Matière : Analyse d'algorithmes et complexité

Contenu de la matière :

1. Analyse d'algorithmes et complexité.
2. Concepts de base de l'orienté objet.
3. Concepts avancés : généricité, traitement d'exceptions, interfaces …
4. Récursivité.
5. Structures séquentielles: piles, files et listes.
6. Structures hiérarchiques: arbres, arbres binaires, arbres de recherche, les tas et les files de priorité.
7. Algorithmes de tri
8. Les ensembles.

Chapitre I : Analyse d'algorithmes et complexité

- Introduction
- Résolution d'un problème en informatique
- Notion d'algorithme
- Langage algorithmique utilisé
- Complexité des algorithmes
- O-notation
- Règles de calcul de la complexité d'un algorithme
- Complexité des algorithmes récursifs
- Types de complexité algorithmique

Introduction

- La résolution d'un problème informatique nécessite tout un travail de préparation.
- L'ordinateur ne résout pas un problème mais utilisé pour la résolution d'un problème via un programme.
- Ce programme doit être bien conçu (établi de manière à envisager toutes les éventualités d'un traitement).
- **Exemple :**

le problème Div(a,b), n'oubliez pas le cas b=0 !

Résolution d'un problème en informatique

- **Les étapes de résolution d'un problème en informatique :**
  - **Etape 1 :** Définition du problème.

    Il faut de déterminer toutes les informations disponibles et la forme des résultats désirés.
  - **Etape 2 :** Analyse du problème.

    Elle consiste à trouver **le moyen** de passer des données aux résultats.

    →Le résultat de cet étape est **un algorithme.**

    "Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème "
  - **Etape 3 :** Ecriture d'un algorithme avec un langage de description algorithmique.
  - **Etape 4 :** Traduction de l'algorithme dans un langage de programmation.

    → Le résultat de cet étape est **un programme** qui sera exécuter sur une machine.

Remarque : Les étapes 1, 2 et 3 se font sans le recours à la machine.

Résolution d'un problème en informatique

- **Les étapes de résolution d'un problème en informatique :**
  - **Etape 5 : La mise au point du programme** qui sera effectuée en deux étapes :
    1. La correction de l'orthographe, c'est ce qu'on appelle syntaxe dans le jargon de la programmation.
    2. Vérification des résultats renvoyés par le programme. Si les résultats obtenus sont inattendus on doit vérifier les étapes précédentes.

       Résultats inattendus peuvent être à cause de :
       - L'analyse n'est pas bien fait
       - Existence des erreurs de logique.
       - L'algorithme n'a pas été bien traduit.
       - …

Notion d'algorithme

- Un algorithme peut être défini comme suit :
  - **Déf 1.** Résultat d'une démarche logique de résolution d'un problème. C'est le résultat de l'analyse.
  - **Déf 2.** Une séquence de pas de calcul qui prend un ensemble de valeurs comme entrée (input) et produit un ensemble de valeurs comme sortie (output).
- **Propriétés :** Un algorithme doit satisfaire les propriétés suivantes :
  - **Généralité :** un algorithme doit toujours être conçu de manière à envisager toutes les éventualités d'un traitement.
  - **Finitude :** Un algorithme doit s'arrêter au bout d'un temps fini.
  - **Définitude** : Toutes les opérations d'un algorithme doivent être définies sans ambiguïté.
  - **Efficacité :** Un algorithme doit être conçu de telle sorte qu'il se déroule en un temps minimal et qu'il consomme un minimum de ressources
- **Exemples :**
  - Algorithme PGCD (Plus Grand Commun Diviseur) de deux nombres u et v.
  - Algorithmes de tri .
  - Algorithmes de recherche .
  - **…**

Notion d'algorithme

- **Remarque :**

  Certains problèmes n'admettent pas de solution algorithmique exacte et utilisable. On utilise dans ce cas des algorithmes heuristiques qui fournissent des solutions approchées.

  → **L'objet de certaines formations en master et doctorat**

Langage algorithmique utilisé

- Pour décrire un algorithme, on va utiliser un langage algorithmique. L'exemple suivant résume la forme générale d'un algorithme :

```
Algorithme PremierExemple;
  Type TTab = tableau[1..10] de reel ;
  Const Pi = 3.14 ;
  Procédure   Double( x : réel);
  Début
    x ← x * 2 ;
  Fin;
  Fonction Inverse( x : réel) : réel;
  Début
    retourner 1/x ;
  Fin;
  Var i, j, k : entier ;
            T : TTab ;
            S : chaine ;
            R : réel ;
Début
  Ecrire (' Bonjour, donner un nombre entier  < 10 :') ;
  Lire (i) ;
  si (i>10) Alors
     Ecrire ('Erreur : i doit être < 10')
  sinon
     Pour j de 1 à i faire
        Lire(R) ;
        Double(R) ;
        T [j] ← R;
     Fin Pour;
     k ← 1 ;
     Tant que (k <=i) faire
        Ecrire (T [k] * Inverse(Pi)) ;
        k ← k + 1;
     Fin TQ;
      S ←  'Programme terminé' ;
      Ecrire(S) ;
  Fin si
Fin.
```

Complexité des algorithmes

- Le temps d'exécution d'un algorithme dépend des facteurs suivants :
  - Les données utilisées par le programme,
  - La qualité du compilateur (langage utilisé),
  - La machine utilisée (vitesse, mémoire,. . .),
  - **La complexité de l'algorithme lui-même,**
- La complexité d'un algorithme est mesurée indépendamment de la machine et du langage utilisés.
- Elle se mesure uniquement en fonction de la taille des données **n** que l'algorithme doit traiter.
- **Exemple :**
  - dans le cas de tri d'un tableau, n est le nombre d'éléments du tableau.
  - dans le cas de calcul d'un terme d'une suite, n est l'indice du terme, ...etc.

O-notation

- Soit la fonction ***T(n)*** qui représente l'évolution du temps d'exécution d'un algorithme

P en fonction du nombre de données n. par exemple :

$$T(n) = cn^2$$ où c est une constante.

- On dit que la complexité de l'algorithme **P** est : $O(n^2)$, puisque qu'il existe une constante ***c*** positive tel que pour n suffisamment grand on a :

$$T(n) \leq cn^2$$

- Cette notation donne une majoration du nombre d'opérations exécutées (temps d'exécution) par l'algorithme **P**.

- Un algorithme dont la complexité est ***O(f(n))*** est un algorithme qui a ***f(n)*** comme fonction de croissance du temps d'exécution.

Règles de calcul de la complexité d'un algorithme

- **R1 : La complexité d'une instruction élémentaire :**

  Une opération élémentaire est une opération dont le temps d'exécution est indépendant de la taille n des données tel que l'affectation, la lecture, l'écriture, la comparaison ...etc.

  La complexité d'une instruction élémentaire est O(1)

- **R2 : La multiplication par une constante**

$$O(c * f(n)) = O(f(n))$$

  Exemple : $O(\frac{n^3}{4}) = O(n^3)$

- **R3 : La complexité d'une séquence de deux modules**

  La complexité d'une séquence de deux modules *M1* de complexité *O(f(n))* et *M2* de complexité *O(g(n))* est égale à la plus grande des complexité des deux modules : *O(max(f(n), g(n)))*

$$O(f(n)) + O(g(n)) = O(max(f(n), (g(n))$$

Règles de calcul de la complexité d'un algorithme

- **R4 : La complexité d'une instruction conditionnelle**

Si (Cond) Alors *[O(h(n))]*
M1; *[O(f(n))]*
Sinon
M2; *[O(g(n))]*
Fin Si;

La complexité d'une instruction conditionnelle est : $Max\{O(h(n)), O(f(n)), O(g(n))\}$

- **R5 : La complexité d'une boucle**

La complexité d'une boucle est égale à la somme sur toutes les itérations de la complexité du corps de la boucle.

Tant que (Condition) faire [O(h(n))]
P ; [O(f(n))] } [m fois]
Fin TQ;

La complexité de la boucle est : $\sum^{m} Max(h(n), f(n))$

où m est le nombre d'itérations

Règles de calcul de la complexité d'un algorithme

- **Exemple 1 :**

```
Algorithme Recherche;
  Var T : tableau[1..n] de entier ;
      x,i : entier ;
   trouv : booleen ;
Début
    Pour i de 1 à n faire
        Lire (T[i]) ;    [O(1)]
    Fin Pour;
    Lire(x) ;            [O(1)]
    Trouv ←faux ;        [O(1)]
    i ←1 ;               [O(1)]
    Tant que (trouv=faux et i<=n ) faire   [O(1)]
        Si (T[i]=x ) Alors   [O(1)]
            Trouv ←vrai ; [O(1)]
        Fin Si;
        i← i + 1; [O(1)]
    Fin TQ;
    Si (trouv=vrai ) Alors      [O(1)]
            Ecrire (x,'existe')   [O(1)]
    Sinon
            Ecrire (x, 'n"existe pas') [O(1)]
    Fin Si;
Fin.
```

$O(\sum_1^n 1) = O(n)$

[O(1)]

$O(\sum_1^n 1) = O(n)$

O(n)

La complexité de l'algorithme est de O(n) + O(1) + O(n) + O(1) = O(n).

Règles de calcul de la complexité d'un algorithme

- **Exemple 2:** La complexité d'un module

```
Pour i de 1 à n faire
  Pour j de i+1 à n faire
    Si (T[i] > T[j]) Alors  [O(1)]
        tmp ←T[i] ;   [O(1)]
        T[i] ←T[j] ;   [O(1)]
        T[j] ←tmp ;   [O(1)]
    Fin Si;
  Fin Pour;
Fin Pour;
```

[O(1)] $O(\sum_1^{n-i} 1) = O(n-i)$ $O(\sum_{i=1}^{n} n-i))$

La complexité de ce module est : $O(\sum^n (n-i) = O((n-1)+(n-2)\ldots+1)$

$= O(\frac{n(n-1)}{2}) = O(\frac{n^2}{2} - \frac{n}{2}) = O(n^2)$

Règles de calcul de la complexité d'un algorithme

- **Exemple 3: Recherche dichotomique**

```
Lire(x) ; [O(1)]
Trouv← faux ; [O(1)]
Inf ← 1 ; [O(1)]
sup ←n ; [O(1)]
Tant que (trouv=faux et inf<sup) faire   [O(1)]
      m ←(inf + Sup ) div 2 ;  [O(1)]
       Si (T[m]=x) Alors [O(1)]
                Trouv ←vrai [O(1)]
       Sinon
          Si (T[m]<x) Alors [O(1)]
                       inf ←m+1; [O(1)]
          Sinon
                    Sup ← m - 1; [O(1)]
          Fin Si;
       Fin Si;
Fin TQ;
```

[O(1)] [O(1)] $O(log_2(n))$

La complexité de l'algorithme est : $O(log_2(n)) = O(log(n))$

Complexité des algorithmes récursifs

- La complexité d'un algorithme récursif peut être calculée par la résolution d'une équation de récurrence

Exemple 1 :

```
Fonction Fact( n : entier) : entier;        [O(T(n))]
Début
        Si (n <= 1) Alors                   [O(1)]
                Fact ←1 ;                   [O(1)]
        Sinon
                retourner (n * Fact(n - 1));    [O(T(n-1))]
        Fin Si;
Fin;
```

La fonction ***T(n)*** peut être caractérisé par l'équation de récurrence suivante :

$$T(n) = \begin{cases} 1 & si\ n \leq 1 \\ 1 + T(n-1) & si\ n > 1 \end{cases}$$

Complexité des algorithmes récursifs

• Pour calculer la solution générale de l'équation précédente, on peut procéder par substitution :

$$
\begin{aligned}
T(n) &= 1 + T(n-1) \\
&= 1 + \left(1 + T(n-2)\right) \\
&= 2 + T(n-2) \\
&= \cdots \\
&= i + T(n-i) \\
&= \cdots \\
&= n - 1 + T(n-n+1) \\
&= n - 1 + 1 = n
\end{aligned}
$$

Donc la complexité de la fonction Fact est en ***O(n)***

Complexité des algorithmes récursifs

- **Exemple 2 : La complexité temporelle de cette fonction**

```
Fonction F(n : entier):entier      O(T(n))
début
    si n=0 alors                          O(1)
        retourner 2                       O(1)
    sinon
        retourner (F(n-1)*F(n-1))     O(T(n-1)+T(n-1)+1)
    fin si
fin
```

- Pour calculer la complexité T(n), nous pouvons utiliser la récurrence suivante :

$$T(n) = \begin{cases} 1 & si\ n = 0 \\ T(n-1) + T(n-1) + 1 & si\ n > 0 \end{cases}$$

Complexité des algorithmes récursifs

- Nous pouvons procéder par substitution pour obtenir la complexité :

$$T(n) = 2 * T(n-1) + 1$$

$$= 2 * (2 * T(n-2) + 1) + 1$$

$$= 2^2 * T(n-2) + 3$$

$$= 2^2 * (2 * T(n-3) + 1) + 3$$

$$= 2^3 * T(n-3) + 7$$

...

$$= 2^i * T(n-i) + 2^i - 1$$

...

$$= 2^n * T(n-n) + 2^n - 1$$

$$= 2 * 2^n - 1$$

Donc la complexité de la fonction Fact est en $O(2^n)$

Types de complexité algorithmique

- **O(1)** : complexité constante, pas d'augmentation du temps d'exécution quand La taille n du problème croit.
- **O(log(n))** : complexité logarithmique , augmentation très faible du temps d'exécution quand le paramètre croit.

Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).

- **O(n) :** complexité linéaire , augmentation linéraire du temps d'exécution quand la taille de problème croit (si la taille double, le temps double).

Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.

- **O(nlog(n)):** complexité quasi-linéaire , augmentation un peu supérieure à O(n).

Exemple : algorithmes qui décomposent un problème en d'autres plus simples,

traités indépendaments et qui combinent les solutions partielles pour calculer la solution générale.

Types de complexité algorithmique

➢ **O(n^2) :** complexité quadratique , quand le paramètre double, le temps d'exécution est multiplié par 4.

Exemple : quelques algorithmes avec deux boucles imbriquées.

➢ **O(n^i) :** complexité polynomiale, quand le paramètre double, le temps d'exécution est multiplie par 2^i .

Exemple : algorithme utilisant i boucles imbriquées.

➢ **$O(i^n)$:** complexité exponentielle, quand le paramètre double, le temps d'exécution est élevé à la puissance 2.

➢ **O(n!) :** complexité factorielle, asymptotiquement équivalente à n^n

❑ **Remarque :**

- ❖ Les algorithmes de complexité polynomiale ne sont utilisables que sur des données réduites, ou pour des traitements ponctuels.
- ❖ Les algorithmes exponentiels ou au delà ne sont pas utilisables en pratique.

Types de complexité algorithmique

➢ **Temps de calcul pour des données de taille 1 million**

Complexité / Vitesse(hertz)	$\log(n)$	n	n^2	2^n
10^6	0.013 ms	1s	278 heures	10000 ans
10^9	0.013 µs	1 ms	¼ heure	10 ans
10^{12}	0.013 ns	1µs	1s	1 semaine

➢ **Conclusion :**

- Selon la Loi de Moore, la rapidité des processeurs double tous les 18 mois (les capacités de stockage suivent la même loi).
- Le volume des données stockées dans les différents systèmes augmente de façon exponentielle.
- il vaut mieux d'optimiser ses algorithmes qu'attendre des années qu'un processeur surpuissant soit inventé.