Diviser pour Régner
Divide and Conquer
ACT

Sophie Tison
Master Informatique S1
Université de Lille

2018-2019

Trois paradigmes d'algorithmes

- Rappels
- Paradigmes d'algorithme
- Complexité des problèmes
- Algorithmique Avancée

Paradigme?
Dilbert comics

Ici paradigme = approche pour résoudre un problème

Trois paradigmes/schémas d'algorithmes
Algorithmic Design Pattern

- Diviser pour régner
- Programmation Dynamique
- Algorithmes Gloutons

Trois paradigmes/schémas d'algorithmes

Algorithmic Design Pattern

- Diviser pour régner
- Programmation Dynamique
- Algorithmes Gloutons

Diviser pour Régner

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 contante)
- Résoudre les sous-problèmes par induction
- Combiner les solutions des sous-problèmes

Simplifier et déléguer

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

Exemples?

Un premier exemple: recherche dichotomique

Version Récursive

Input: T trié croissant, x, $g \leq d$
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.
Correction: par induction

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 m = (g + d)/2;
    if (T[m] < x) then
        g = m + 1;
    else
        d = m;
    end
end
return ( T[g]==x);
```


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);
```


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 ≤ d, si x est dans T T[g] ≤ x ≤ T[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.

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
    // Invariant: T trié, g ≤ d, si x est dans T T[g] ≤ x ≤ T[d]
    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
    // Invariant: T trié, g ≤ d, si x est dans T T[g] ≤ x ≤ T[d]
    int m = (g + d)/2 ;        /* Θ(log n) 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
    // Invariant: T trié, g ≤ d, si x est dans T T[g] ≤ x ≤ T[d]
    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);
```


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
    // Invariant: T trié, g ≤ d, si x est dans T T[g] ≤ x ≤ T[d]
    int m = (g + d)/2 ; /* Θ(log n) boucles en Θ(1) */
    if (T[m] < x) then
        g = m + 1;
    else
        d = m
    end
end
return ( T[g]==x);
```

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

La complexité de la Version récursive?

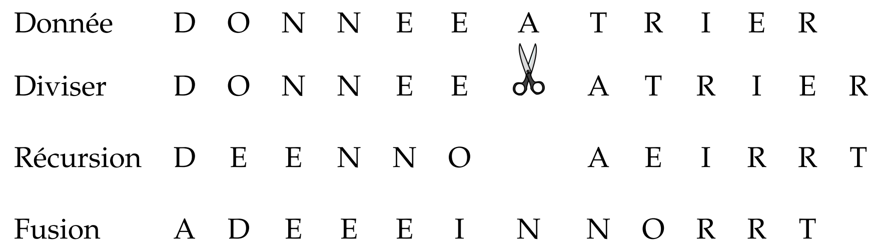
boolean *rechDico*(int x, int g, int d)
if $(g == d)$ **then**
  return $(T[g] == x)$;
else
  int $m = (g + d)/2$;
  if $(T[m] < x)$ **then**
    return $(rechDicho(x, m + 1, d))$;
  else
    return $(rechDicho(x, g, m))$;
  end
end

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 $O(\log n)$

Un autre exemple?



Un autre exemple: Le Tri fusion- Merge Sort

Input: L Liste
Output: L triée
Liste *triF*(L)
if $(L.length \leq 1)$ **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

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:

$$Comp(n) <= 2 * Comp(n/2) + n$$

$$Comp(1) = 0$$

Analyse de la Complexité?

$$Comp(n) <= 2 * Comp(n/2) + n$$

$$Comp(1) = 0$$

Analyse de la Complexité: Méthode 1

en utilisant l'arbre des appels

$$Comp(n) <= 2 * Comp(n/2) + n$$

$$Comp(1) = 0$$

On représente l'arbre (voir tableau!)

complexité en $O(n \log n)$.

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

$Comp(n) <= 2 * Comp(n/2) + n$

$2 * Comp(n/2) <= 4 * Comp(n/4) + n$

$\vdots$

$2^i * Comp(n/2^i) <= 2^{i+1} * Comp(n/2^{i+1}) + \mathrm{n}$

$\vdots$

$2^k * Comp(n/2^k) = 0$

Soit en sommant et en simplifiant:

$Comp(n) <= n * k$ ou encore

$Comp(n) <= n * \log n$

Analyse de la Complexité: Méthode 3
Le Master Theorem -version allégée

Master Theorem -version allégée

Si $T(n) = kT(\lfloor n/b \rfloor) + O(n^d)$, avec $k > 0, b > 1, d \geq 0$ alors

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

Exemple de la Recherche dichotomique:
$T(n) = T(n/2) + O(1)$ donc $k = 1, b = 2, d = 0$
On est dans le cas 2, et $T(n) = O(\log n)$

Le Master Theorem pour le Merge Sort

Master Theorem -version allégée

Si $T(n) = kT(\lfloor n/b \rfloor) + O(n^d)$, avec $k > 0, b > 1, d \geq 0$ alors

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

$Comp(n) = 2 * Comp(n/2) + O(n)$
Donc $k = b = 2, d = 1$
On est encore dans le cas 2, la complexité est ici $O(n \log n)$.

Un troisième exemple

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

```
          3 8 4 1 3 2
    ×       2 3 5 5 6
    -----------------
        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
  7 6 8 2 6 4
  -------------------
  9 0 4 8 6 1 3 3 9 2
```

combien d'opérations élémentaires? n^2

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: $k = 4, b = 2, d = 1 < 2 = \log_2 4$

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

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. $(x_1 - x_2) * (y_2 - y_1)$

On obtient un algorithme dont la complexité est donnée par:
$C(n) = 3C(n/2) + O(n)$
Par le Master Theorem avec $k = 3, b = 2, d = 1$,

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

C'est l'algorithme de Karatsuba.

La multiplication d'entiers

Donnée:
x, y entiers < 0 àn 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 (ou à la russe ou ..) 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 Tomm-Cook, toujours par "Diviser pour régner", par exemple en $O(n^{\log_3 5})$ pour la variante où l'entier est divisé en trois parties.
- Algorithme Schoenage-Strassen, basé sur la FFT en $O(n \log n \, log(\log n))$
- On ne sait si on peut faire mieux!

Un dernier exemple: La paire de points les plus proches

Le problème :
Soit P un ensemble de n 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)

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 composante et de calculer la distance entre un point et son successeur!

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

En dimension 2?

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

Diviser pour régner?

Idée: 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?

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 $(cpt \leq n/2)$ **then**
    ajouter x à G
  else
    ajouter x à D;
  end
  $cpt++$;
end

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

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 ≤ n/2) then
        ajouter x à G
    else
        ajouter x à D;
    end
    cpt + +;
end
```

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?

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?

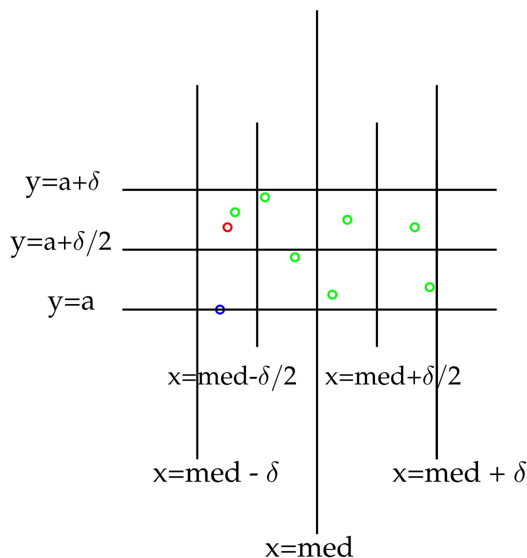
Un lemme...

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 $med - \delta \leq x \leq med + \delta$.

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 $> \delta$.

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 $O(n \log n)$.

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.

Quelques exemples classiques

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

Bilan

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