Structures de données abstraites

Notions d'abstraction

L'abstraction consiste à penser à un objet en termes d'actions que l'on peut effectuer sur lui, et non pas en termes de représentation et d'implémentation de cet objet.

Notions de Types Abstraits de Données (TAD)

- Un **type abstrait** est une spécification de données décrivant l'ensemble des opérations associées à ces données et l'ensemble des **propriétés** de ces **opérations**. Cette spécification ne précise pas la représentation interne.
- De ce fait, les algorithmes sont indépendants de la représentation interne des données.

Définition d'un TAD

Spécification mathématique d'un ensemble de valeurs et de l'ensemble des opérations qu'elles peuvent effectuer. Il est qualifié d'abstrait car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter (indépendant d'un langage).

Exemples :

- Dans un algorithme qui manipule des *entiers*, on s'interesse non pas à la représentation des entiers, mais aux **opérations définies sur les entiers** : **+**, **-**, *****, **/**
- Type *booléen*, ensemble de deux valeurs (*faux*, *vrai*) muni des opérations : ***non***, ***et***, ***ou***

Spécification d'une structure de données

Une structure de données peut être **spécifiée** selon deux niveaux d'abstraction, du plus abstrait au plus concret :

- **Une spécification abstraite ou Type Abstrait de Données (TAD) :** Description des propriétés générales et des opérations qui décrivent la structure de données. (**Spécifications mathématique**)
- **Une spécification opérationnelle ou Structure de Données Concrète (SDC)** : Description d'une forme d'implémentation informatique choisie pour représenter et décrire la structure de donnée.

Spécification abstraite

Un type abstrait est la donnée…

- de sa **signature** décrivant la **syntaxe** du type avec le nom des opérations
- des **propriétés** des opérations du type, sa sémantique qui seront données sous forme d'un ensemble d'axiomes (formules logiques). On introduit alors une **sémantique** (une signification) aux objets de la signature.

Signature d'un TAD

- **Type (sorte)** : Le nom du TAD
- **Utilise** : Les noms des types des objets utilisés par le TAD
- **Opérations** : Pour chaque opération, l'énoncé des types des objets qu'elle **reçoit** et qu'elle **renvoie**

Syntaxe d'écriture de la signture d'un type abstrait de données

Sorte : … les noms de types définis par le TAD, ce sont les types au sens des langages de prog
Utilise: … types déjà définis (types de bases ou types construits) utilisés par le TAD
Opérations : … cette partie décrit la syntaxe du TAD

nom1 : $Type_1$ x $Type_2$ x … ⟶ $Type'_1$ x $Type'_2$x …

Propriétés d'un type abstrait (sémantique)

Donne une sémantique (signification) aux sortes et aux opérations de la signature au moyen d'axiomes agiss ant sur les variables. Précise :

- **Préconditions** : Les domaines de définition (ou d'application) des opérations
- **Axiomes** : propriétés des opérations

On parle d'un Type Abstrait Algébrique (TAA) lorsque la sémantique est définie par un système d'équations

Syntaxe d'écriture des propriétés d'un type abstrait de données

Préconditions : le domaine d'application d'une opération
… **est définie si** …
Axiomes / Sémantique : décrivant sans ambiguïté ses opérations
$axiome_1$ Axiome : décrit logiquement ce que fait une composition d'opérations
Sémantique : explique ce que fait chaque opération
$axiome_n$

Opérateurs

Les opérations sont divisées en plusieurs types :

- Les **constructeurs** : ils permettent de créer un nouvel "objet" du type que l'on est en train de définir.
- les **modificateurs** : ils permettent de modifier les objets et leur contenu.
- les **accesseurs**: ils donnent des informations sur l'état de l'objet (valeurs, affichages, …)
- **Constante** : opérateur sans argument

On ne manipule un type abstrait qu'avec les fonctions qui lui sont associées !

Règles lors de l'analyse

A la définition d'un TAD, il faut faire attention à:

- la complétude : A-t-on donné un nombre suffisant d'opérations pour décrire toutes les propriétés du type abstrait ?
- la consistance (non contradictoire) : N'y a-t-il pas des compositions d'axiomes contradictoires ?

Comment bien définir les opérations d'un TAD ?

- Choisir des identifiants significatifs
- Bien préciser les conditions de bon fonctionnement (préconditions)

Exemples

Exemple : TAD Booléen

Sorte : Booléen **utilise :** /
Opérations :

vrai	: -> Booléen
faux	: -> Booléen
non	: Booléen -> Booléen
et	: Booléen x Booléen -> Booléen
ou	: Booléen x Booléen -> Booléen

Les constantes sont représentées sous forme d'opérations qui n'ont pas d'arguments.

Type valeur de retour

Nom des opérations

Deux arguments de type Booléen

Préconditions : /

Aucune précondition

Axiomes :

Soit, a, b : Booléen
non(vrai) = faux
non(non(a)) = a
vrai et a = a
faux et a = a
a ou b = non(non(a) et non(b))

FinTAD Booléen

Exemple : TAD Pile d'entiers

Type : pile **utilise :** entiers, booléen
opérations :

créer	:	-> pile
empiler	:	pile, entier -> pile
dépiler	:	pile -> pile, entier
estVide	:	pile -> Booléen

Préconditions :
dépiler(p) **est défini si** estVide(p) est faux
Axiomes :
soit, i : entier, p : pile
depiler(empiler(p, i)) = (p, i)
estVide(créer()) est vrai
estVide(empiler(p, i)) est faux
FinTAD Pile d'entiers

Exemple de TAD : Vecteur

Type : Vecteur **utilise :** Entier, Elément
Opérations :

vect	:	Entier -> Vecteur
changer_ième	:	Vecteur x Entier x Elément -> Vecteur
ième	:	Vecteur x Entier -> Elément
taille	:	Vecteur -> Entier

Préconditions :
vect(i) est_défini_si i >= 0
ième(v, i) est_défini_si 0 <= i <= taille(v)
changer_ième(v, i, e) est_défini_si 0 <= i <= taille(v)
Axiomes :
Soit, s: Séquence, i: entire, k: entire, e: élément
ième(changer-ième(s,i,e),i) = e
ième(changer-ième(s,i,e), j) = e **si** i=j **sinon** ième(s,j)
FinTAD Vecteur

Réutilisation des TAD

Quand on défint un type, on peut réutiliser des types déjà définis. La signature est l'union des signatures des types utilisés enrichie des nouvelles opérations. Le type hérite des propriétés des types qui le constitue.

Structure de données (Concrète) SDC

Manière d'organiser les données pour les traiter plus facilement. Une structure de donées **implémente concrètement** un type abstrait.

- Correspond à **l'implémentation d'un TAD**
- Composé d'un algorithme pour chaque opération, plus éventuellement des données spécifiques à la structure pour sa gestion
- Un même TAD peut donner lieu à plusieurs structures de données avec des performances différentes

Implémentation d'un TAD (TAD -> SDC)

Pour implémenter un TAD :

- **Déclarer la structure** de données retenue pour représenter le TAD : *l'interface (.hpp)*
- **Définir les opérations** dans un langage particulier : *La réalisation (.cpp)*

Plusieurs implémentations sont possibles pour un même TAD !

Exigences :

- Conforme à la spécification du TAD
- Efficace en terme de complexité d'algorithme

Pour implémenter on utilise :

- Les **types élémentaires** (primitifs, simples, atomiques) : contiennent une unique information
  - ⇨ Pointeurs, Entiers (int), Réels (floats), Caractères (char)…
- les **types composés** (agrégats, structures, classes) : contiennent plusieurs informations atomiques
  - ⇨ Tableaux (array), Enregistrements = Structures (struct), Classes, …

    Les types composés peuvent être :

    - Homogènes : les inforamtions sont toutes du même type (primitif ou composé)
    - Hétérogènes : les inforamtions sont de types différents

- les **types prédéfinis** (classes perso)

En résumé

- Le **TAD** définit le comportement de la structure de données et offre un niveau d'abstraction élevé (**Que peut-on faire avec?**)
- Le **SDC** est l'implémentation de ce TAD (**Comment cela est-il implémenté ?**)

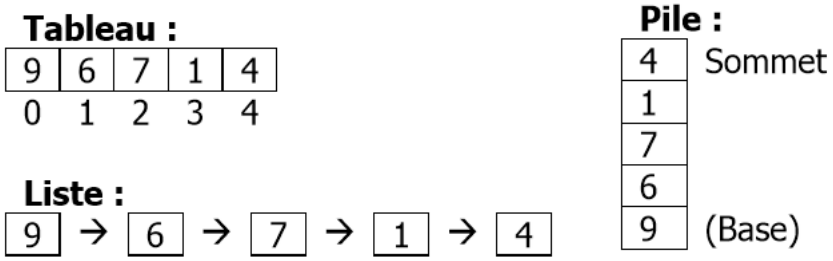
> **Précision concernant l'enregistrement :**
> Un enregistrement (appelé aussi **Structure** dans certains langages ou **Classe** en OOP) est un type, ou encore un méta-type complexe construit à partir de types plus simples.
>
> Un enregistrement est un ensemble d'éléments de types différents repérés par un **nom**. Les éléments d'un enregistrement sont appelés des **champs**.

Classification des structures de données

Vue d'ensemble des structures de données

On peut classer les structures de données en deux grandes catégories :

- **Les structures de données linéaires :** Permettent de relier des données en séquences (on peut numéroter (indexer) les éléments). Contient un unique premier et unique dernier élément, tous les autres éléments ont un unique prédécésseur et un unique successeur.
    - Acces **direct :** les éléments peuvent être accédés dans n'importe qul ordre (tableau, Enregistrements, Vecteurs)
    - Acces **séquentiel :** les éléments doivent être accédés dans un certain ordre (pile, files, listes)



- **Les structures de données non linéaires :** Permettent de relier un élément à plusieurs autres éléments (Arbres, Graphes, Ensembles (sets) Dictionnaires (maps)).

Le choix d'un type

repose sur l'utilisation que l'on doit en faire :

- algorithme à appliquer sur ces données
- mode d'accès à un élément spécifique (direct ou séquentiel)
- capacité de s'étendre dynamiquement

Structures de données linéaires

Une structure de linéaire pourra être représentée de 2 façons en mémoire (2 façons différentes d'implémenter ces TADs:

- **Position contigues** : tableaux
- **Éléments liés** : listes chainées, Enregistrements + pointeurs ou références

1. Pile (stack)

Une pile est un **TAD linéaire à accès séquentiel** de type **LIFO** (Last In, First Out). C'est une liste où les éléments sont **ajoutés et retirés à une même extrémité** nommée sommet.

Particulièrement adaptee pour des situations où l'accès à un élément implique d'avoir d'abord traité tous les élements plus récemment ajouté.

- **Applications** : évaluation d'expressions (parsing), sauvegarde des états (undo)
- **Complexité :** Les opérations sur les piles sont toutes en **O(1)**
- **Signature minimale** : empiler (push), dépiler (pop), estVide (isEmpty)

2. File (queue)

Une file est un **TAD linéaire à accès séquentiel** de type FIFO (First In, First Out). C'est une liste où les éléments sont ajoutés à une extrémité (la queue) et retirés à l'autre (la tête).

Ce type est utilisé pour des situations où l'accès à un élément présuppose d'avoir accédé aa tous les éléments ajoutés avant lui (buffers, priorités…)

- **Applications :** Traitement ordonnés, systèmes d'exploitation, réseaux, simulations…
- **Complexité :** Les opérations sur les files sont toutes en **O(1)**
- **Signature minimale :** ajouter (enqueue), retirer (dequeue), estVide (isEmpty)

Il existe plusieurs variantes de files, les plus répandues sont :

- **File de priorité :** chaque élément est muni d'une priorité permettant de déterminer quel élément retirer.
- **Deque (D**ouble **E**nded **Q**ueue) : les opérations peuvent se faire aux 2 extrémités de la file.

3. Listes

Une liste est un **TAD linéaire à accès séquentiel**. Le premier élément d'une liste se nomme **tête**, et le dernier se nomme **queue**. Les piles et les files ne sont en fait que des cas particuliers de listes.

- **Application :** Gestion des collections d'informations homogènes (type de prédilection pour cet usage).
- **Complexité :** Les opérations sur les listes sont entre **O(1)** et **O(n)**
- **Signature minimum :** premier (first), ajouter (add), retirer (pop), successeur (next)

Le fait de pouvoir accéder, depuis un élément, à l'élément suivant grâce à l'opération *successeur* introduit le terme de **liste chainée.** Si on ajoute l'opération *prédécesseur*, on parlera de **liste doublement chainée.** Si on définit la tête de la liste comme étant le successeur de la queue, on obtient une liste circulaire.

Souvent les opérations d'ajout et de retrait sont spécialisées de façon à s'appliquer à un élément précis de la liste.

Algorithmes

Définitions

Algorithme : Un algorithme décrit un traitement sur un nombre fini de **données** (éventuellement aucune). Un algorithme est la composition d'un ensemble fini d' **étapes** , chaque étape étant formée d'un nombre fini d' **opérations** dont chacun est :

- **Définie** de façon rigoureuse et non ambiguë
- **Effective**, c'est-à-dire pouvant être réalisée par une machine (correspond à une action pouvant être réalisée à la main en un temps fini).

De plus, l'algorithme doit être :

- présenté de manière lisible
- pas trop long (décomposition en modules si besoin)
- pas de goto
- favorise la récursivité
- **clairement spécifier le problème résolu**

Déterministe : Toute execution de cet algorithme sur les mêmes données donne lieu à la même suite d'opérations.

Heuristique : méthode (qui ne passe pas par l'analyse détaillée du problème mais par son appartenance ou adhérence à une classe de problèmes déjà identifié) qui fournit rapidement (temps polynomial) une solution réalisable, pas nécéssairement optimale, pour un problème d'optimisation difficile. Utilisé lors d'une complexité « hors du possible » par une approche statistique.

⇨ Analyse heuristique : basée sur l'expérience

Statistique : méthode de la force brute, on tente toutes les combinaisons possible pour trouver la solution la plus optimale.

Complexité : l'analyse de la complexité sert à quantifier les deux grandeurs physiques "**temps d'exécution**" et "**place mémoire**" dans le but de comparer entre eux différents algorithmes qui résolvent le **même** problème.

⇨ Établir des résultats **généraux** permettant d'estimer l'efficacité **intrinsèque** de la méthode utilisée par un algorithme indépendamment de la machine, du langage, du compilateur et de tous les détail d'implémentation pour produire un énoncé du genre :

"sur toute machine, et quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour les données de grande taille" ou encore:

"L'algorithme A est optimal en nombre de comparaisons pour résoudre le problème Q."

Mesure de la complexité

Le temps d'exécution d'un algorithme est toujours proportionnel au **nombre** des **opérations fondamentales** qu'il exécute. Il est alors possible de comparer des algorithmes traitant ce problème selon cette mesure simplifiée. Exemples d'opération fondamentales :

- Nombre de **comparaisons** entre un élément et une liste pour une recherche d'un élément en mémoire vive
- Nombre **d'accès** à la mémoire secondaire pour une recherche en mémoire morte
- Nombre de **comparaisons** et nombre **d'échange** d'éléments pour le tri d'une liste
- Nombre de **multiplications** et **d'additions** pour une multiplication de matrices

Lors du choix d'un algorithme, il faut savoir faire des compromis et prendre en compte le contexte dans lequel le programme sera développé et utilisé.

Comparaison de deux algorithmes

La complexité d'un algorithme est **fonction de la taille des données** ; il est donc important de connaître la rapidité de croissance de cette fonction lorsque la taille des données croît.

Une approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non ou pour comparer entre eux différents algorithmes.

Pour n grand (n **= taille des données**), il est secondaire de savoir si un algo fait $n + 1$ ou $n + 2$ opérations. Les constantes multiplicatives ont-elles aussi peu d'importance.

Algorithme A_1 de complexité $D_1(n) = n^2$
Algorithme A_2 de complexité $D_2(n) = 2n$

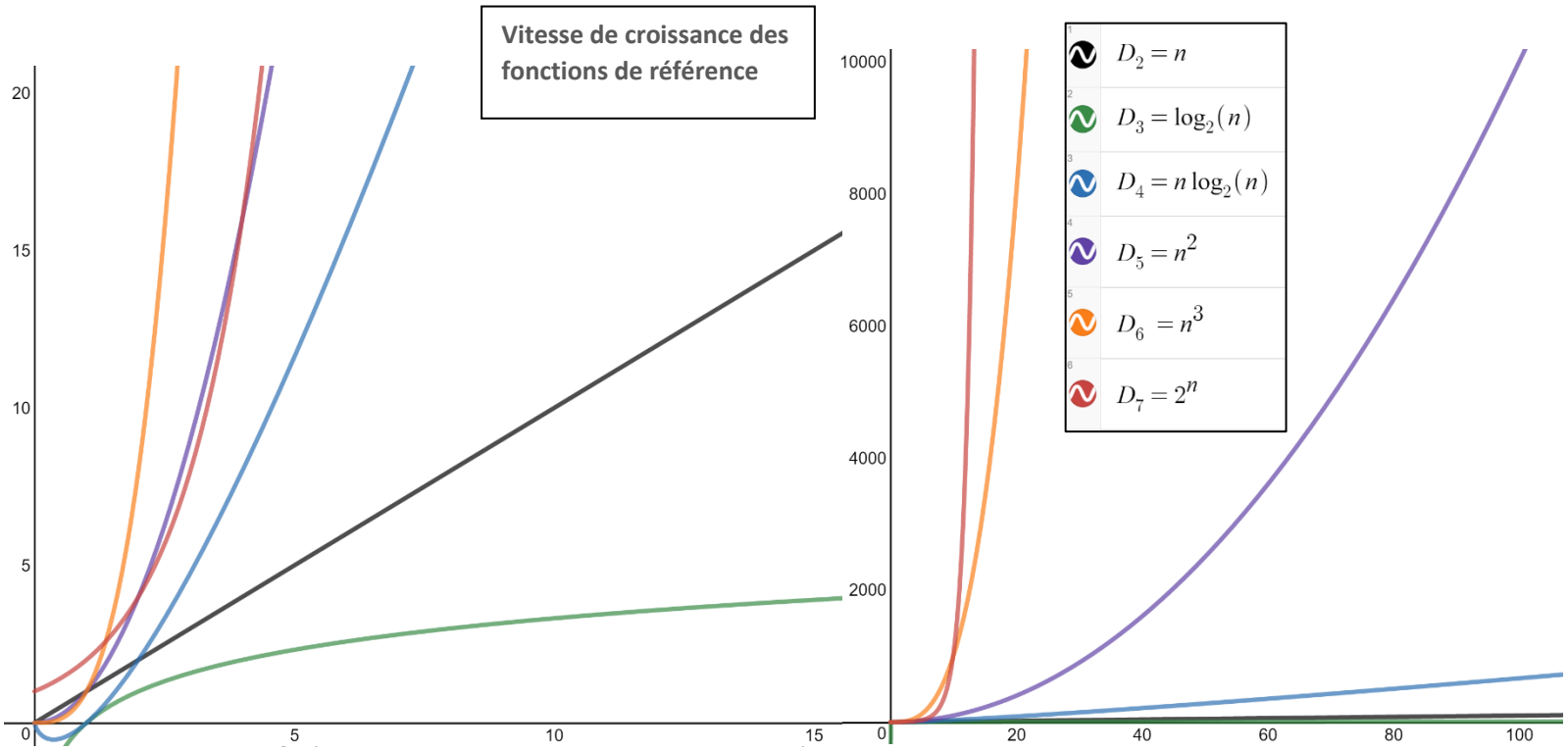
A_2 est meilleur que A_1 pour presque tous les n (dés que $n > 2$) ; de même si $D_1(n) = 3n^2$ et que $D_2(n) = 25n$, A_2 est meilleur que A_1 pour $n > 8$.

Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $D_1(n) = k_1n^2$ et $D_2(n) = k_2n$, l'algo A_2 est toujours meilleur que A_1 à partir d'un certain n, car la fonction $f(n) = n2$ croît beaucoup plus vite que la fonction $g(n) = n$.

⇨ **Ordre de grandeur asymptotique** de $f(n)$ eststrictement plus grand que celui de $g(n)$.

Pour analyser la complexité $D_A(n)$ d'un algorithme A, on détermine l'ordre de grandeur assymptotique de $D_A(n)$: on cherche dans une **échelle de compraison** une fonction qui a une rapidité de croissance voisine de $D_A(n)$.

On dit qu'un algorithme A_1 est meilleur qu'un algorithme A_2, pour n grand, lorsque l'ordre de grandeur de la complexité de A_1 est inférieur à celui de A_2.



les ordres de grandeur asymptotiques des fonctions ***log***$_2$(***n***), ***n***, ***n***log$_2$(***n***), ***n***^{2}, ***n***^{3}, 2^n vont en croissant strictement ; ces fonctions **forment une échelle de comparaison**.

Estimation du temps d'exécution de 7 algorithmes pour différentes tailles ***n*** des données d'un problème sur un ordinateur pouvant effectuer 10^6 opérations par seconde. **Plus n est grand plus l'écart entre les temps d'exécution se creusent.**

Tableau A. Temps d'exécution

Complexité Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	1 µs	6.6 µs	0.1 ms	0.6 ms	10 ms	1 s	4 x 10^{16} a
$n = 10^3$	1 µs	9.9 µs	1 ms	9.9 ms	1 s	16.6 min	∞
$n = 10^4$	1 µs	13.3 µs	10 ms	0.1 s	100 s	11.5 j	∞
$n = 10^5$	1 µs	16.6 µs	0.1 s	1.6 s	2.7 h	31.7 a	∞
$n = 10^6$	1 µs	19.9 µs	1 s	19.9 s	11.5 j	31.7 x 10^3 a	∞

Le tableau B donne une estimation de la taille maximale des données que l'on peut traiter par chacun des algorithmes en temps d'exécution fixé sur le même ordinateur.

Tableau B. Taille maximum des données

Complexité Temps de calcul	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1 s	∞	∞	10^6	63 x 10^3	10^3	100	19
1 min	∞	∞	6 x 10^7	28 x 10^5	7.7 x 10^3	390	25
1 h	∞	∞	36 x 10^8	13 x 10^7	60 x 10^3	15 x 10^2	31
1 jour	∞	∞	86 x 10^9	27 x 10^8	29 x 10^4	44 x 10^2	36

Il est clair que certains algorithmes sont utilisables pour résoudre des problèmes sur un ordinateur et que d'autres ne le sont pas, ou peu utilisable.

Les algorithmes utilisables pour de grande taille sont ceux qui s'exécutent en temps :

- **Constant** : complexité moyenne de certaines méthodes de hachage
- **Logarithmique** : recherche dichotomique ou opérations sur arbres binaires de recherche
- **Linéaire** : recherche séquentielle d'un élément dans un tableau non trié
- **n log(n)** : les bons algorithmes de tri

Les algorithmes qui prennent un temps **polynomial**, cad de l'ordre de n^k avec $k > 0$, ne sont utilisables que pour $k < 2$. Lorsque $2 <= k <= 3$ on ne peut traiter que les problèmes de taille moyenne et si $k > 3$, on ne peut traiter que les petits problèmes.

Les algorithmes en temps **exponentiel**, cad de l'ordre de 2^n sont à peu près inutilisables, sauf pour des problèmes de très petites taille. (***n*** petit).

Le tableau C montre comment la taille des données et le temps d'exécution varient l'un en fonction de l'autre

Tableau C. Evolutions mutuelles du temps et de la taille des données

Complexité	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t + 3.32	10 x t	(10 + ε) x t	100 x t	1000 x t	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	10 x n	(10 - ε) x n	3.16 x n	2.15 x n	n + 3.32

Si on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie presque pas la taille des données que l'on peut traiter avec un algo exponentiel, alors que l'on multiplie par 10 la taille des données traitables par un algo linéaire. **Il faut donc rechercher des algo efficaces, même si les progrès technologiques acroissent les performances du matériel.**