <= n * k$ ou encore
 $Comp(n) <= n * \log n$

Analyse de la Complexité: Méthode 1

EN UTILISANT L'ARBRE DES APPELS

$$Comp(n) \le 2 * Comp(n/2) + n$$

 $Comp(1) = 0$

On représente l'arbre (voir tableau!) complexité en $O(n \log n)$.

Diviser la taille des sous-problèmes



Avoir un arbre d'appels de hauteur logarithmique

Le principe

Le principe

Analyse de la Complexité: Méthode 3

LE MASTER THEOREM -VERSION ALLÉGÉE

Master Theorem -version light

Si
$$T(n) = aT(\lfloor n/b \rfloor) + O(n^d)$$
, avec $a > 0, b > 1, d \ge 0$ alors

- ightharpoonup si $d > \log_h a$, $T(n) = O(n^d)$
- ightharpoonup si $d = \log_b a, T(n) = O(n^d \log n)$
- ightharpoonup si $d < \log_b a, T(n) = O(n^{\log_b a})$

Exemple de la Recherche dichotomique:

$$T(n) = T(n/2) + O(1)$$
 donc $a = 1, b = 2, d = 0$
On est dans le cas 2, et $T(n) = O(\log n)$

LE MASTER THEOREM POUR LE MERGE SORT

Master Theorem -version allégée

Si $T(n) = aT(|n/b|) + O(n^d)$, avec $a > 0, b > 1, d \ge 0$ alors

- ightharpoonup si $d > \log_h a$, $T(n) = O(n^d)$
- ightharpoonup si $d = \log_b a$, $T(n) = O(n^d \log n)$
- ightharpoonup si $d < \log_b a, T(n) = O(n^{\log_b a})$

Comp(n) = 2 * Comp(n/2) + O(n)

Donc a = b = 2, d = 1

On est encore dans le cas 2, la complexité est ici $O(n \log n)$.

Le principe

LA MULTIPLICATION D'ENTIERS

Donnée: x, y entiers > 0 à n chiffres

Sortie: x * y

On compte le nombre d'opérations élémentaires sur les chiffres.

Algorithme de l'école primaire

$$\begin{array}{r} \times \begin{array}{r} 3 & 8 & 4 & 1 & 3 & 2 \\ & 2 & 3 & 5 & 5 & 6 \\ \hline 2 & 3 & 0 & 4 & 7 & 9 & 2 \\ & 1 & 9 & 2 & 0 & 6 & 6 & 0 \\ 1 & 9 & 2 & 0 & 6 & 6 & 0 \\ 1 & 1 & 5 & 2 & 3 & 9 & 6 \\ \hline 7 & 6 & 8 & 2 & 6 & 4 \\ \hline 9 & 0 & 4 & 8 & 6 & 1 & 3 & 3 & 9 & 2 \end{array}$$

combien d'opérations élémentaires? environ n^2 multiplications de chiffres, et environ $2n^2$ additions de chiffres

Un troisième exemple

Le principe

DIVISER POUR MULTIPLIER

Supposons $n = 2^k$

Soit x_1 (reps. y_1) formé des n/2 premiers chiffres (les plus à gauche) de x (resp. de y), x_2 (resp. y_2) formé des n/2 derniers.

Donc:
$$x = 10^{n/2}x_1 + x_2$$
, $y = 10^{n/2}y_1 + y_2$
et: $x * y = (10^{n/2}x_1 + x_2) * (10^{n/2}y_1 + y_2)$
Soit: $x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$

On obtient un algorithme dont la complexité est donnée par:

$$C(n) = 4 * C(n/2) + 0(n)$$

Master Theorem: $a = 4, b = 2, d = 1 < 2 = \log_2 4$

On obtient encore un algorithme en $O(n^2)$. \bigcirc

Le principe Exemples

L'ASTUCE DE KARATSUBA

 $x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$ Il suffit de faire trois multiplications d'entiers à au plus n/2 chiffres:

- 1. $x_1 * y_1$
- 2. $x_2 * y_2$
- 3. $(x_1 x_2) * (y_2 y_1) = x_1 * y_2 + x_2 * y_1 x_1 * y_1 x_2 * y_2$

On obtient un algorithme dont la complexité est donnée par:

$$C(n) = 3C(n/2) + O(n)$$

Par le Master Theorem avec a = 3, b = 2, d = 1,

On obtient un algorithme en $O(n^{\log_2 3})$ ($\log_2 3 \approx 1.58$).

C'est l'algorithme de Karatsuba.

Le principe

UN DERNIER EXEMPLE: LA PAIRE DE POINTS LES PLUS PROCHES

Le problème :

 $\overline{\text{Soit } P \text{ un ensemble de } n \text{ points du plan.}}$

On cherche la paire de points les plus proches (pour la distance euclidienne).

Pour simplifier, on supposera que tous les points ont une abscisse différente (pas nécessaire mais plus simple pour la présentation)

LA MULTIPLICATION D'ENTIERS

Donnée:

x, y entiers < 0 à n chiffres

Sortie: x * y

On compte le nombre d'opérations élémentaires sur les chiffres.

- ▶ Algo de l'école primaire: $O(n^2)$
- ▶ Multiplication à l'égyptienne plus efficace mais aussi en $O(n^2)$, si on compte les opérations sur les chiffres.
- Algorithme de Karatsuba en $O(n^{\log_2 3})$
- ▶ Algorithme de Toom-Cook, toujours par "Diviser pour régner", en $O(n^{\log_3 5})$ en divisant en trois parties.
- ► Algorithme Schoenage-Strassen (1971), basé sur la FFT en $O(n \log n \log(\log n))$
- Algorithme de Furer (2007) en $O(n \log nK^{O(\log^* n)})$
- Algorithmes en $O(n \log n)$ en 2019, pas encore utilisables en pratique
- ► On ne sait si on peut faire mieux! (voir article de gazette IM 07/21: https://www.gdr-im.fr/gazette/)

Le principe

EN DIMENSION 1?

Le problème :

Soient *n* points d'une droite.

On cherche la paire de points les plus proches (pour la distance euclidienne).

Il suffit de trier par une coordonnée et de calculer la distance entre un point et son successeur!

On obtient un algorithme en $O(n \log n)$

Le principe

EN DIMENSION 2?

Une méthode exhaustive serait en $O(n^2)$.

Le principe

DIVISER L'ENSEMBLE DES POINTS EN DEUX

```
Input: P_x la liste des points de P triée par x croissant cpt = 0;

for x dans P_x do

if (cpt \le n/2) then
| ajouter x à G
else
| ajouter x à D;
end
| cpt + +;
```

Remarque: *G* et *D* sont triés aussi par *x* croissant.

DIVISER POUR RÉGNER?

<u>Idée:</u> Diviser le plan en deux parties, chacune contenant la moitié des points.

Par exemple, on peut le couper par la droite x = med avec med la valeur médiane des x.

Remarque: pourquoi a-t-on supposé les abscisses des points sont toutes différentes?

Le principe

DIVISER L'ENSEMBLE DES POINTS EN DEUX

```
Input: P_x la liste des points de P triée par x croissant cpt = 0; for \underline{x} dans P_x do

if \underline{(cpt \le n/2)} then
| ajouter \underline{x} à \underline{G}
else
| ajouter \underline{x} à \underline{D};
end
\underline{cpt + +};
```

Coût= celui du tri soit en $O(n \log n)$.

DIVISER POUR RÉGNER?

- ▶ Diviser le plan en deux parties, chacune contenant la moitié des points.
- ► On calcule la paire des points plus proches dans chacune des deux parties.
- ► Problème? Peut-être que la paire est composée d'un point de chaque partie!
- ► Comment faire?

Le principe

UN LEMME...

Remarque: si deux points sont à distance $< \delta$ et de part et d'autre de la droite x = med, ils sont à distance au plus δ de la droite.

On peut donc limiter la recherche aux points de S l'ensemble des points (x, y) tels que $med - \delta \le x \le med + \delta$.

Lemma

Soit $\delta = min(min_G, min_D)$

Soit S trié par y croissant.

Alors si deux points de chaque côté de la droite sont séparés par au moins 7 points dans la liste, ils sont à distance $> \delta$.

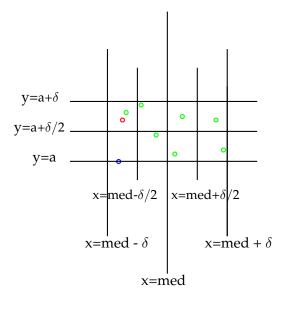
RÉCAPITULATIF

- 1. Trier les points selon *x*
- 2. Partager en deux parties de même cardinal (à 1 près) par une droite verticale x = med, G, D.
- 3. Résoudre à gauche et à droite. cela donne une distance minimale min_G à gauche, min_D à droite.
- 4. Trouver, si il existe un point de G et un point de D, à distance inférieure à $\delta = min(min_G, min_D)$; si oui, soit la distance minimale min_{GD} entre un point de G et un point de D
- 5. La distance minimale est donc le minimum de min_G , min_D , ou min_{GD} selon la réponse du 4.

Coût du point 4?

Le principe

Au plus un point par "carré"



RÉCAPITULONS

- 1. Initialisation
 - 1.1 trier les points selon x pour construire S_x :
 - 1.2 trier les points selon y pour construire S_y
- 2. Partager en deux parties de même cardinal (à 1 près) par une droite verticale x = med: on obtient $G(G_x, G_y)$, $D(D_x, D_y)$.
- 3. Résoudre à gauche et à droite. Cela donne une distance minimale min_G à gauche, min_D à droite.
- 4. Trouver, si il existe un point de G et un point de D, à distance inférieure à $\delta = min(min_G, min_D)$; on notera alors la distance minimale min_{GD} entre un point de G et un point de D
- 5. La distance minimale est donc le minimum de min_G , min_D , resp. min_{GD} .

La complexité est donnée par l'équation C(n) = 2C(n/2) + O(n)L'algorithme sera en $O(n \log n)$.

Le principe

QUELQUES EXEMPLES CLASSIQUES

- les algorithmes de recherche dichotomique, recherche de la médiane, ..
- le tri fusion, le tri rapide
- ▶ les algorithmes de multiplication: entiers (Karatsuba), matrices (Strassen)
- La transformée de Fourier rapide, FFT
- ► Géométrie algorithmique: Enveloppe convexe.

LA PAIRE DE POINTS LES PLUS PROCHES

Le problème

Soit *P* un ensemble de *n* points du plan.

On cherche la paire de points la plus proche (pour la distance euclidienne).

On a obtenu un algorithme en $O(n \log n)$.

L'algorithme peut être généralisé en dimension > 2.

Le principe

BILAN

- ▶ Diviser un problème en problèmes de taille nettement plus petite i.e. a priori divisée par une constante.
- ► Attention à la recombinaison!
- On peut éventuellement appliquer le <u>Master Theorem</u> pour la complexité
- ► Souvent plus facile à écrire de façon récursive
- ▶ Peut être intéressant à paralléliser (surtout si division et recombinaison peu coûteuses).