Introduction à l'algorithmique

Marc Zeitoun

Master MEEF, Prépa CAPES Mathématique, 13/9/2018

Objectif

- Introduction à l'algorithmique.
- Un aperçu de quelques algorithmes.
- Un aperçu de quelques techniques algorithmiques.

L'algorithmique avant l'ordinateur

- −300 ? Algorithme d'Euclide.
- 820 Algorithmes d'Al-Khwârîzmî.
  - Résolution des équations du second degré.

L'algorithmique avant l'ordinateur

- −300 ? Algorithme d'Euclide.
- 820 Algorithmes d'Al-Khwârîzmî.
  - Résolution des équations du second degré.
- 1623 Description d'une machine à calculer de Schickard.
- 1645 Machine à calculer de Pascal.
- 1710 Machine à calculer de Leibniz (+ calcul en base 2).
- 1801 Métier à tisser de Jacquart.
- 1842 Programme de Lovelace, Machine de Babbage.

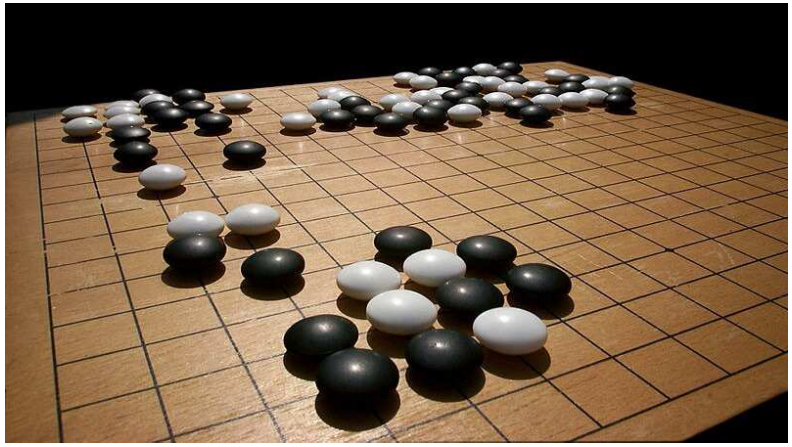
Limites des ordinateurs

- Peut-on résoudre n'importe quel problème avec un ordinateur ?

- Combien de temps faut-il pour résoudre un problème ?
  Combien de mémoire faut-il pour résoudre un problème ?

Des progrès impressionnants

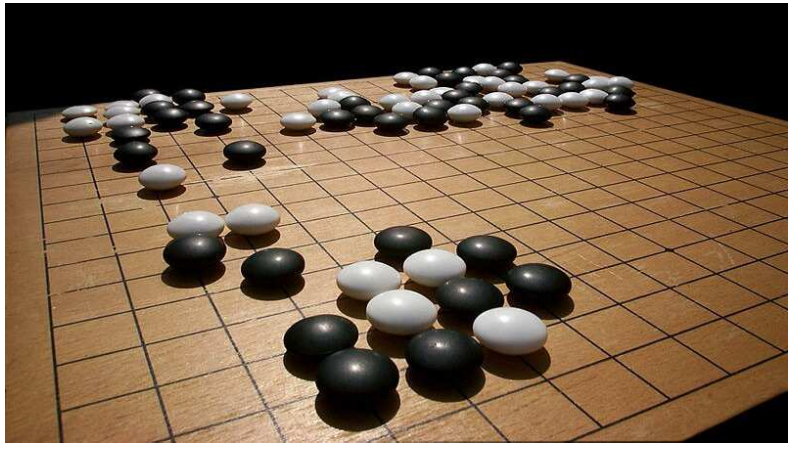
2016 : AlphaGo



Des progrès impressionnants

2016 : AlphaGo

AlphaGo



2015 : Gérard Berry
(12 décembre à Bordeaux !)

Gérard Berry : « L'ordinateur est complètement con »

« Fondamentalement, l'ordinateur et l'homme sont les deux opposés les plus intégraux qui existent. » Entretien avec Gérard Berry, informaticien et professeur au Collège de France, médaille d'or 2014 du CNRS.

Algorithmes

Algorithme : procédure de calcul

- définie par un enchaînement d'opérations simples,
- prenant en entrée une valeur,
- calculant en sortie une valeur.

Un algorithme est conçu pour résoudre un problème.

Exemples de problèmes

Problème 1

Donnée : Un nombre entier positif n en base 2.

Question : n est-il pair ?

Problème 2

D. : Un nombre entier positif n en base 10.

Q. : n est-il premier ?

Problème 3

D. : Un programme en C.

Q. : Le programme est-il syntaxiquement correct ?

Exemples de problèmes (2)

Problème 4

Donnée Un graphe donné par une liste d'adjacence.
Question Le graphe est-il 3-coloriable ?

Problème 5

D. Une grille de Sudoku $n \times n$.
Q. La grille a-t-elle une solution ?

Problème 6

D. Un puzzle Eternity.
Q. Le puzzle a-t-il une solution ?

Exemples de problèmes (2)

Problème 4

Donnée : Un graphe donné par une liste d'adjacence.

Question : Le graphe est-il 3-coloriable ?

Problème 5

D. Une grille de Sudoku $n \times n$.

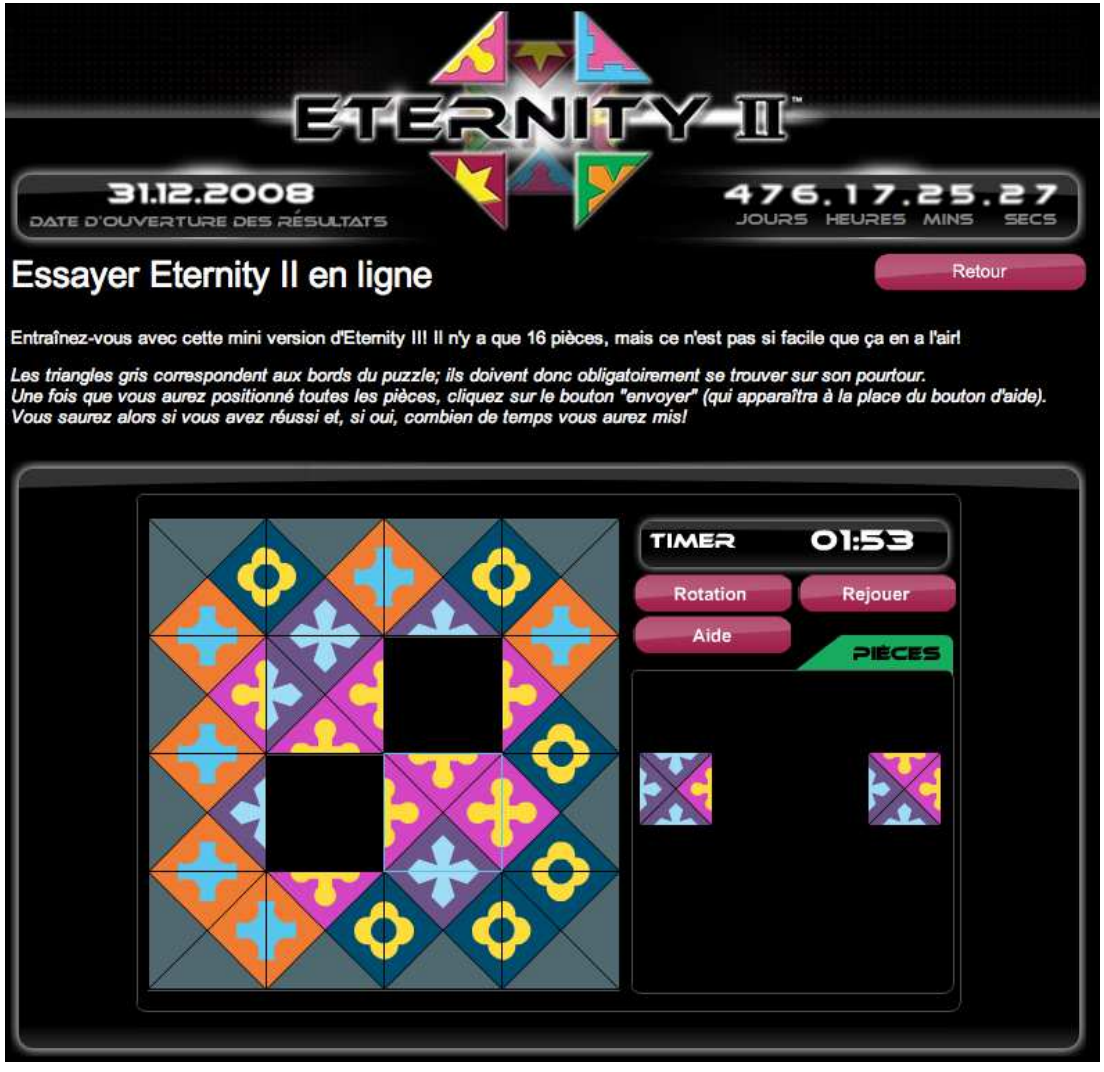
Q. La grille a-t-elle une solution ?

Problème 6

D. Un puzzle Eternity.

Q. Le puzzle a-t-il une solution ?

Problème 6 : Eternity II



Exemples de problèmes (3)

Problème 7

Donnée Un programme.
Question Le programme s'arrête-t-il sur une entrée ?

Problème 8

D. Un programme.
Q. Le programme s'arrête-t-il sur toute entrée ?

Problème 9

D. Des pièces de Tetris $P_1, \ldots, P_n$.
Q. Peut-on paver le plan avec $P_1, \ldots, P_n$?

Problème 10

D. Une équation Diophantienne.
Q. L'équation a-t-elle une solution ?

Exemples de problèmes (3)

Problème 7
Donnée : Un programme.
Question : Le programme s'arrête-t-il sur une entrée ?

Problème 8
D. : Un programme.
Q. : Le programme s'arrête-t-il sur toute entrée ?

Problème 9
D. : Des pièces de Tetris $P_1, \ldots, P_n$.
Q. : Peut-on paver le plan avec $P_1, \ldots, P_n$?

Problème 10
D. : Une équation Diophantienne.
Q. : L'équation a-t-elle une solution ?

Exemples de problèmes (4)

Problème 11

Donnée Un entier n non premier.

Question Une décomposition de n en produit de deux entiers $\neq 1, n$

Problème 12

D. Deux entiers m, n.

Q. Le PGCD de m et n.

Problème 13

D. Une équation $ax^2 + bx + c = 0$, où $a, b, c \in \mathbb{Z}$.

Q. L'équation a-t-elle des solutions dans $\mathbb{R}$, si oui lesquelles ?

Exemples de problèmes (5)

Problème 13

Donnée Une liste d'entiers $a_1, \ldots, a_n$,
Question Une permutation triée $a'_1, \ldots, a'_n$ de ces entiers

Problème 14

D. Une liste de points dans le plan $\mathbb{N} \times \mathbb{N}$.
Q. Une description du plus petit ensemble convexe contenant tous ces points.

Problème 15

D. Une liste de points dans le plan $\mathbb{N} \times \mathbb{N}$.
Q. La distance minimale entre deux points distincts.

Problème 16

D. Un texte (suite de caractères).
Q. Une suite de caractères plus courte représentant le même texte.

Légende pour les problèmes précédents

-  : On connaît un algorithme qui résout le problème de façon efficace (en temps polynomial par rapport à la **taille de l'entrée**).
  Ce n'est pas du tout évident pour le problème de primalité.

Légende pour les problèmes précédents

-  : On connaît un algorithme qui résout le problème de façon efficace (en temps polynomial par rapport à la **taille de l'entrée**).
  Ce n'est pas du tout évident pour le problème de primalité.

-  : On connaît un algorithme, mais le meilleur connu fait un nombre exponentiel d'opérations par rapport à la **taille de l'entrée**.

Légende pour les problèmes précédents

-  : On connaît un algorithme qui résout le problème de façon efficace (en temps polynomial par rapport à la **taille de l'entrée**).
  Ce n'est pas du tout évident pour le problème de primalité.

-  : On connaît un algorithme, mais le meilleur connu fait un nombre exponentiel d'opérations par rapport à la **taille de l'entrée**.

-  : On **sait** qu'il **n'existe pas** d'algorithme (avec la notion actuelle d'algorithme).

Un problème fondamental : l'arrêt

Donnée Un programme P et une entrée x de ce programme.

Question P s'arrête-t-il sur l'entrée x ?

Un problème fondamental : l'arrêt

Donnée Un programme P et une entrée x de ce programme.

Question P s'arrête-t-il sur l'entrée x ?

Turing : il n'existe aucun algorithme pour résoudre ce problème !

Un problème fondamental : l'arrêt

Donnée Un programme P et une entrée x de ce programme.

Question P s'arrête-t-il sur l'entrée x ?

Turing : il n'existe aucun algorithme pour résoudre ce problème !

- Par l'absurde, supposons qu'il existe un tel algorithme, `halt`.

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

- Ce programme `halt` prendrait 2 chaînes de caractères en arguments :
  - un programme `P`, et
  - un argument `x` de `P`.

Pas d'algorithme pour résoudre le problème de l'arrêt

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

Pas d'algorithme pour résoudre le problème de l'arrêt

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

Regardons le programme suivant :

```
def Contradiction(P):
    if halt(P,P):
        while(True):
            print("Je boucle")
    else:
        print("Je m'arrête")
```

Pas d'algorithme pour résoudre le problème de l'arrêt

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

Regardons le programme suivant :

```
def Contradiction(P):
    if halt(P,P):
        while(True):
            print("Je boucle")
    else:
        print("Je m'arrête")
```

Quand on lance `Contradiction(Contradiction)`, 2 cas possibles :

1. Soit `Contradiction(Contradiction)` s'arrête,
   alors `halt(Contradiction,Contradiction)` renvoie `True`,
   donc on passe dans la boucle `while` et on ne s'arrête pas.

Pas d'algorithme pour résoudre le problème de l'arrêt

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

Regardons le programme suivant :

```
def Contradiction(P):
    if halt(P,P):
        while(True):
            print("Je boucle")
    else:
        print("Je m'arrête")
```

Quand on lance `Contradiction(Contradiction)`, 2 cas possibles :

2. Soit `Contradiction(Contradiction)` ne s'arrête pas, alors `halt(Contradiction,Contradiction)` renvoie `False`, donc on ne passe pas dans la boucle `while` et on s'arrête.

Pas d'algorithme pour résoudre le problème de l'arrêt

```
def halt(P, x):
    # Renvoie
    # True si P s'arrête sur l'entrée x
    # False sinon.
```

Regardons le programme suivant :

```
def Contradiction(P):
    if halt(P,P):
        while(True):
            print("Je boucle")
    else:
        print("Je m'arrête")
```

Quand on lance `Contradiction(Contradiction)`, 2 cas possibles.

– Dans les 2 cas, il y a une contradiction.

– Donc l'algorithme `halt` **ne peut pas exister**.

Algorithmes

Algorithme : procédure de calcul

- définie par un enchaînement d'opérations simples,
- prenant en entrée une valeur,
- calculant en sortie une valeur.

Un algorithme est conçu pour résoudre un problème.

Un algorithme provient toujours d'une **solution** au problème.

Éléments des algorithmes

Très peu d'instructions.

Exécution séquentielle.

- Variables et affectation,
- Opérations d'entrée–sortie,
- Structures conditionnelles,
- Structures répétitives.

Qualité des algorithmes

Pour être utilisable, un algorithme doit être

- correct,
- efficace en temps de calcul,
- le moins gourmand possible en mémoire.

la correction d'un algorithme ne peut pas être testée automatiquement.

Compromis temps de calcul / espace mémoire utilisé.

Notion de complexité

Pour mesurer l'efficacité en temps d'un algorithme, on évalue le nombre d'opérations élémentaires

- sur une entrée de taille n, en fonction de n,
- dans le pire cas.

Le temps de calcul doit être fini !

Efficacité

Temps de calcul, sur un ordinateur 1GHz, pour

- Des complexités en n, $n\log_{10}(n)$, n^2, 2^n,
- et des tailles de problèmes de $n = 10$, 100, 1000, 10^6, 10^9.

	n	$n\log_2(n)$	n^2	2^n
$n = 10$	✓	✓	✓	✓
$n = 100$	✓	✓	✓	$> 10^{13}$ ans ✗
$n = 1000$	✓	✓	✓	$> 10^{280}$ ans ✗
$n = 10^6$	✓	✓	16mn	✗
$n = 10^9$	1s	9s	> 31 ans ✗	✗

✓ = moins de 10^{-2}s

Âge estimé de l'univers : $< 5.10^9$ ans

Algorithmique et programmation

- On utilise souvent un pseudo-langage pour présenter les algorithmes.
- Certains langages de programmation ont une syntaxe proche.
- **Exemple** : Python.

Variables et affectation

- Variable = emplacement mémoire
  - où on mémorise une valeur,
  - d'un type donné (entier, chaîne, liste, Booléen).
  - accessible par un nom.

Variables et affectation

- Variable = emplacement mémoire
  - où on mémorise une valeur,
  - d'un type donné (entier, chaîne, liste, Booléen).
  - accessible par un nom.
- Même rôle qu'une mémoire dans une calculatrice.

Variables

- A chaque instant, une variable contient une seule valeur.
- Une variable a un nom qui commence par une lettre.
- Exemple : x, x1, i, resultat.

Variables

- A chaque instant, une variable contient une seule valeur.
- Une variable a un nom qui commence par une lettre.
- Exemple : x, x1, i, resultat.

- Choisir des noms explicites pour les variables importantes.
- Les paramètres d'un algorithme sont aussi vus comme des variables.
- Un algorithme peut utiliser des variables supplémentaires.
- Indiquer clairement les variables utilisées par vos algorithmes.

Types

- Ensemble de valeurs,
- et d'opérations légales sur les éléments du type.

Affectation, syntaxe pseudo-code

- Affectation : mémorisation d'une valeur dans une variable.
- Notation pseudo-code :

  $\langle \texttt{nom\_variable} \rangle \leftarrow \langle \texttt{valeur} \rangle$
- **Exemples**

  $\texttt{resultat} \leftarrow 3$

  $\texttt{resultat} \leftarrow \texttt{resultat} + 1$

  La $2^{\text{ème}}$ affectation suppose que **`resultat`** contient déjà une valeur.

  Dans ce cas, son effet est d'incrémenter la variable **`resultat`**.

Affectation, syntaxe Python

- Affecter une valeur à une variable se fait avec le symbole =.
- Ce symbole n'a donc **pas** la même signification qu'en mathématiques.
- `x = 3` signifie : ranger 3 dans la variable x.

  L'ancienne valeur de x est perdue.

Conditionnelles

Pseudo-code

```
si ⟨condition⟩ alors
  ⟨instruction 1⟩
  ⟨instruction 2⟩
  ...
fin si
```

Python

```
if condition:
    instruction1
    instruction2
    ...
```

Note : en python, les blocs sont délimités par l'indentation..

- La condition est évaluée.
- Si elle est vraie, les instructions de la conditionnelle sont exécutées.
- Sinon, on passe à l'instruction suivante.

Conditionnelles

On peut ajouter une partie « sinon ».

Pseudo-code

```
si ⟨condition⟩ alors
  ⟨instruction 1⟩
  ⟨instruction 2⟩
  ...
sinon
  ⟨instruction 3⟩
  ⟨instruction 4⟩
  ...
fin si
```

Python

```
if condition:
    instruction1
    instruction2
    ...
else:
    instruction3
    instruction4
    ...
```

- La condition est évaluée.
- Si elle est vraie, les instructions 1, 2, ...sont exécutées.
- Sinon, , les instructions 3, 4, ...sont exécutées.

Expressions booléennes

Les conditions sont des expressions booléennes.

Pseudo-code	Python
`Vrai`	`True`
`Faux`	`False`

que l'on peut obtenir par des comparaisons

`<`, `>`, `==`, `!=`, `<=`, `>=`

Pour me pas confondre l'affectation et l'égalité, on note l'égalité `==`.

Expressions booléennes

Les booléens se combinent par les opérateurs logiques usuels.

Pseudo-code	Python
`Et`	`and`
`Ou`	`or`
`Non`	`not`

L'évaluation d'une expression Booléenne se fait

- de gauche à droite,
- s'arrête dès le résultat connu.

Pas de risque de division par 0 :

```
if (a != 0) and (b/a != x):
    ...
```

Répétition : boucle « pour tout »

Pseudo-code

```
pour n ← 0 à k faire
    ⟨instruction 1⟩
    ⟨instruction 2⟩
    ...
fin pour
```

Python

```
for n in range(k+1):
    instruction1
    instruction2
    ...
```

- La variable n prend successivement les valeurs 0, 1, ... k.
- Pour chaque valeur, les instructions sont exécutées

Répétition : boucle « tant que »

Pseudo-code

tant que ⟨condition⟩ **faire**
  ⟨instruction 1⟩
  ⟨instruction 2⟩
  ...
fin tant que

Python

```
while condition:
    instruction1
    instruction2
    ...
```

1. La condition est évaluée.
2. Si elle est fausse, on passe à l'instruction qui suit la boucle.
3. Si elle est vraie,
   - Les instructions de la boucle sont exécutées.
   - On revient en 1.

Rupture de boucle

On peut

- sortir d'une boucle depuis le bloc d'instructions :

```
# Sort de la boucle courante
break
```

- Revenir tester la condition sans finir le bloc d'instructions :

```
# Revient tester la condition,
# sans finir le bloc d'instructions
continue
```

Fonctions

En python, les algorithmes sont encapsulés dans des fonctions.

Une fonction est

- d'abord définie,
- puis appelée (utilisée).

Une fonction travaille à partir de paramètres.

```
def delta(a,b,c):
    return b*b - 4*a*c
```

Valeur retournée

Une fonction calcule habituellement une valeur.

On indique la valeur retournée ainsi :

Pseudo-code	Python
`Retourner` ⟨`valeur`⟩	`return resultat`

- L'effet est de faire terminer la fonction.
- Si la fonction est utilisée par un autre algorithme, la valeur retournée peut être utilisée. **Ex : utilisation de la fonction delta :**

```
if delta(a,b,c) < 0:
    ...
```

- Attention : même à l'intérieur d'une bouclel, return termine la fonction.

Récursivité

Exemple : calcul de $n! = 1 \times 2 \times \ldots \times n$.

```
def fact1(n):
  res = 1
  for i in range(1,n+1):
    res = i*res
  return res
```

$$0! = 1$$

$$n! = n \times (n-1)! \quad \text{si } n > 0$$

Récursivité

Exemple : calcul de $n! = 1 \times 2 \times \ldots \times n$.

```
def fact1(n):
  res = 1
  for i in range(1,n+1):
    res = i*res
  return res
```

$$0! = 1$$

$$n! = n \times (n-1)! \quad \text{si } n > 0$$

```
def fact_rec(n):
  if n <= 0:
    return 1
  return n * fact_rec(n-1)
```

Complexité

- Complexité d'un algorithme : nombre d'opérations « élémentaires » effectuées sur une entrée de taille n, dans le cas le pire.
- C'est donc une fonction de $\mathbb{N}$ dans $\mathbb{N}$.
- Ce qui importe est l'ordre de grandeur de ces fonctions.

Ordre de grandeur, notation $O()$

- $f = O(g)$: pour n assez grand, f est majorée par $c.g$ pour $c > 0$.

$$f = O(g) \iff \exists K \in \mathbb{N}, \exists c > 0, \forall n \; n > K \Rightarrow f(n) \leqslant cg(n).$$

Exemples

- $42n^7 + 2017n^4 + 1111n^3 + 2 = O(n^7)$
- $42n^7 + 2017n^4 + 1111n^3 + 2 = O(n^8)$
- $n^{1000} = O(1.1^n)$
- $n \log(n) = O(n^2)$

Recherche d'un élément

- Un tableau est une suite de valeurs de même type rangées séquentiellement en mémoire.
- Les éléments d'un tableau à n éléments sont `T[0]`, `T[1]`, ..., `T[n-1]`.

Problème

Entrée un tableau `T` d'entiers, trié, sa taille `n` et un entier `x`.

Sortie un indice `i` tel que `T[i]==x`.
-1 s'il n'y a pas de tel indice.

Recherche dichotomique

- Algorithme naïf : on teste chacune des n cases.
  - Complexité $O(n)$.
  - On n'exploite pas le fait que le tableau est trié.

Recherche dichotomique

- Algorithme naïf : on teste chacune des n cases.
  - Complexité $O(n)$.
  - On n'exploite pas le fait que le tableau est trié.
- Meilleur algorithme ? Complexité ?

Exercices

- Présenter le calcul du PGCD de deux entiers.
- Présenter le calcul d'une suite récurrente linéaire donnée par u_0, u_1, a, b et la relation

$$u_{n+2} = au_{n+1} + bu_n \qquad (n \geqslant 0)$$

- Évaluer la complexité de votre algorithme.
- Application : algorithme donnant une valeur approchée de $\log_2(n)$.

Récursivité et complexité

Calcul de 2^n.

- En utilisant $2^n = 2^{n-1} + 2^{n-1}$
- En utilisant $2^n = 2 \times 2^{n-1}$
- En utilisant $2^n = \begin{cases} (2^{\lfloor n/2 \rfloor})^2 & \text{si n est pair} \\ 2 \times (2^{\lfloor n/2 \rfloor})^2 & \text{si n est impair} \end{cases}$

Récursivité et complexité

```
def pow2(n):
    if n <= 0:
        return 1
    return pow2(n-1) + pow2(n-1)
```

Récursivité et complexité

```
def pow2(n):
    if n <= 0:
        return 1
    return 2*pow2(n-1)
```

Récursivité et complexité

```
def pow2(n):
    if n <= 0:
        return 1
    a = pow2(n//2)
    if n % 2 == 0:
        return a*a
    else:
        return 2*a*a
```

Récursivité et complexité

Calcul du $n^{\text{ème}}$ terme de la suite de Fibonacci.

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_n &= f_{n-1} + f_{n-2} \quad \text{si } n \geqslant 2
\end{aligned}
$$

Écrire

- une version itérative de complexité $O(n)$,
- une version récursive de complexité $2^{O(n)}$,
- une version récursive de complexité $O(n)$,
- une version récursive de complexité $O(\log n)$.

Diviser pour régner

Méthode récursive pour résoudre un problème :

1. Découper le problème en plusieurs sous-problèmes analogues.
2. Résoudre récursivement ces sous-problèmes.
3. Recombiner les solutions sur les sous-problèmes pour obtenir une solution du problème.

Exemple : recherche dichotomique.

Master theorem

Soit $f : \mathbb{N} \to \mathbb{N}$. Si pour tout $n \in \mathbb{N}$, on a

$$f(n) = af(\lfloor n/b \rfloor) + O(n^c)$$

avec $a, b > 1$, $c \geqslant 0$ alors

- Si $\log_b a > c$, on a $f(n) = O(n^{\log_b a})$
- Si $\log_b a = c$, on a $f(n) = O(n^c \log(n))$
- Si $\log_b a < c$, on a $f(n) = O(n^c)$

Notes :

- $\lfloor n/b \rfloor$ peut être remplacé par $\lceil n/b \rceil$.
- On utilise souvent $f(n) \leqslant af(\lfloor n/b \rfloor) + O(n^c)$.

Master theorem : exemples

- Recherche dichotomique : $T(n) = T(n/2) + O(1)$.
- Tri fusion : $T(n) = T(n/2) + O(n)$.
- Algorithme de multiplication matricielle par blocs.
- Algorithme de multiplication matricielle de Strassen.
- Multiplication naïve de 2 nombres à n chiffres : $O(n^2)$.
- Algorithme de Karatsuba.

Algorithmes de tri à connaître

- Tri à bulle.
- Tri par insertion.
- Tri par sélection.
- Tri par fusion.
- Tri rapide.

Tri par insertion

Suppose: $n =$ taille de t

▷ La première case de t est $t[1]$

TRI-INSERTION(t, n)

pour $j \leftarrow 2$ à n **faire**

  $clé \leftarrow t[j]$

  $i \leftarrow j - 1$

  tant que $i > 0$ et $t[i] > clé$ **faire**

    $t[i+1] \leftarrow t[i]$

    $t[i] \leftarrow clé$

    $i \leftarrow i - 1$

  fin tant que

fin pour

Tri fusion

Principe :

- Trier récursivement
  - la moitié gauche,
  - la moitié droite

  du tableau.
- Fusionner les 2 moitiés déjà triés en un tableau trié.

Tri fusion : complexité

Le nombre d'affectations $c(n)$ pour un tableau de taille n vérifie une récurrence.

Laquelle ?

Tri fusion : complexité

Le nombre d'affectations $c(n)$ pour un tableau de taille n vérifie une récurrence.

Laquelle ?

$$c(n) = 2.c(n/2) + O(n)$$

Tri fusion : complexité

Le nombre d'affectations $c(n)$ pour un tableau de taille n vérifie une récurrence.

Laquelle ?

$$c(n) = 2.c(n/2) + O(n)$$

Master theorem :

$$c(n) = O(n.log(n))$$

Algorithmes de tri : borne inférieure

- Pour tout algorithme de tri procédant par comparaisons, il existe au moins une entrée de taille n sur lequel le tri effectue au moins

$$\lfloor n \ln n \rfloor - n \text{ comparaisons}$$

Algorithmes de tri : borne inférieure

- Pour tout algorithme de tri procédant par comparaisons, il existe au moins une entrée de taille n sur lequel le tri effectue au moins

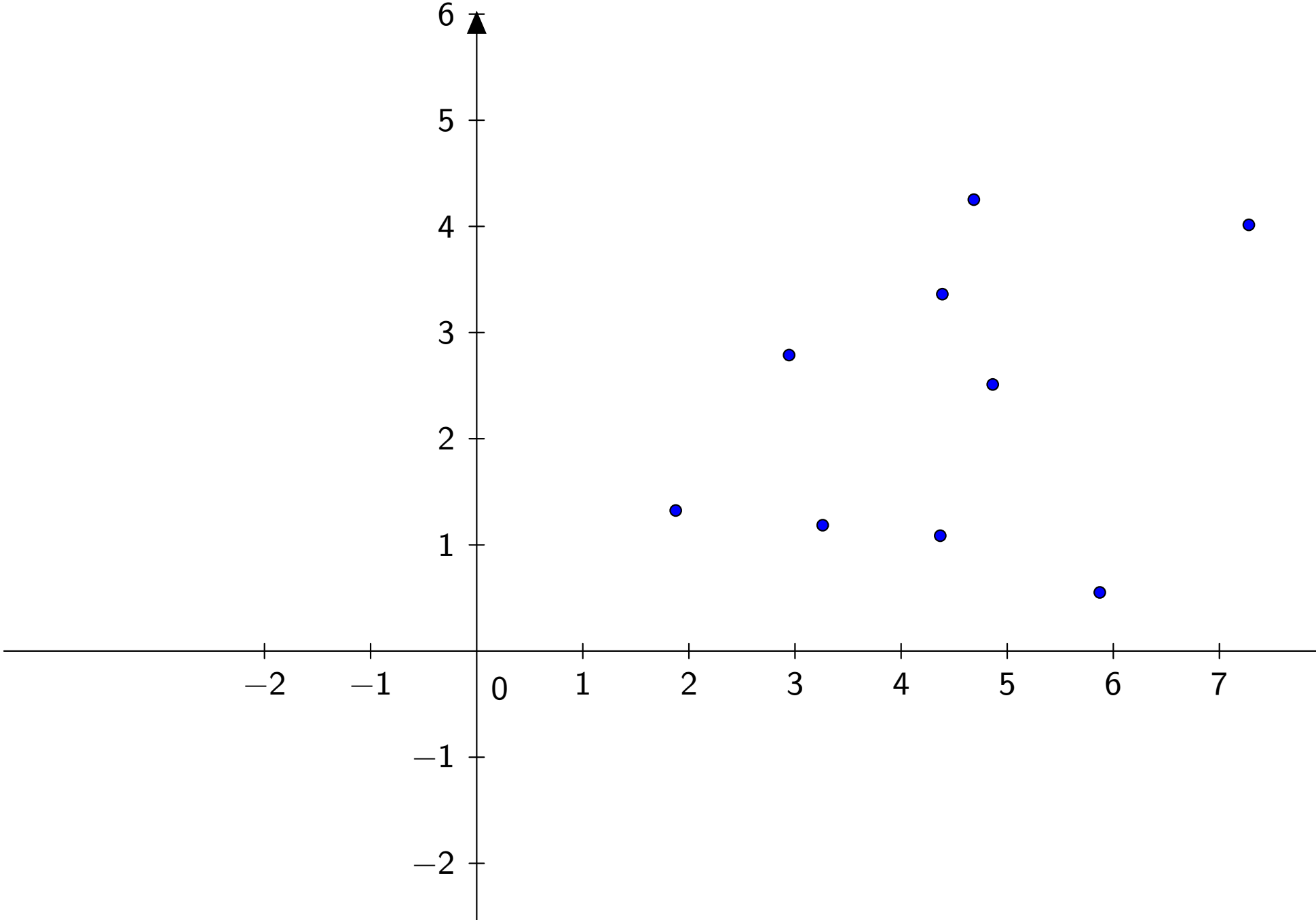
$$\lfloor n \ln n \rfloor - n \text{ comparaisons}$$

- Le tri fusion est donc optimal.
- Cf. Éléments d'algorithmique, D. Beauquier, J. Berstel, Ph. Chrétienne, `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`.

Exercice

- Preuve que l'algorithme du tri par sélection est correct.
- Complexité dans le cas le pire.

Points les plus proches



Points les plus proches

D. Une liste de points dans le plan $\mathbb{N} \times \mathbb{N}$.

Q. La distance minimale entre deux points distincts.

2 algorithmes

- Algorithme naïf : quelle complexité ?
- Algorithme amélioré, diviser pour régner : $O(n \log n)$.

Diviser pour régner

D. Deux entiers a, b de n chiffres chacun.

Q. Le produit ab.

2 algorithmes :

- Algorithme naïf : quelle complexité ?
- Algorithme de Karatsuba : $O(n^{1.59})$.

Multiplication d'entiers

- Objectif : multiplier 2 nombres de n chiffres.
- Opération élémentaire : multiplication de 2 chiffres.
- Multiplication naïve : combien de multiplications ?

Algorithme naïf

- Objectif : multiplier 2 nombres de n chiffres.
- $k = n/2$.
- Algorithme récursif :

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

- Nombre de multiplications : $M(n) = 4M(\lceil n/2 \rceil)$
- Master theorem ?

Algorithme de Karatsuba

- Objectif : multiplier 2 nombres de n chiffres.
- $k = n/2$.
- Algorithme récursif :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (a - b)(c - d) \times 10^k + bd$$

- Nombre de multiplications : $M(n) = 3M(\lceil n/2 \rceil)$
- Exercice : résoudre cette récurrence (master theorem).

Pour aller plus loin

- Éléments d'algorithmique. D. Beauquier, J. Berstel, Ph. Chrétienne
  `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`
- Types de données et algorithmes. Ch. Froidevaux, M-C. Gaudel, M. Soria
  `https://www.lri.fr/~mcg/PDF/FroidevauxGaudelSoria.pdf`