NFP136- Cours 2
ALGORITHMES ET COMPLEXITÉ

PLAN

- Définition d'un algorithme
- Un exemple
- Présentation des algorithmes
- Évaluation d'un algorithme
- Complexité

DÉFINITION D'UN ALGORITHME

- **Procédure de calcul bien définie qui prend en entrée un ensemble de valeurs et produit en sortie un ensemble de valeurs.** (Cormen, Leiserson, Rivert)

- **Ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, il retourne une réponse après un nombre fini d'opérations.** (Beauquier,Berstel,Chrétienne)

- Phase 1 **ENONCÉ DU PROBLÈME**

  Spécification
- Phase 2 **ÉLABORATION DE L'ALGORITHME**
- Phase 3 **VÉRIFICATION**
- Phase 4 **MESURE DE L'EFFICACITÉ**

  Complexité
- Phase 5 **MISE EN OEUVRE**

  Programmation

UN EXEMPLE
Le tri par sélection ordinaire

ÉNONCÉ DU PROBLÈME

- **Entrée:**

  Un tableau d'entiers

- **Sortie:**

  Le même tableau, trié par ordre croissant

L'ALGORITHME- Tri sélection

Principe

tant que il reste plus d'un élément non trié faire

- chercher le plus petit parmi les non triés ;

- échanger le premier élément non trié avec le plus petit trouvé;

fait;

EXEMPLE (suite)

éléments triés éléments non triés				
7	8	15	**5**	10
5	8	15	**7**	10
5	7	15	**8**	10
5	7	8	15	**10**
5	7	8	10	**15**
5	7	8	10	15

Tri sélection- plus détaillé

```
tri-selec (tableau  A de n entiers)
début
  pour  i = 0 à n-2   faire
    // les éléments de 0 à i-1 sont déjà triés
    // i : place où on doit mettre le suivant plus petit
    // recherche de la place J du plus petit des non triés
      j = i;
      pour  k = i+1 à n-1   faire
            si  A[k] < A[j]  alors
                j = k ;
            finsi ;
      fait;
      échanger (A[i] , A[j])  ;
  fait;
fin
```

Tri sélection- en Java

```
public static void tri_selection (int[] A){
  int temp, j;
  for (int i=0; i< A.length-1; i++ ){
      j = i;
      for( int k=i; k<=A.length-1; k++ ){
            if( A[k]<A[j] ) j = k;
      }
      //echange
      temp=A[j]; A[j]=A[i]; A[i]=temp;
  }
}
```

Evaluation des algorithmes

Complexité des algorithmes

ÉVALUATION D'UN ALGORITHME

EFFICACITÉ

- **TEMPS D'EXÉCUTION**
- **MÉMOIRE OCCUPÉE**

Non mesurable par des tests

EXEMPLE

n entier est-il premier ?

n = 1148 ? non, divisible par 2

n = 1147 ? ??? (31 * 37)

NOMBRE D'OPÉRATIONS EN FONCTION DU NOMBRE DES DONNÉES

- **PIRE DES CAS**
- **EN MOYENNE**

EXEMPLES

- **VOYAGEUR DE COMMERCE (N VILLES)**

  Données: distances entre chaque paire de villes

  Nombre de données = N^2

  Enumération et évaluation des (N-1)! tournées possibles et sélection de la plus économique

- **MIN DE N^2 NOMBRES qui représentent des distances entre N villes**

  Même nombre de données: $M=N^2$

Temps de calcul si 10^{-9} secondes par comparaison

N	10	20	30	40	50
Taille (M=) N^2	100	400	900	1600	2500
Min de N^2 nombres	0,1 µs	0,4 µs	0,9 µs	1,6 µs	2,5 µs
Min de (N-1)! nombres	363 µs	>3 an	10^{16} ans	10^{30} ans	10^{45} ans

OPERATIONS "ÉLÉMENTAIRES"

- **On compte les macro-opérations**

  + - * / tests (> ≤) et ou ...

TAILLE D'UN PROBLÈME
=
NOMBRE DE DONNÉES

Codage raisonnable des données

EXEMPLE (ajout, sous condition, d'éléments du tableau B à des éléments du tableau A ; on suppose que les 2 tableaux ont plus de m+n éléments)

AjoutTab (….)

```
début
   pour  i = 0 à m-1    faire
                A(i) =B(i) ;                                  1
    m           pour  j = i à n+i-1    faire
                  n       si  A(i) < B(j)  alors              1
   fois         fois              A(i) =A(i)+B(j) ;            2
                          finsi ;
                fait;
   fait;
fin
```

nombre d'opérations

Pire des cas, test A(i)<B(j) toujours vrai

m(1+ 3n)= 3nm+m opérations

Meilleur cas, test A(i)<B(j) toujours faux

m(1+ n)= nm+m opérations

EXEMPLE (suite)

tri-selection

```
début
  pour  i = 0 à n-2   faire
        j = i ;                                  1
        pour  k = i+1 à n-1   faire
          si  A[k] < A[j]  alors                 1
              j = k ;                            1
          finsi ;
        fait;
        échanger (A[i] , A[j]) ;                 3
  fait;
fin
```

n-i-1 fois

Pour chaque i,

1+2(n-i-1)+3 opérations

nombre d'opérations *(pire des cas)*

$$4(n-1)+2\sum_{i=0}^{n-2}(n-i-1)=4(n-1)+2\sum_{i=1}^{n-1}i=$$

4 (n-1)+ n(n-1) = n^2 +3n - 4

Rappel : somme des n-1 premiers entiers = n(n-1)/2

Algorithme "efficace"

=

Algorithme polynomial

EXEMPLE (suite): $\mathbf{n^2 + 3n - 4}$

Problèmes "intraitables"

→

Théorie de la complexité

EXEMPLE: **voyageur de commerce**

COMPLEXITE DES ALGORITHMES

D_n = {données de taille n}

$coût_A$ (d) = coût de l'algorithme A sur la donnée d

- **PIRE DES CAS**

$$Max_A(n) = \max_{d \in Dn} coût_A(d)$$

- borne sup du coût
- assez facile à calculer
- souvent réaliste

- **COMPLEXITE EN MOYENNE**

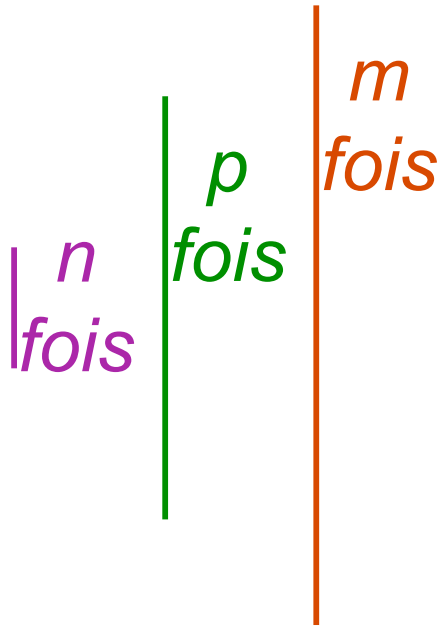
p(d) = probabilité de la donnée d

Moy_A (n) = $\sum_{d \in D_n} p(d) * coût_A(d)$

- **connaître la distribution des données**
- **souvent difficile à calculer**
- **intéressant si comportement usuel de l'algorithme éloigné du pire des cas**

EXEMPLE : produit de 2 matrices

```
static float[][] produitMatrices
                    ( float[][] A,float[][] B ){
   //A[m,n] et B[n,p] résultat C[m,p]
   float[][] C = newfloat[A.length][B[0].length];
   for(int i=0;i< A.length;i++){
       for(int j=0;j< B[0].length;j++){
             float s=0;
             for(int k=0;k< B.length;k++)
                   s=s+A[i][k]*B[k][j];
             C[i][j]=s;
       }
   }
   return C;
}
```



pire cas = cas moyen=mp(2n+2) opérations

NOTATIONS DE LANDAU

f , g fonctions $\mathbb{N} \rightarrow \mathbb{N}$

- **Notation O**

f = O(g) *lire : " f est en O de g "* **⇔**

Il existe $n_0 \in \mathbb{N}$ et c constante t.q. $f(n) \leq c\, g(n)$

pour tout $n \geq n_0$

EXEMPLE

algo A : $f(n) = 4n^3+2n+1$ opérations

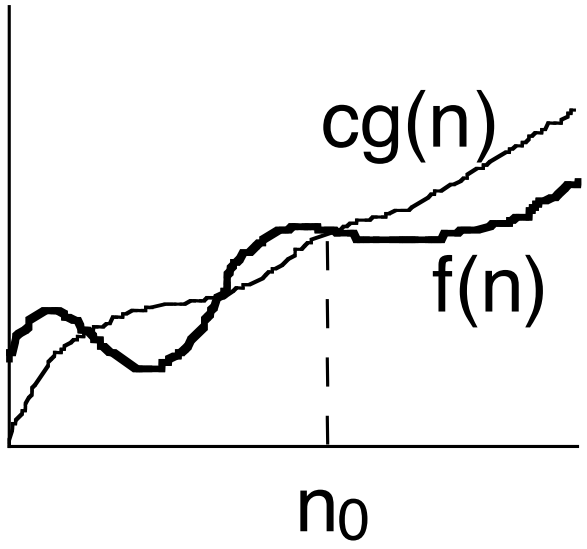
- O

pour tout $n \geq 1$: $f(n) \leq 7n^3$

$n_0=1$ $c=7$ $g(n)=n^3$

donc $f(n) = O(n^3)$

Notation O → majorant du nombre d'opérations d'un algorithme



f = O(g(n))

Complexité →

- **comportement des algorithmes quand le nombre de données augmente**
- **comparaison entre algorithmes**

Algorithme polynomial

$f(n) = a_0 n^p + a_1 n^{p-1} + \ldots + a_{p-1} n + a_p$

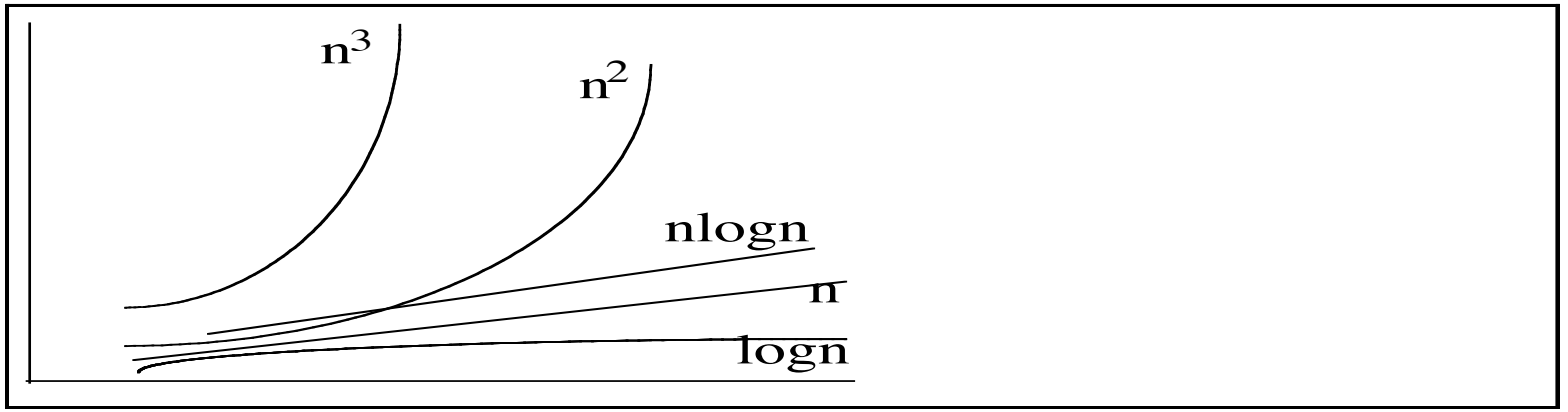
$f = O(n^p)$

EXEMPLE (fin) $f(n) = n^2 + 3n - 4$

Le tri-selection est un algorithme polynomial en $O(n^2)$

“Bonne complexité”

O(log n) ou O(n) ou O(n log n)



Allure de quelques courbes

n = 10^6 **1 μs par opération**

$\log_2 n$	n	$n\log_2 n$	n^2	n^3
20μs	1 s	20 s	12 j	32 Ka

De l'effet des progrès de la technologie sur la résolution des problèmes

	taille max des problèmes traitables en 1 heure avec un ordinateur :		
nombre d' opération s	actuel (le plus rapide)	100 fois plus rapide	1000 fois plus rapide
N	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2 (soit N_2^2 opérations)	$10\ N_2$	$31{,}6\ N_2$
n^5	N_3 (soit N_3^5 opérations)	$2{,}5\ N_3$	$3{,}98\ N_3$
2^n	N_4 (soit 2^{N4} opérations)	$N_4 + 6{,}64$	$N_4 + 9{,}97$
3^n	N_5 (soit 3^{N5} opérations)	$N_5 + 4{,}19$	$N_5 + 6{,}29$

Même si la puissance des ordinateurs augmente, certains "gros" problèmes ne pourront sans doute jamais être résolus

Une illustration de l'amélioration de complexité : recherche séquentielle et recherche dichotomiqe d'un élément dans un tableau trié

Le problème

Etant données :

- Un tableau T de n entiers triés par ordre croissant
- Un entier x

Ecrire un algorithme qui teste si x appartient à T :
recherche de x dans T

Recherche séquentielle

Idée de l'algorithme :

- Parcourir le tableau T. S'arrêter :

  . soit parce qu'on trouve x

  . Soit parce qu'on trouve un élément > x

On a immédiatement une idée de la complexité au pire :

Au pire on parcourt les n cases du tableau en faisant pour chaque case un nombre constant de comparaisons donc complexité **O(n)**

Recherche séquentielle- en java

```
static boolean recherche_lin (int x, int [] T) {
  int i=0;
  boolean rep=false;
  for ( ; i < T.length && (rep==false); i++)  {
      if (T[i]==x)  {rep=true;}
      if (T[i]>x)  { break;}
  }
   System.out.println("Nombre d'iterations "+(i+1));
   return(rep);
}
```

Recherche dichotomique

Idée de l'algorithme :

- Si x est dans T alors il serait dans un intervalle d'indices [d..f[; d initialisé à 0 et f après la fin de T
- On compare x par rapport à T[m] où m est **l'indice du milieu du tableau**
  - Si x = T[m] on s'arrête car on a trouvé x
  - Si T[m] < x on en déduit que x ne peut pas se trouver dans l'intervalle d..m ; on « bouge » d
  - Sinon (T[m] >x) on en déduit que x ne peut pas se trouver dans l'intervalle m..f ; on « bouge » f
- On s'arrête soit parce qu'on a trouvé x soit parce que l'intervalle [d..f[devient vide

Recherche dichotomique- en java

```
static boolean recherche_dicho (int x, int [] T) {
    int d=0;
    int f = T.length;
    //si x est dans le tableau, il le serait dans [d..f[ (f exclu)
    int  m; boolean trouve = false;
    int cptr=0;
    while  ((trouve == false) && (d<f)){
              cptr++;
              m = (d+f)/2;
              if (T[m]==x) trouve=true;
              else if (T[m] < x) d=m+1; else f=m;
    }
    System.out.println("Nombre d'iterations "+cptr);
    return(trouve);
}
```

Complexité en O(log n)

Soit k le nombre de passages dans la boucle while.

on divise le nombre d'éléments restants par 2

jusqu'à ce qu'il n'en reste qu'un (k divisions)

((((n/2)/2)/2)/../2)=1

Donc $n/2^k = 1$ et ainsi $k=\log_2 n$

Comparaison expérimentale

- On génère aléatoirement un tableau trié de n éléments (T[0] dans [0,10] et T[i] dans [T[i-1], T[i-1]+10])
- on recherche ensuite un même élément par la méthode séquentielle puis par la méthode dichotomique

Comparaison expérimentale

```
G:\NFP136JAVA\javapourVari>java dicho
Entrez le nombre d'elements -> 1000
On va maintenant faire des recherches
Entrez un element a rechercher -> 236
----Recherche lineaire
Nombre d'iterations 50
236 apparait dans le tableau
----Recherche dichotomique
Nombre d'iterations 10
236 apparait dans le tableau
```

Comparaison expérimentale

```
G:\NFP136JAVA\javapourVari>java dicho
Entrez le nombre d'elements -> 10000000
On va maintenant faire des recherches
Entrez un element a rechercher -> 456987
----Recherche lineaire
Nombre d'iterations 101533
456987 n'apparait pas dans le tableau
----Recherche dichotomique
Nombre d'iterations 24
456987 n'apparait pas dans le tableau
```

log(10000000) ≈ 24

REMARQUES PRATIQUES

- comptage grossier des opérations (+ - / * tests …) mais attention aux boucles
- ce qui est à l'extérieur des boucles est souvent négligeable
- on écrit O(logn) au lieu de $O(\log_2 n)$
- hiérarchie (n > 1)

  logn < n < nlogn < n2 < n3 <<