COURS ALGORITHMES ET COMPLEXITÉ

Evaluation et complexité des algorithmes

ÉVALUATION D'UN ALGORITHME

EFFICACITÉ D'UN ALGORITHME :

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

EXEMPLE DE PROBLEME A RESOUDRE

n entier est-il premier ?

n = 1148 ? non, divisible par 2

n = 1147 ? ??? (en fait, 1147=31*37)

UN SECOND EXEMPLE

- **PARTITION DE N ENTIERS**
  - Données : N entiers positifs
  - Donc taille des données = nombre d'entiers = N
  - Problème : existe-t-il un sous-ensemble de ces entiers dont la somme est exactement la moitié de la somme totale des N entiers ?

  → Par exemple (somme=310) : ***64***, ***36***, ***7***, ***10***, 2, ***15***, ***23***, 74, 49, 30

  - Algorithme pour résoudre le problème ?
    - Enumérer les 2^N sous-ensembles possibles,
    - Evaluer la somme des entiers dans chacun d'entre eux, et conserver celui (ou ceux) qui convien(nen)t, s'il(s) existe(nt).

Temps de calcul si 10^{-9} secondes par opération élémentaire (1 GHz)

N	10	20	50	100	200
Somme de N nombres	0,01 µs	0,02 µs	0,05 µs	0,1 µs	0,2 µs
Enumération de 2^N objets	1 µs	1 ms	>10 jours	>30 000 milliards d'années	>3.10^{43} années

EVALUER LE TEMPS D'EXECUTION = EVALUER LE NOMBRE D'OPÉRATIONS A EXECUTER PAR L'ALGORITHME, EN FONCTION DE LA TAILLE DES DONNÉES

- **OPERATIONS « ELEMENTAIRES » :**

  +, -, *, /, >, <, =, ET, OU

- **TAILLE DES DONNEES = ESPACE OCCUPE**

==> PLUSIEURS « MESURES » POSSIBLES :

- **MEILLEUR DES CAS (PEU PERTINENT),**
- **EN MOYENNE,**
- **PIRE DES CAS.**

COMPLEXITE DES ALGORITHMES

- **« PIRE DES CAS »**

Nombre maximum d'opérations effectuées par l'algorithme sur des données de taille n (nombre exprimé en fonction de n)

→ borne supérieure du coût

→ assez facile à calculer (en général)

→ « souvent » réaliste

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

AjoutTab (…)

```
début
    pour  i = 0 à m-1   faire
  m  |      A[i] := B[i] ;                          1
     |      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 élémentaires ?

Pire des cas, test A[i]<B[j] toujours vrai

m(1+ 3n)= 3nm+m opérations élémentaires

Meilleur des cas, test A[i]<B[j] toujours faux

m(1+ n)= nm+m opérations élémentaires

Complexité du tri par sélection ?

Tri-selection (rappel)

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

n-i-1 fois

Pour chaque i,

1+2(n-i-1)+3

opérations

nombre d'opérations *(pire des cas)*

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

$$\mathbf{4\ (n-1) + n(n-1) = n^2 + 3n - 4}$$

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

- **COMPLEXITE EN MOYENNE**

(A chaque donnée est associée une proba.)

Espérance du nombre d'op. effectuées par l'algorithme sur l'ensemble des données de taille n (exprimée en fonction de n)

→ Nécessite de connaître (ou de pouvoir estimer) la distribution des données

→ Souvent difficile à calculer

→ Intéressant si le comportement « usuel » de l'algorithme est éloigné du pire cas

EXEMPLE : produit de 2 matrices

```
static float[][] produitMatrices (float[][] A,
  float[][] B){
  //Données A[m,n] et B[n,p] ; Résultat C[m,p]
  float[][] C = new float[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;
}
```

n fois — *p fois* — *m fois*

pire cas = cas moyen = 1+mp(3n+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 "*** $\Leftrightarrow$

il existe deux constantes entières n_0 et c :

$f(n) \leq c.g(n)$, pour tout entier $n > n_0$

(En pratique : f=O(g) $\Leftrightarrow$ Il existe c' : $f(n) \leq c'.g(n)$ pour n>1)

EXEMPLE

Tri par sélection : $f(n) = n^2+3n-4$ opérations

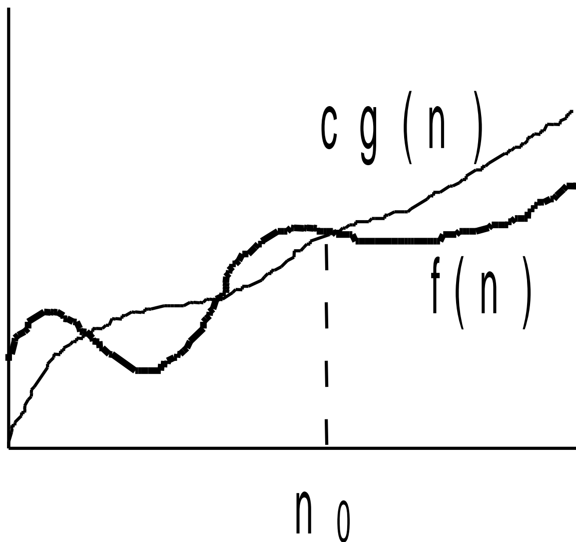
→ $f(n) = O(?)$

pour tout $n \geq 1$: $f(n) \leq 4n^2$

$n_0=1$ $c=4$ $g(n)=n^2$

donc $f(n) = O(n^2)$

Notation O : majorant du nombre d'opérations élémentaires d'un algorithme (*comportement asymptotique*)



f = O(g(n))

Complexité des algorithmes :

- **analyse du comportement des algorithmes quand la taille des données n augmente,**
- **permet la comparaison entre algorithmes.**

Algorithme polynomial : $f(n) = O(n^p)$ pour une certaine constante p

Tri par sélection (rappel) : **$f(n) = n^2+3n-4 = O(n^2)$**

En général, un algorithme polynomial est considéré comme « bon » !

Autre mesure : complexité « spatiale » (ou complexité en mémoire)

- **espace de stockage utilisé en plus de celui permettant de stocker les données,**
- **exprimer en fonction de la taille des données n, permet un autre type de comparaison entre algorithmes quand n augmente.**

Ainsi, si un algorithme prend comme donnée un tableau de n entiers, et réalise une copie de ce tableau au cours de son exécution, sa complexité en mémoire sera O(n).

Complexité des problèmes :

- **un problème sera considéré comme « facile » (resp. « *polynomial* ») s'il peut être résolu par un algorithme efficace / rapide (resp. polynomial),**
- **ainsi, le problème du tri de n entiers est polynomial.**

Attention :

- **Aucun algorithme efficace / polynomial connu pour un problème ==> on *ignore* s'il est facile / polynomial**
- **En revanche, cela n'exclut pas qu'il *POURRAIT* être facile / polynomial, en utilisant la bonne stratégie !**
- **Ainsi, trier n entiers est un problème polynomial, mais peut-il être résolu plus rapidement qu'en $O(n^2)$?**

Complexité des Problèmes **VS** Complexité des Algorithmes

Algorithme « efficace »

= Algorithme polynomial

==> Problème polynomial

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

Problèmes « difficiles » :

= qui n'admettent pas d'algorithme polynomial

(notion formalisée par la *Théorie de la Complexité*)

EXEMPLE : **partition d'entiers**

UN AUTRE EXEMPLE (ILLUSTRATION)

- **REPARTITION DE TACHES**
  - Données : n tâches, chacune avec un début d_i et une fin f_i
  - Problème : répartir les n tâches $t_1,\ldots,t_n$ entre un nombre minimum de processeurs, chacun ne pouvant effectuer que des tâches ***compatibles*** (= qui ne se chevauchent pas)
  - Principe de l'algorithme résolvant le problème : trier les tâches par ordre croissant des dates de fin f_i, puis, processeur par processeur, affecter des tâches compatibles de façon *gloutonne*, dans l'ordre de leurs f_i

ALGORITHME REPARTITION-TACHES

Donnée : T=liste de tâches,

Sortie : Tab=tableau listant, pour chaque processeur, les tâches à effectuer

<u>début</u>

proc = -1; */* nombre de processeurs déjà utilisés */*

trier et numéroter les t_i dans l'ordre croissant des f_i ($f_1 \leq f_2 \leq \ldots \leq f_n$)

tant que il reste des tâches à affecter faire

    proc++; Tab[proc] = {tâche non affectée de plus petit indice j};

    pour **i = 1** à **n** faire

        si **$d_i \geq f_j$ et t_i non encore affectée** alors

            Tab[proc] = Tab[proc]U{t_i};

            j = i; */* t_j = dernière tâche affectée */*

        finsi

    fait

fait

<u>fin</u>

EXEMPLE

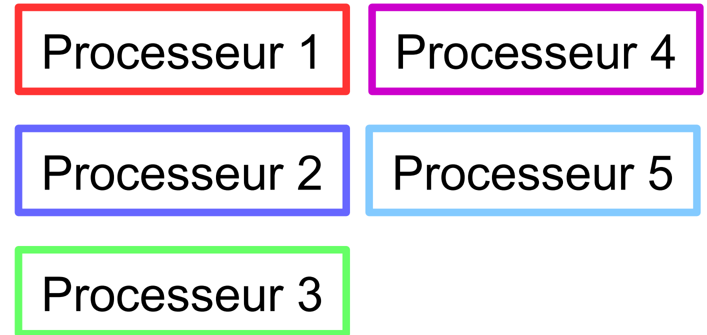
DE 11 TACHES

i	1	2	3	4	5	6	7	8	9	10	11
di	1	3	0	5	3	5	6	8	8	2	12
fi	4	5	6	7	8	9	10	11	12	13	14

d_i = date de début de la tâche t_i

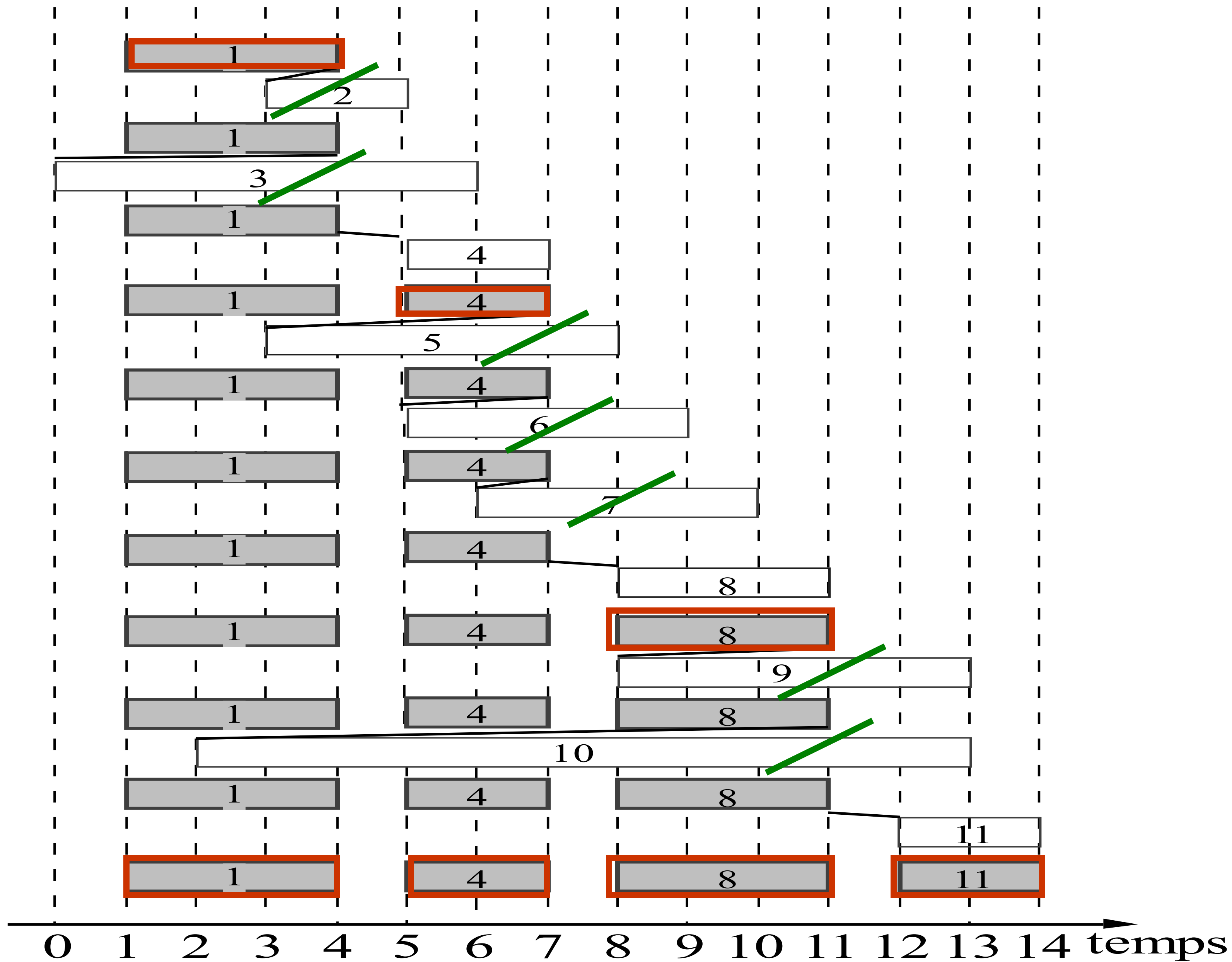
f_i = date de fin de la tâche t_i

==> tâches triées selon les f_i



Algorithme correct et complexité pire cas = tri + $O(n^2)$

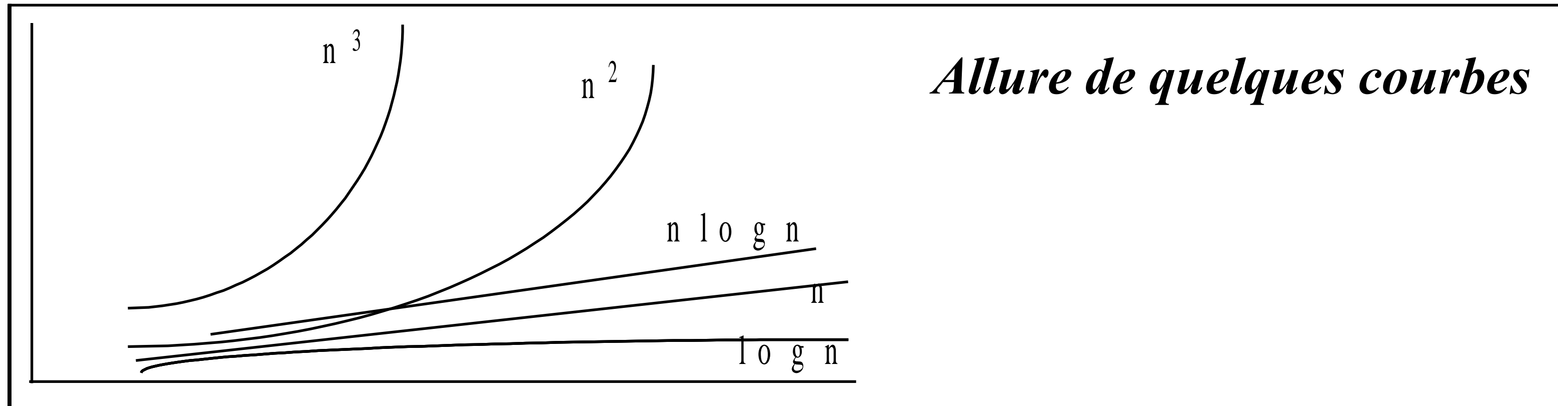
$\Rightarrow$ Complexité au pire cas en $O(n^2)$ pour le tri par sélection, donc algorithme glouton de même complexité au pire cas pour ce problème, qui est donc polynomial (comme le tri) !



Processeur 1

“Bonne complexité” (données volumineuses) :

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



*Comparaison chiffrée (1 an < 3,33*10⁷s) :*

n = 10^9, et 1 ns par opération élémentaire

$\log_2 n$	n	$n\log_2 n$	n^2	n^3
?				

De l'effet des progrès de la technologie sur la résolution des problèmes (cf ED)

	taille max des problèmes traitables en 1 heure avec un ordinateur :		
nombre d' opérations	**actuel (le plus rapide)**	**100 fois plus rapide**	**1000 fois plus rapide**
N	**N1**	**100 N1**	**1000 N1**
n^2	**N2 (soit N_2^2 opérations)**	**10 N2**	**31,6 N2**
n^5	**N3 (soit N_3^5 opérations)**	**2,5 N3**	**3,98 N3**
2^n	**N4 (soit 2^{N4} opérations)**	**N4 + 6,64**	**N4 + 9,97**
3^n	**N5 (soit 3^{N5} opérations)**	**N5 + 4,19**	**N5 + 6,29**

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

Complexité des algorithmes récursifs

- Faire des appels récursifs en cascade peut amener à en générer un très grand nombre, et un algorithme récursif consomme donc parfois beaucoup plus de mémoire que son équivalent itératif (recopie des paramètres dans des variables locales à chaque appel) : cf ED (et le cours *Introduction*)
- La complexité des algorithmes récursifs peut être difficile à évaluer (compter le nombre d'appels générés pouvant être problématique)

Une illustration de l’amélioration de complexité : recherche séquentielle vs recherche dichotomique d’un élément dans un tableau trié

Définition du problème

Etant donnés :

- 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

- On parcourt le tableau T, et on s'arrête :
  - Soit parce qu'on trouve x,
  - Soit parce qu'on trouve un élément > x.

Complexité au pire cas ?

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

Recherche séquentielle (en Java)

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

Recherche dichotomique

Idée de l'algorithme :

- Eviter la recherche séquentielle en utilisant la structure triée du tableau,
- Complexité au pire cas en O(log n).

Implémentation et analyse ? ==> cf ED :

- Version itérative,
- Version récursive (+ avantages/inconvénients).

REMARQUES FINALES

- Complexité des problèmes vs complexité des algorithmes : peut-on trier n entiers plus rapidement qu'en $O(n^2)$?
- Comptage « grossier » des opérations (+, -, /, *, tests, etc.) ; faire attention aux boucles !
- On écrit plutôt $O(\log(n))$ que $O(\log_2(n))$.
- Hiérarchie des complexités (pour $n > 1$) :
  $\log(n) < n < n \log(n) < n^2 < n^3 << 2^n$