Diviser pour Régner
Divide and Conquer
AAC

Sophie Tison
Université Lille 1
Master Informatique 1ère année

Trois paradigmes d'algorithmes

Trois paradigmes d'algorithmes

- Programmation Dynamique

Trois paradigmes d'algorithmes

- Programmation Dynamique
- Algorithmes Gloutons

Trois paradigmes d'algorithmes

- Programmation Dynamique
- Algorithmes Gloutons
- Diviser pour régner

Glouton versus Dynamique

Dans les deux cas on peut se représenter l'ensemble des solutions sous forme d'un arbre, les solutions étant construites incrémentalement et pouvant être vues comme une suite de choix.

Glouton versus Dynamique

Dans les deux cas on peut se représenter l'ensemble des solutions sous forme d'un arbre, les solutions étant construites incrémentalement et pouvant être vues comme une suite de choix.

Programmation dynamique: on parcourt toutes les solutions mais de nombreux noeuds de l'arbre correspondent aux mêmes sous-problèmes et l'arbre peut donc représenté de façon beaucoup plus compacte comme un graphe (DAG: directed acyclic graph).

Glouton versus Dynamique

Dans les deux cas on peut se représenter l'ensemble des solutions sous forme d'un arbre, les solutions étant construites incrémentalement et pouvant être vues comme une suite de choix.

Programmation dynamique: on parcourt toutes les solutions mais de nombreux noeuds de l'arbre correspondent aux mêmes sous-problèmes et l'arbre peut donc représenté de façon beaucoup plus compacte comme un graphe (DAG: directed acyclic graph).

Algorithme glouton: on construit uniquement et directement-sans backtracking- une -et une seule- branche de l'arbre qui correspond à une solution optimale.

Diviser pour Régner: Le principe

- cas de base: savoir résoudre les ”petits” problèmes ‘
- Diviser le problème en sous-problèmes
- Résoudre les sous-problèmes par induction
- Combiner les solutions des sous-problèmes

DIVISER POUR RÉGNER: LE SCHÉMA

```
DivReg (Probleme P)
si P est "petit"
  CasBase(P);
sinon{
 Diviser P en P1, ..Pk;
 Combiner_Solution(DivReg(P1),
                   DivReg(P2),...DivReg(Pk)); }
finsi
```

Exemples?

UN PREMIER EXEMPLE: RECHERCHE DICHOTOMIQUE

Version Récursive:

```
//T trié croissant
//retourne Vrai SSi x  présent dans T[g..d]
boolean RechDico(int x, int g, int d){
if (g>d) return false;
 else {
  int m=(g+d)/2;
  if (T[m]==x) return true;
  else if (T[m] < x) return rec_dicho(x,m+1,d);
  else return rec_dicho(x,g,m-1); }
}
```

UN PREMIER EXEMPLE: RECHERCHE DICHOTOMIQUE

Version Récursive Bis:

```
//T trié croissant
//retourne Vrai SSi x  présent dans T[g..d]
//g<=d
boolean RechDico(int x, int g, int d){
if (g>=d) return (T[g]==x);
 else {
  int m=(g+d)/2;
  if  (T[m] < x) return rec_dicho(x,m+1,d);
  else return rec_dicho(x,g,m); }
}
```

UN PREMIER EXEMPLE: RECHERCHE DICHOTOMIQUE

Version Itérative:

```
//T trié croissant
//T.length >=1
//retourne Vrai SSi n présent dans  T
boolean it_dicho(int x){
int g=0;
int d=T.length-1;
     while (g<d) {
  int m=(g+d)/2;
  if (T[m] < x) g=m+1;
  else d=m;}
  return (T[g]==x);
}
```

UNE PARENTHÈSE SUR LA PREUVE

Version Itérative:

```
//retourne Vrai SSi n présent dans  T
//T trié croissant
//T.length >=1
boolean it_dicho(int x){
int g=0;
int d=T.length-1;
     while (g<d) {
     //Invariant g <=d et
     //  si x présent dans T, T[g]<=x<=T[d]
  int m=(g+d)/2;
  if (T[m] < x) g=m+1;
  else d=m;}
  return (T[g]==x);
}
```

Une Parenthèse sur la preuve

Il faut montrer:

- la correction de l'invariant;
- la terminaison

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de $\log n$,
  - chaque boucle est en

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de $\log n$,
  - chaque boucle est en $O(1)$:
  - l'algorithme est en

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de $\log n$,
  - chaque boucle est en $O(1)$:
  - l'algorithme est en $O(\log n)$

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de $\log n$,
  - chaque boucle est en $O(1)$:
  - l'algorithme est en $O(\log n)$
- Version récursive:
  $A(1) = 1$, $A(n) = 1 + A(n/2)$, si $A(n)$ est le nombre d'appels récursifs:

La complexité?

Soit $n = T.length$.

- Version itérative:
  - le nombre de boucles est de l'ordre de $\log n$,
  - chaque boucle est en $O(1)$:
  - l'algorithme est en $O(\log n)$

- Version récursive:
  $A(1) = 1, A(n) = 1 + A(n/2)$, si $A(n)$ est le nombre d'appels récursifs:
  l'algorithme est en $O(\log n)$

Un autre exemple?

UN AUTRE EXEMPLE: LE TRI FUSION- MERGE SORT

```
//Liste à   trier
Liste triFusion(L){
si L.length<=1
  retourne L; //cas Base
sinon {
  milieu= L.length/2;
  retourne
    fusion(triFusion(L[0..milieu-1]),
           triFusion(T[milieu..L.length-1]));}
finsi}
```

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

Complexité?

Comptons le nombre de comparaisons d'éléments:

La fusion de deux listes triées de k éléments nécessite

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.

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:

$$Comp(n)$$

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

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

Analyse de la Complexité: Méthode 1

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

$$Comp(1) = 0$$

On représente l'arbre (voir tableau!)

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

$\vdots$

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

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

$\vdots$

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

Soit en sommant et en simplifiant:

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

$\vdots$

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

Soit en sommant et en simplifiant:

$Comp(n) <= n * k$

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

$\vdots$

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

Soit en sommant et en simplifiant:

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

Analyse de la Complexité: Méthode 2

Supposons $n = 2^k$.

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

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

$\vdots$

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

$\vdots$

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

Soit en sommant et en simplifiant:

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

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

Le Master Theorem -version allégée

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

- si $d > \log_b a, T(n) = O(n^d)$

Le Master Theorem -version allégée

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

- si $d > \log_b a, T(n) = O(n^d)$
- si $d = \log_b a, T(n) = O(n^d \log n)$

Le Master Theorem -version allégée

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

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

Le Master Theorem -version allégée

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

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

Exemple de la Recherche dichotomique:

Le Master Theorem -version allégée

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

- si $d > \log_b a, T(n) = O(n^d)$
- si $d = \log_b a, T(n) = O(n^d \log n)$
- 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 $a = 1, b = 2, d = 0$

Le Master Theorem -version allégée

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

- si $d > \log_b a, T(n) = O(n^d)$
- si $d = \log_b a, T(n) = O(n^d \log n)$
- 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

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

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

Le Master Theorem pour le Merge Sort

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

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

$Comp(n) = 2 * Comp(n/2) + O(n)$
Donc $a = b = 2, d = 1$

Le Master Theorem pour le Merge Sort

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

- si $d > \log_b a, T(n) = O(n^d)$
- si $d = \log_b a, T(n) = O(n^d \log n)$
- 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 $O(n \log n)$.

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.

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:

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: n^2 opérations élémentaires

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:

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:

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:

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

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 donc un algorithme dont la complexité est donnée par:

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 donc un algorithme dont la complexité est donnée par:
$C(n) = 4 * C(n/2) + 0(n)$

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 donc un algorithme dont la complexité est donnée par:
$C(n) = 4 * C(n/2) + 0(n)$
Master Theorem: $a = 4, b = 2, d = 2 = \log_2 4$

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 donc un algorithme dont la complexité est donnée par:
$C(n) = 4 * C(n/2) + 0(n)$
Master Theorem: $a = 4, b = 2, d = 2 = \log_2 4$
On obtient encore un algorithme en n^2.

Une astuce

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

Une astuce

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

UNE ASTUCE

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

On obtient un algorithme dont la complexité est donnée par:

Une astuce

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

On obtient un algorithme dont la complexité est donnée par:
$C(n) = 3C(n/2) + O(n)$

Une astuce

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

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

UNE ASTUCE

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

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 $O(n^{\log_2 3})$ ($\log_2 3 \approx 1.58$).

UNE ASTUCE

$x * y = 10^n(x_1 * y_1) + 10^{n/2}(x_1 * y_2 + x_2 * y_1) + x_2 * y_2$
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. $|x_1 - y_1| * |y_1 - y_2|$

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.

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.

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: $O(n^2)$

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 la russe (ou égyptienne) plus efficace mais aussi en $O(n^2)$, si on compte les opérations sur les chiffres.

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 la russe (ou égyptienne) plus efficace mais aussi en $O(n^2)$, si on compte les opérations sur les chiffres.
- Algorithme de Karatsuba en $O(n^{\log_2 3})$

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 la russe (ou 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 Tomm-Cook, toujours par "Diviser pour régner" en $O(n^{\log_3 5})$

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 la russe (ou 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 Tomm-Cook, toujours par "Diviser pour régner" en $O(n^{\log_3 5})$
- Algorithme Schoenage-Strassen, basé sur la FFT en $O(n \log n \, log(\log n))$

Un dernier exemple: La paire de points la plus proche

Le problème :
Soit P un ensemble de n points du plan.
On cherche la paire de points la plus proche (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 la plus proche (pour la distance euclidienne).

En dimension 1?

Le problème :
Soient n points d'une droite.
On cherche la paire de points la plus proche (pour la distance euclidienne).

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

En dimension 1?

Le problème :
Soient n points d'une droite.
On cherche la paire de points la plus proche (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

En dimension 1?

Le problème :
Soient n points d'une droite.
On cherche la paire de points la plus proche (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

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.

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.

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 supposée les abscisses des points sont toutes différentes?

DIVISER POUR RÉGNER?

```
Soit P_x la liste des points de P
                              triée par x croissant.
cpt=0;
 pour chaque x  dans  P_x  {
   si cpt <=n/2  ajouter  x  à  G
   sinon ajouter x  à  D
   cpt++;}
```

DIVISER POUR RÉGNER?

```
Soit P_x la liste des points de P
                              triée par x croissant.
cpt=0;
 pour chaque x  dans  P_x  {
   si cpt <=n/2  ajouter  x  à  G
   sinon ajouter x  à  D
   cpt++;}
```

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

DIVISER L'ENSEMBLE DES POINTS EN DEUX

```
Soit P_x la liste des points de P
                              triée par x croissant.
cpt=0;
 pour chaque x  dans  P_x  {
   si cpt <=n/2  ajouter  x  à  G
   sinon ajouter x  à  D
   cpt++;}
```

Coût?

DIVISER L'ENSEMBLE DES POINTS EN DEUX

```
Soit P_x la liste des points de P
                              triée par x croissant.
cpt=0;
 pour chaque x  dans  P_x  {
   si cpt <=n/2  ajouter  x  à  G
   sinon ajouter x  à  D
   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.

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.

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?

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!

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 un 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.
   Trouver , la distance minimale min_{GD} entre un point de G et un point de D
4. La distance minimale est donc le minimum de min_G, min_D, (resp. de min_G, min_D, min_{GD})

Coût du point 4?

RÉCAPITULATIF

1. Trier les points selon x
2. Partager en deux parties de même cardinal (à un 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.
   Trouver si il existe un point de G et un point de D, à distance inférieure à $\delta = min(min_G, min_D)$, la distance minimale min_{GD} entre un point de G et un point de D
4. La distance minimale est donc le minimum de min_G, min_D, (resp. de min_G, min_D, min_{GD})

Coût du point 4?

UN LEMME...

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

UN LEMME...

Remarque: si deux points sont à distance $< \delta$ et de part et d'autre d ela 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 $med - \delta \leq x \leq med + \delta$.

UN LEMME...

Remarque: si deux points sont à distance $< \delta$ et de part et d'autre d ela 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 sont séparés par au moins 15 *points dans la liste, ils sont à distance* $> \delta$.

Remarque: On peut améliorer le lemme .
Preuve: au tableau!

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 un près) par une droite verticale $x = med$: on obtient G, D.
3. Résoudre à gauche et à droite. Cela donne une distance minimale min_G à gauche, min_D à droite.
4. Trouver , 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. de min_G, min_D, min_{GD}).

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 un près) par une droite verticale $x = med$: on obtient 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 $\delta = min(min_G, min_D)$, 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. de min_G, min_D, min_{GD}).

Donc la complexité est donnée par l'équation

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 un près) par une droite verticale $x = med$: on obtient 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 $\delta = min(min_G, min_D)$, 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. de min_G, min_D, min_{GD}).

Donc la complexité est donnée par l'équation
$C(n) = 2C(n/2) + O(n)$

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 un près) par une droite verticale $x = med$: on obtient 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 $\delta = min(min_G, min_D)$, 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. de min_G, min_D, min_{GD}).

Donc la complexité est donnée par l'équation
$C(n) = 2C(n/2) + O(n)$

L'algorithme sera en $O(n \log n)$.

Quelques exemples classiques

- les algorithmes de recherche dichotomique, recherche de la médiane, ..

Quelques exemples classiques

- les algorithmes de recherche dichotomique, recherche de la médiane, ..
- le tri fusion

Quelques exemples classiques

- les algorithmes de recherche dichotomique, recherche de la médiane, ..
- le tri fusion
- les algorithmes de multiplication: entiers (Karatsuba), matrices (Strassen)

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

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

Bilan

- Diviser un problème en problèmes de taille nettement plus petites

Bilan

- Diviser un problème en problèmes de taille nettement plus petites
- Appliquer si possible le *Master Theorem* pour la complexité

Bilan

- Diviser un problème en problèmes de taille nettement plus petites
- Appliquer si possible le *Master Theorem* pour la complexité
- Souvent plus facile à écrire en récursif...même si cela peut se faire en itératif

Bilan

- Diviser un problème en problèmes de taille nettement plus petites
- Appliquer si possible le *Master Theorem* pour la complexité
- Souvent plus facile à écrire en récursif...même si cela peut se faire en itératif
- Peut être intéressant à paralléliser