Algorithmique Avancée et Complexité

Master : Cyber Sécurité et Intelligence Artificielle

CSIA

Intitulé du Master : Cyber Sécurité et Intelligence Artificielle

Semestre : 01

Intitulé de l'UE : Unité méthodologie

Intitulé de la matière : Algorithmique avancée et Complexité

Enseignant responsable de la matière: Bouafia zouheyr

Crédits : 4

Coefficients : 2

Objectifs de l'enseignement

- ✓ Analyser la complexité et l'efficacité des algorithmes.
- ✓ Concevoir des algorithmes en appliquant des stratégies de conceptions.
- ✓ Savoir manipuler des structures de données complexes : Listes, Piles, Files, Arbres

Connaissances préalables recommandées

- ✓ Algorithmique et structures de données.

Contenu de la matière

1. Notions d'algorithmes.
2. Théorie de la complexité : Complexité temporelle – ordre - NP-Complétude.
3. Récursivité et le paradigme "Diviser pour régner".
4. Algorithmes de Tri : Tri par tas - Tri rapide.
5. Structures de données élémentaires : Piles et files - Listes chainées - Arbre et Tas – Graphe.
6. Programmation dynamique.
7. Algorithmes voraces (gloutons).

Mode d'évaluation : Contrôle continu, Mini projet, examen final.

Bibliographie

- Introduction à l'algorithmique, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Cliord Stein, Dunod, Paris 204-2010.
- Initiation a l'algorithmique et a la programmation en C, Remy Malgouyres, Rita Zrour, Fabien Feschet, Dunod 2008-2011.
- Algorithmique et programmation en Java, Vincent Granet, Dunod, Paris, 2000.
- Levy, J. J., & Cori, R. Algorithmes et Programmation. École Polytechnique
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.

Qu’est-ce que l’algorithmique

- *L’algorithmique est l’étude des algorithmes.*
- *Un algorithme est suite finie d’opérations élémentaires constituant un schéma de calcul ou de résolution d’un problème.*
- **Double problématique de l’algorithmique**
  - Trouver une méthode de résolution (exacte ou approchée) du problème.
  - Trouver une méthode **efficace.**
    - Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre

Différences entre algorithmes et programmes

- Un **programme** est la réalisation (l'implémentation) d'un algorithme au moyen d'un langage donné (sur une architecture donnée).
- Il s'agit de la mise en œuvre du principe.
- Par exemple, lors de la programmation on s'occupera parfois explicitement de la gestion de la mémoire (allocation dynamique en C) qui est un problème d'implémentation ignoré au niveau algorithmique.

Exemple

- Exemple de tâche: Décider si un tableau L est trié en ordre croissant.
- Raisonnement: Un tableau L est trié si tous ses éléments sont dans l'ordre croissant
- Algorithme : Une fonction vérifiant cette propriété, supposera donc le tableau L, de taille n, trié au départ et cherchera une contradiction.

Exemple

Algorithme

```
Fonction trie (L: tab, n: entier) : booleen
Variables i, n: entier;
                Ok: booleen ;
Début
Ok ← vrai
pour i de 1 à n faire
  si (L[i ] > L[i +1]) alors
    Ok ← Faux
  Finsi
Finpour
Retourner OK
Fin trie
```

Programme

```
boolean trie (tab L, int n)
{ Ok =true;
For (i =0; i< n ; i++)
{ if (L[i ] > L[i +1])
   Ok= Faux; }
return Ok;}
```

Problématique de l'algorithmique

- La question la plus fréquente que se pose à chaque programmeur est la suivante: **Comment choisir parmi les différentes approches pour résoudre un problème?**
- Exemples: Trier un tableau: algorithme de tri par insertion ou de tri rapide?…, etc

1. La correction: résout il bien le problème donné?
2. L'efficacité: en combien de temps et avec quelles ressources?

La théorie de la complexité

- La théorie de la complexité est une branche de l'informatique théorique, elle cherche à calculer, formellement, la complexité algorithmique nécessaire pour résoudre un problème P au moyen de l'exécution d'un algorithme A.
- La théorie de la complexité vise à répondre aux besoins d'efficacité des algorithmes (programmes):
- Elle permet:
  - ✓ Classer les problèmes selon leur difficulté.
  - ✓ Classer les algorithmes selon leur efficacité.
  - ✓ Comparer les algorithmes résolvant un problème donné afin de faire un choix sans devoir les implémenter.

La théorie de la complexité

- L'efficacité d'un algorithme peut être évaluée par:
  - Rapidité (en terme de temps d'exécution)
  - Consommation de ressources (espace de stockage, mémoire utilisée)
- La théorie de la complexité : étudie l'efficacité des algorithmes.
  - On s'intéresse dans ce cours, essentiellement, à l'efficacité en terme de temps d'exécution.

Temps d’exécution

- On ne mesure pas la durée en heures, minutes, secondes, ... :

  1. cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;

  2. de plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;
- On utilise des unités de temps abstraites proportionnelles au nombre d'opérations effectuées .
- Chaque instruction basique consomme une unité de temps (affectation d'une variable, comparaison, +, - ,*, /, ...)

Temps d’exécution

Règles:

- Chaque instruction basique consomme une unité de temps (affectation d’une variable, lecture, écriture, comparaison,…).
- Chaque itération d’une boucle rajoute le nombre d’unités de temps consommés dans le corps de cette boucle.
- Chaque appel de fonction rajoute le nombre d’unités de temps consommées dans cette fonction.
- Pour avoir le nombre d’opération effectuées par l’algorithme, on additionne le tout.

Temps d'exécution

- L'algorithme suivant calcule $n!=n*n-1*n-2*\cdots*2*1$, avec 0!=1.

```
fonction factorielle(n)
fact ← 1 ;                          initialisation : 1
i ← 2 ;                             initialisation : 1
tant que (i <= n)faire              itération : au plus n-1 comparai
fact ← fact * i ;                   multiplication + affectation : 2
i ← i + 1 ;                         addition + affectation : 2
fin tant que
renvoyer fact;                      renvoi d'une valeur : 1
```

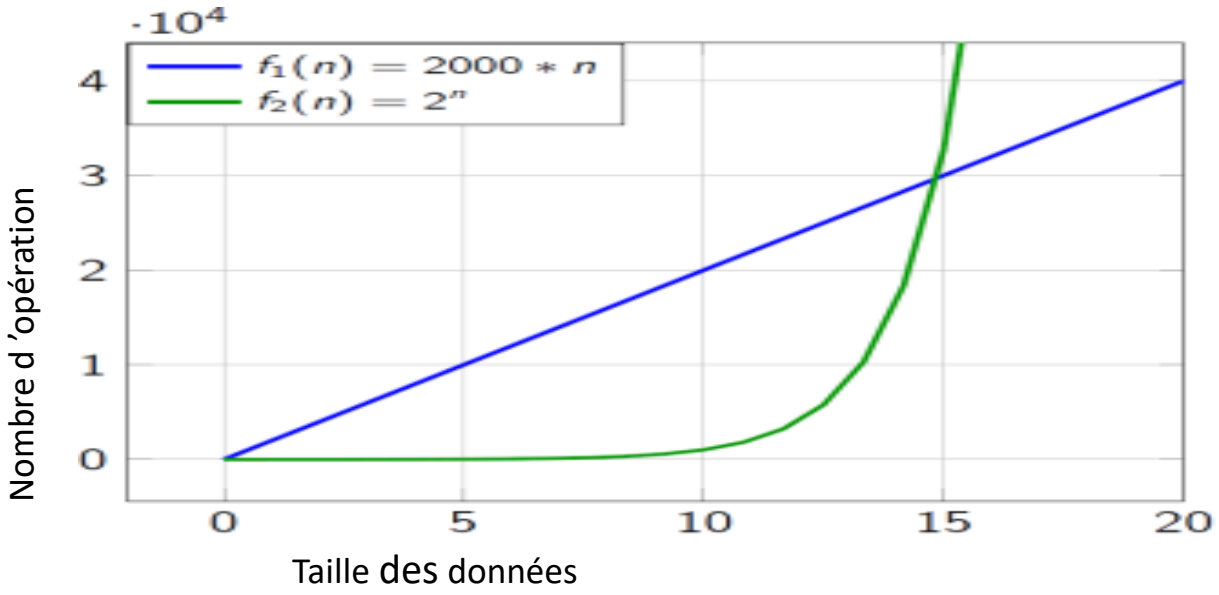
- Avec le test à faire à chaque itération, le nombre total d'opérations élémentaires est :
- $T(n)=1+1+(n-1)*5+1+1=5n-1$
- Cette algorithme consomme un temps linéaire

Complexité algorithmique

- La complexité d'un algorithme est une mesure de sa performance **asymptotique dans le pire cas ;**
- **Asymptotique** : On s'intéresse a des données très grandes
- les petites valeurs ne sont pas assez informatives
- **Pire cas** : On s'intéresse a la performance de l'algorithme dans les situations ou le problème prend le plus de temps a résoudre ;
- on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé ;

Complexité algorithmique

- **Comportement asymptotique :** Soit deux algorithmes résolvant un problème (de taille de données n) en un temps f1(n) et f2(n), respectivement ;



- Quel algorithme préférez-vous ?
- La courbe verte semble correspondre a un algorithme beaucoup plus efficace ..
- ... mais seulement pour de très petites valeurs !

Complexité algorithmique

Notations de Landau

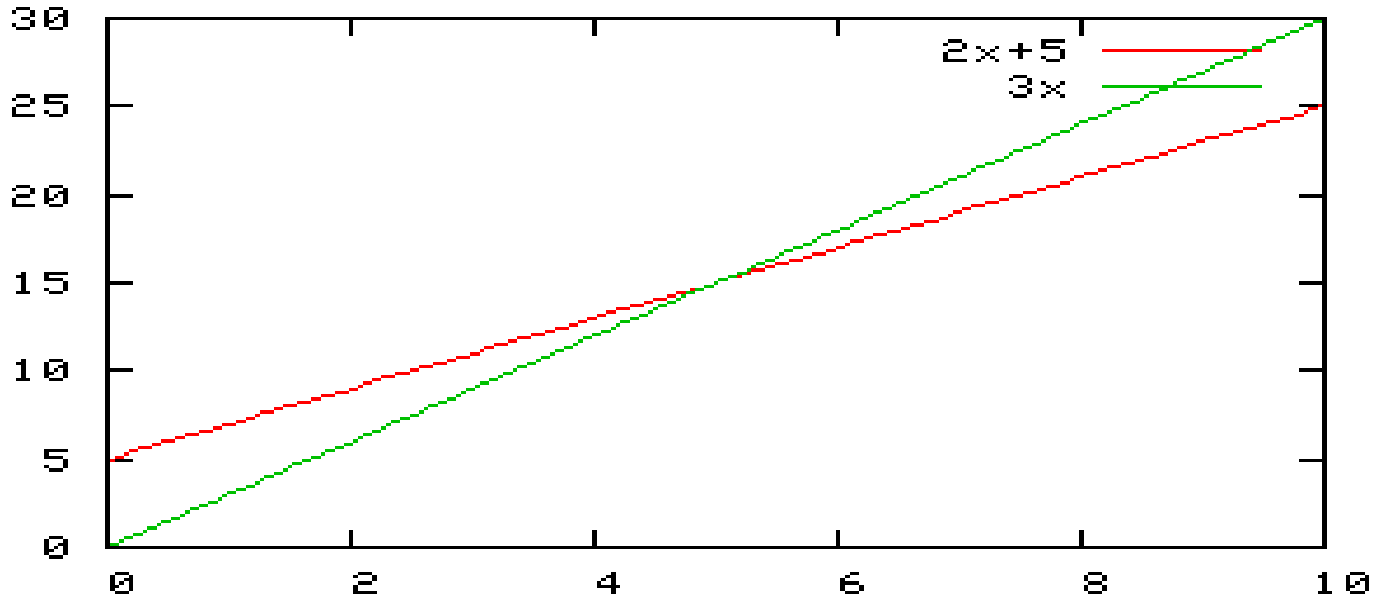
- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;
- On aura donc recours a une approximation de ce temps de calcul, représentée par les notations :
  - La notation O (grand O) ou (de l'ordre de)
  - La notation Ω (grand oméga)
  - La notation Θ (grand theta)

Notation grand O

- Soit *f(n)* le nombre d'opérations effectués par un algorithme ; on dit que le temps mis par l'algorithme est *O(g(n))* ou bien *f(n) ϵ O(g(n))* et s'il existe une constante positive *c* et il existe une constante *n0* telle que *f(n) ≤ c g(n) . ∀ n ≥ n0*
- *O(g(n)) = { f :IN → IN | ∃* c>0 et n0 ≥*0 tels que 0≤ f(n) ≤ cg(n) ∀n ≥ n0 }*
- *On dit que g(n) est une borne supérieure asymptotique pour f(n), on note abusivement f(n)=* O*(g(n)). (**le pire des cas**)*

Notation grand O

- *Exemple : $f(n)=2n+5 = O(n)$ car $2n+5 \leq 3n \ \forall n \geq 5$ d'où $c=3$ et $n0=5$*



Notation Ω

- *Ω(g(n)) = { f :IN → IN | ∃ c>0* et *n0 ≥ 0 tels que f(n) ≥ cg(n) ∀ n ≥ n0 }*
- *Si f(n) ∈ Ω(g(n)),* on *dit que g(n) est une borne inférieure asymptotique pour f(n), on note abusivement f(n)= Ω(g(n)).(**le meilleur des cas**)*
- Exemple : f(n)=2n+5= Ω (n) car 2n+5≥ *2n ∀ n≥1 d'où c=2 et n0=0*

Notation Θ

- *Θ(g(n)) = { f :IN → IN | ∃ c1>0, ∃ c2>0 et n0 ≥0 tels que c1.g(n) ≤ f(n) ≤ c2.g(n) ∀n ≥ n0 }*
- *Si f(n) ∈ Θ(g(n)) On dit que g(n) est une borne asymptotique pour f(n), on note abusivement f(n) = Θ(g(n)). (**Tous les cas**)*
- *Exemple : f(n)=2n+5= Θ(n) car 2n ≤ 2n+5 ≤ 3n ∀n ≥ 5 d'où c1=2, c2=3 et n0=5*

Complexité algorithmique

- **Notations de Landau : Simplification sur le calcul du temps d'exécution**
- On préfère avoir une idée du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée ;
- Retenir que les termes dominants et mettre à 1 les constantes multiplicatives,
- Exemple : f(n)=4n^3–5n^2+2n+3
- Retenir le terme de plus haut degré : $\boldsymbol{4n^3}$
- Mettre à un les coefficients : $\boldsymbol{n^3}$
- Nous avons donc f($\boldsymbol{n}$)=$\boldsymbol{O}$($\boldsymbol{n^3}$)

Complexité algorithmique

- De façon générale, les règles de la notation en O sont les suivantes:
- Les termes constants : O(c) = O(1)
- Les constantes multiplicatives sont omises: O(cT) = c O(T)=O(T)
- L'addition est réalisée en prenant le maximum:

  O(T1)+O(T2)= O(T1+T2)=max(O(T1); O(T2))
- La multiplication reste inchangée:

  O(T1) O(T2)= O(T1*T2)

Complexité algorithmique

- *Cas d'une instruction simple : Les instructions de base (lecture, écriture,* affectation …) prennent un temps constant, noté O(1).
- *Cas d'une suite d'instructions: le temps d'exécution d'une séquence est* déterminé par la règle de la somme.

Traitement1 $T_1(n)$

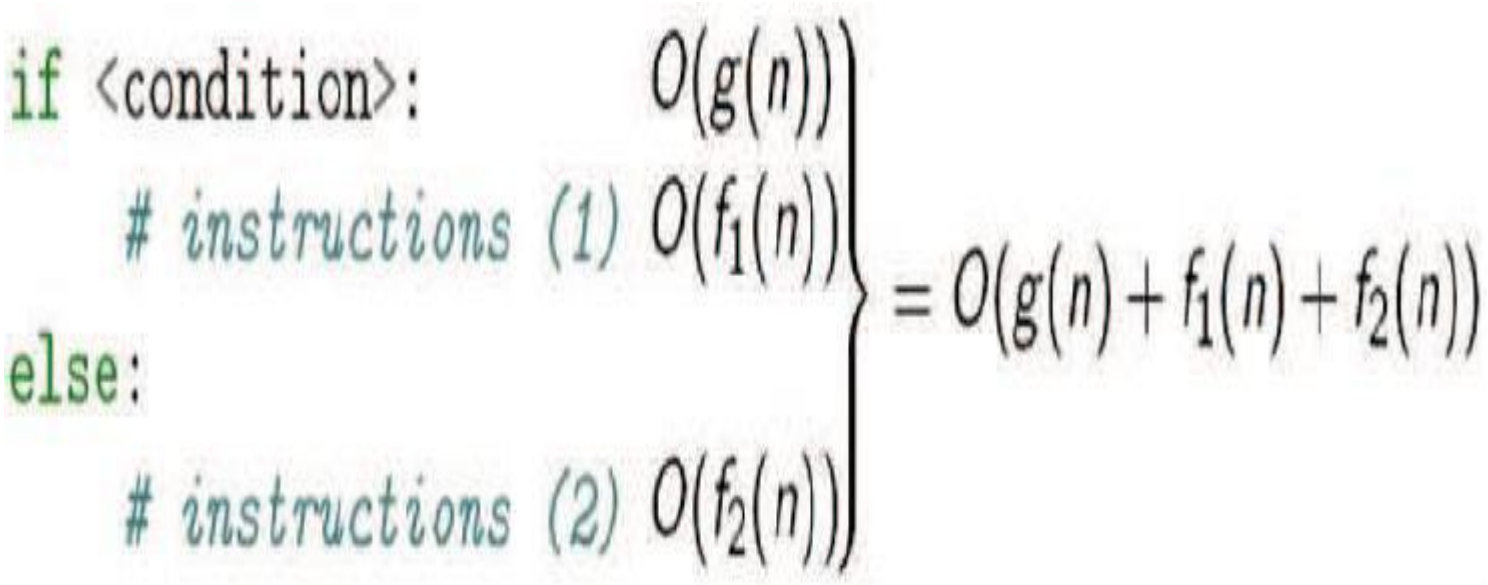
Traitement2 $T_2(n)$

$$T(n) = T_1(n) + T_2(n)$$

$$O(T) = O(T1 + T2) = \max(O(T1); O(T2))$$

Complexité algorithmique

- *Cas d'une branche conditionnelle: le temps d'exécution est déterminé aussi par* la règle de la somme.



Complexité algorithmique

- *Cas d'une boucle: On multiplie la complexité du corps de la boucle par le* nombre d'itérations.
- Exemple pour la boucle while (tant que), la complexité se calcule comme suit:

```
# en supposant qu'on a m itérations
while <condition>:
    # instructions
```

$$\left.\begin{matrix} O(g(n)) \\ O(f(n)) \end{matrix}\right\} = O(m*(g(n)+f(n)))$$

Complexité algorithmique

Pour calculer la complexité d'un algorithme:

- On calcule la complexité de chaque partie de l'algorithme.
- On combine ces complexités conformément aux règles déjà vues.
- On effectue sur le résultat les simplifications possibles déjà vues.

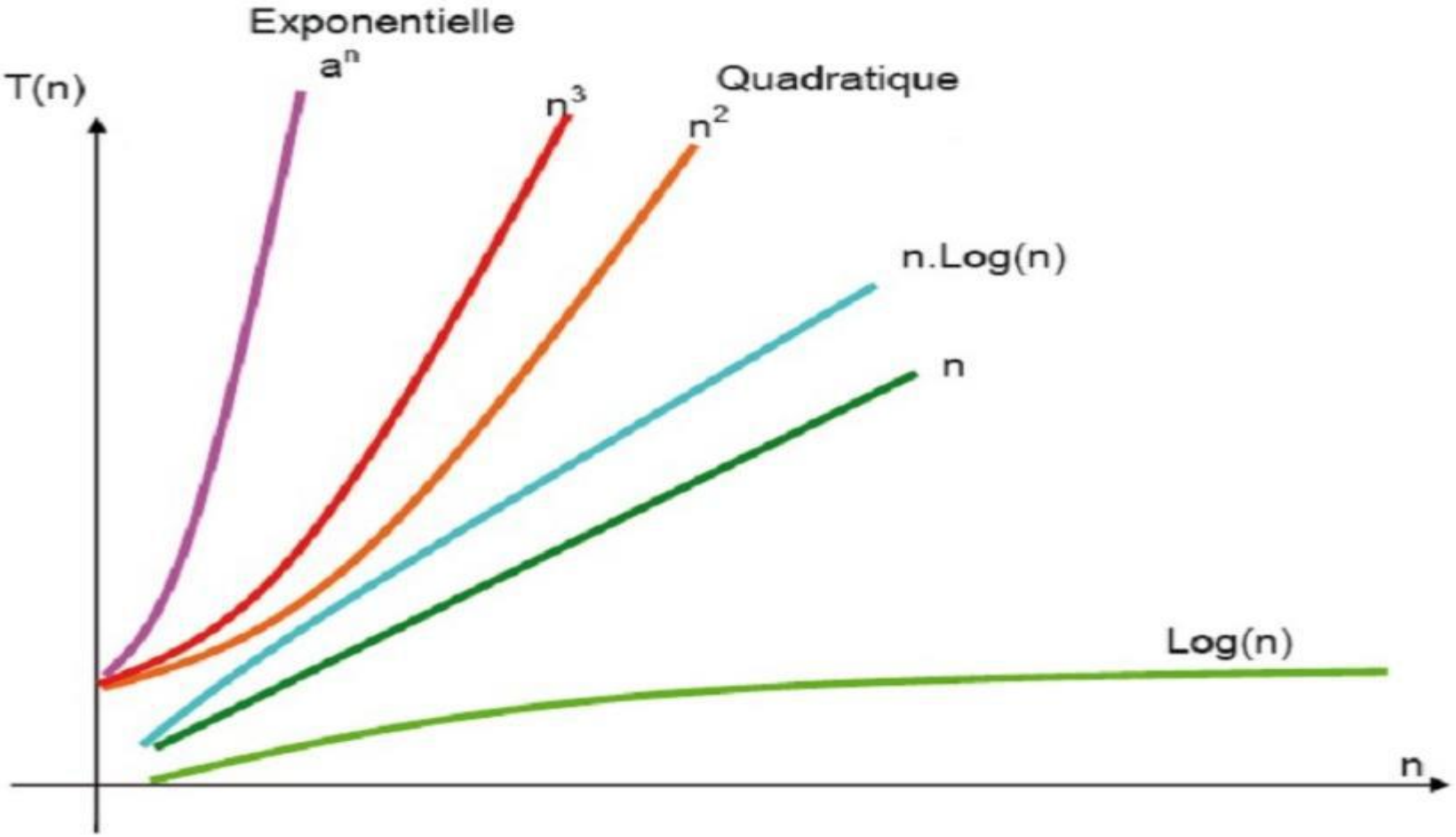
Complexité algorithmique

- On peut ranger les fonctions équivalentes dans la même classe.
- Deux algorithmes de la même classe sont considérés de même complexité.
- Les classes de complexité les plus fréquentes (par ordre croissant selon O(.))

Complexité algorithmique

Complexité	Classe
$O(1)$	constant
$O(logn)$	logarithmique
$O(n)$	linéaire
$O(nlogn)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel
$O(n!)$	factorielle

Complexité algorithmique



Complexité algorithmique

Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme, si on suppose qu'une instruction est de l'ordre de la μs;

T.\C.	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} *siècles*
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	*astronomique*
10000	$13\mu s$	$1/100s$	$1/7s$	$1,7mn$	*astronomique*
100000	$17\mu s$	$1/10s$	$2s$	$2,8h$	*astronomique*

Exemple : recherche séquentielle

- *Principe:* Le principe consiste à parcourir un tableau d'éléments dans l'ordre de ses indices jusqu'à ce qu'un élément recherché soit trouvé ou bien que la fin du tableau soit atteinte et l'élément recherché est alors inexistant.
- Soit T un tableau contenant les10 éléments suivants:

T :	12	10	0	-5	8	12	-2	2	40	-1
	1	2	3	4	5	6	7	8	9	10

Pour x =-2, le programme affichera que ‘’-2 existe’’
Pour x =5, le programme affichera que ‘’5 n'existe pas’’

Exemple : recherche séquentielle

Algorithme:

Fonction Recherche_séquentielle (x:entier,T:tab, n:entier) : booleen

Var

i: entier

trouve: booleen

Début

i ← 0 O(1)

trouve ← faux O(1)

Tant que ((i<n) et (non trouve)) faire O(1) nombre d'itération = n

i ← i +1 O(1)

si (T[i] = x) alors O(1)

trouve ← vrai O(1)

finsi

fintq

Retourner (trouve) O(1)

Fin

T(n)= 1+1 + 6 *n + 2 = 6n+4

O(T) = max(O(1) , O(n * 1) , O(1) = O(n)

Exemple : Recherche dichotomique

- On veut chercher un élément dans un tableau trié dans le sens croissant.
- Le but de cette recherche est de diviser l'intervalle de recherche par 2 à chaque itération. Pour cela, on procède de la façon suivante:
- Soient premier et dernier les extrémités gauche et droite de l'intervalle dans lequel on cherche la valeur x, on calcule m, l'indice de l'élément médian: m=(premier + dernier) div 2

Exemple : Recherche dichotomique

- Il y a 3 cas possibles:
  - ✓X<T[m], l'élément x s'il existe, il se trouve dans l'intervalle [premier... m-1]
  - ✓ X > T[m], l'élément de valeur x, s'il existe, se trouve dans l'intervalle [m+1... Dernier]
  - ✓X=T[m], x est trouvé, la recherche est terminée
- La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou que l'intervalle de recherche soit vide.

Exemple : Recherche dichotomique

Algorithme: Procédure recherche_dichotomique ((x:entier, T:tab, n:entier)
Var premier, dernier, m: entier; trouve : booléen
Début
premier ←1; dernier ← n ; trouve ← faux ;
répéter
m ← (premier + dernier) div 2
si (x < T[m]) alors
dernier ← m-1
sinon si(x>T[m]) alors
premier ← m+1
sinon
trouve ← vrai
Finsi
Finsi
Jusqu'à (trouve=vrai ou premier > dernier)
Si (trouve) alors
écrire(x, "existe et son indice est'', m)
Sinon
écrire(x, " introuvable ")
Finsi
fin

Exemple : Recherche dichotomique

- ***Calcul de complexité:***

  Supposant que le tableau est de taille n une puissance de 2, ($n= 2^q$) .

  Le pire des cas pour la recherche d'un élément est de continuer à diviser jusqu'à obtenir un tableau de taille 1.

  q est le nombre d'itérations nécessaires pour aboutir à un tableau de taille 1.

Exemple : Recherche dichotomique

$$itration1 \leftrightarrow \frac{n}{2} = \frac{n}{2^1}$$
$$itration2 \leftrightarrow \frac{n}{4} = \frac{n}{2^2}$$
$$itration3 \leftrightarrow \frac{n}{8} = \frac{n}{2^3}$$
$$itration4 \leftrightarrow \frac{n}{16} = \frac{n}{2^4}$$
$$\ldots \leftrightarrow \ldots$$
$$itrationq \leftrightarrow \frac{n}{2^q}$$

dernière itération → taille tableau = 1

$$\frac{n}{2^q} = 1$$
$$2^q = n$$
$$q = log_2(n)$$

→ La complexité = $log_2(n)$

Optimalité des Algorithmes de recherche

- Recherche séquentielle O(n)
- Recherche dichotomique $O(\log_2(n))$
- L'algorithme de recherche séquentielle est de complexité linéaire et celui de recherche dichotomique est de complexité logarithmique.
- L'algorithme de recherche dichotomique est plus optimal que l'algorithme de recherche séquentielle pour les tableau déjà trié.

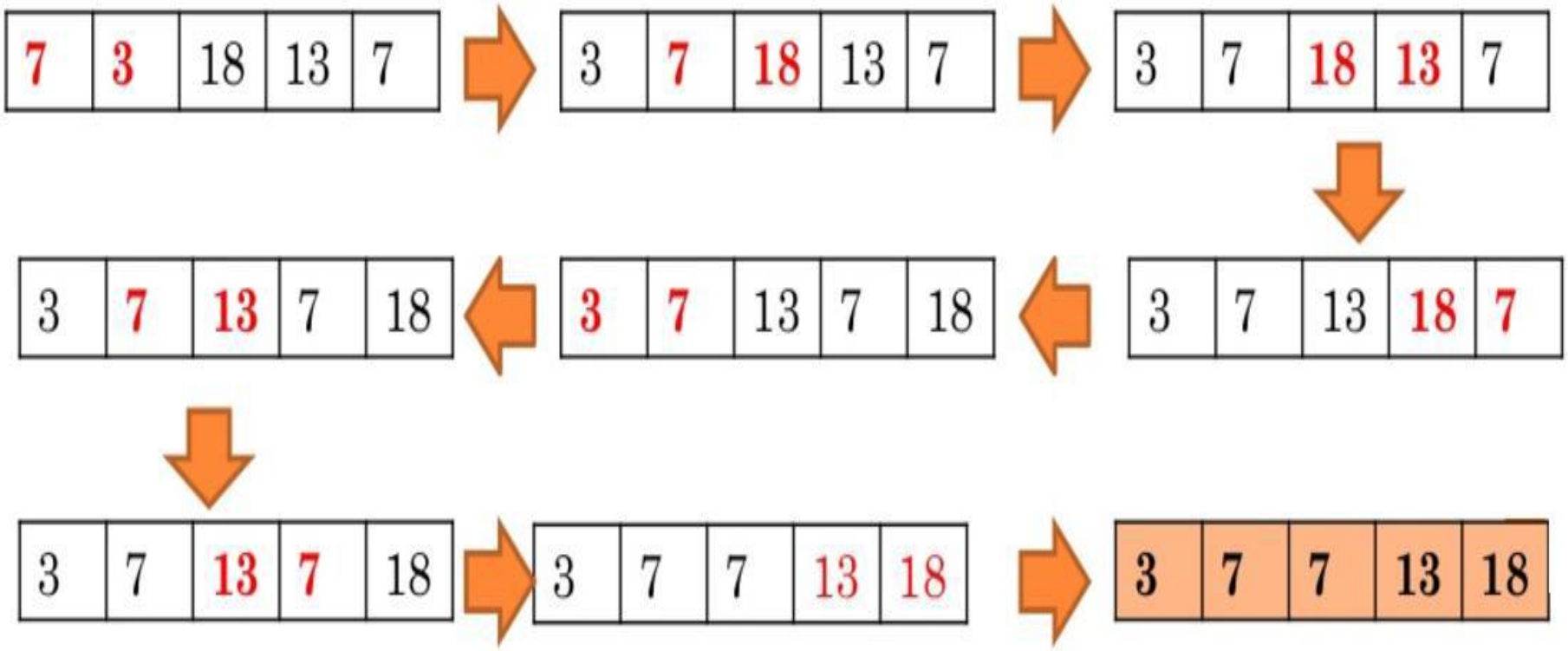
Exemple : algorithme de tri

Principe: Tri à bulles

- Elle consiste à répéter le traitement suivant: parcourir les éléments du tableau de 1 à (n-1): si l'élément i est supérieur à l'élément (i+1) alors, on les permute.
- Le programme s'arrête lorsqu'aucune permutation n'est réalisable après le parcours complet du tableau.

Exemple : algorithme de tri

Tri à bulles



Exemple : algorithme de tri

Algorithme: Procédure Tri_à_bulles (T:tab, n:entier)

```
Var echange: booléen
     i: entier
Début
répéter
   echange ←faux
    pour i de 1 à n-1 faire
        si T[i] > T[i+1] alors
            x ← T[i]
            T[i] ← T[i+1]
            T[i+1] ← x
            echange ← vrai
        finsi
fin pour
jusqu'à (echange = faux)
Fin
```

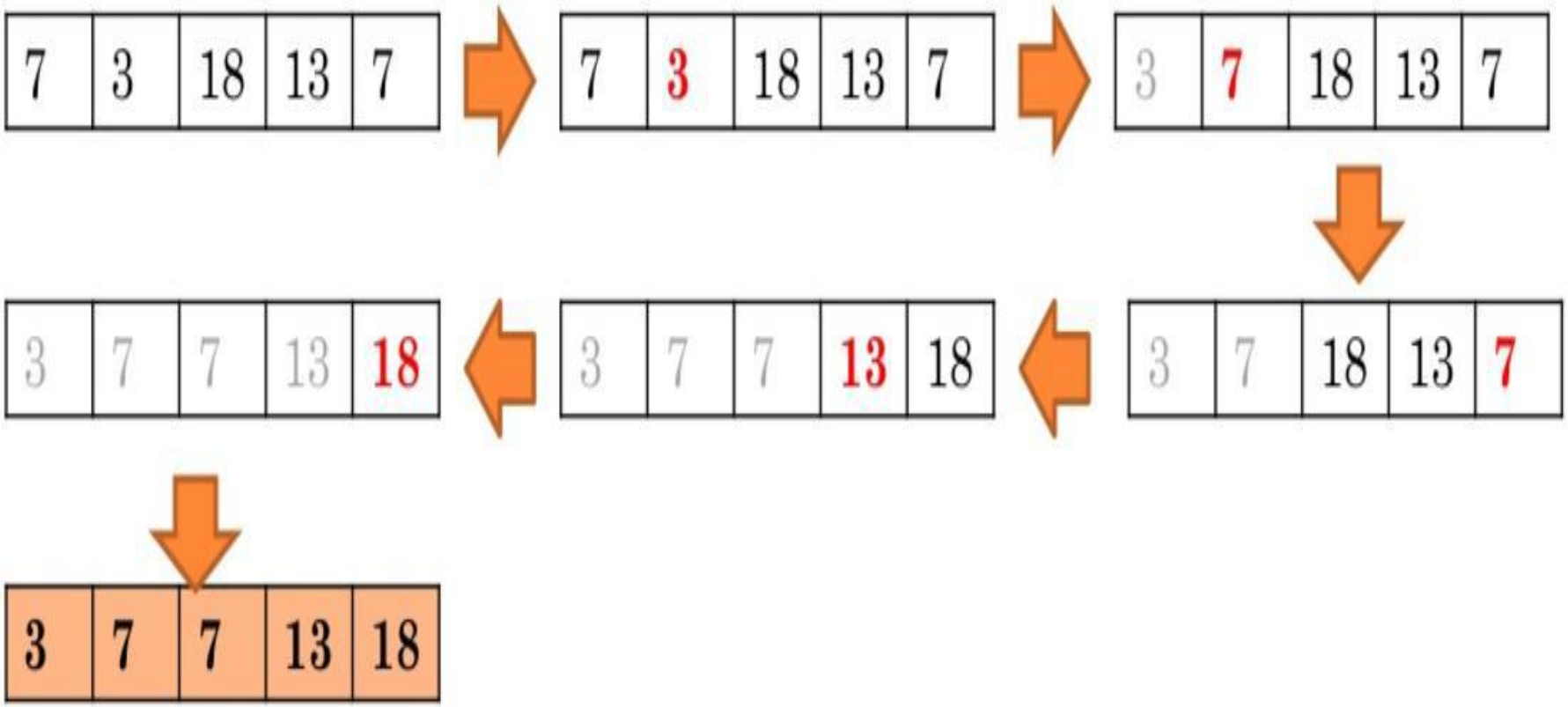
$O(T) = O(n^2)$

Exemple : algorithme de tri

- *Principe: Tri par sélection*
- Elle consiste à chercher l'indice du plus petit nombre du tableau (de 1 à n) et le permuter avec l'élément 1 du tableau, chercher ensuite le plus petit nombre (de 2 à n) et le permuter avec l'élément 2 du tableau, répéter ce traitement jusqu'à arriver à l'élément n-1.

Exemple : algorithme de tri

Tri par sélection



Exemple : algorithme de tri

Algorithme: Procédure Tri_par_sélection (T:tab, n:entier)
Var i, j, x, indmin: entier
Début
pour i de 1 à n-1 faire
    indmin ← i
    pour j de i+1 à n faire
        si T[j] < T[indmin] alors
            indmin ← j
        finsi
    finpour
    x ← T[i]
    T[i] ←T[indmin]
    T[indmin] ← x
fin pour
Fin

$O(T) = O(n^2)$

Exercice

- Ecrire une fonction en langage algorithmique qui prend en paramètre un entier n et un tableau a deux dimensions de taille n × n d'entiers. La fonction calcule la somme des éléments du tableau.
- Donnez la complexité de cet algorithme.