Université Bordeaux

Licence Informatique et Maths/Info : Semestre 3

Algorithmes et structures de données 1

Carole Blanc Année 2014-2015

Complexité

Définition 1.1 L'**efficacité** d'un algorithme est mesurée par son coût (**complexité**) en temps et en mémoire.

La complexité d'un algorithme se mesure en calculant :

- le **nombre d'opérations** élémentaires,
- la **taille de la mémoire** nécessaire,

pour traiter une donnée de taille **n**.

1-Introduction

- Complexité
- Récursivité
- Type abstrait
- Containeur
- Implémentation

Complexité

On considèrera dans ce cours que la complexité des instructions élémentaires les plus courantes sur un ordinateur a un temps d'exécution constant égal à 1.

Centre d'intérêt pour l'algorithmique c'est **l'ordre de grandeur** au voisinage de l'infini de la fonction qui exprime le nombre d'instructions ou la taille de la mémoire.

Question : L'infini de quoi ?

Complexité

Définition 1.2 On définit les trois complexités suivantes :

- Complexité dans **le pire** des cas :
  $C^{>}_A(n)=\max\{C_A(d), d \text{ donnée de taille } n\}$
- Complexité dans **le meilleur** des cas :
  $C^{<}_A(n)=\min\{C_A(d), d \text{ donnée de taille } n\}$
- Complexité **en moyenne** :

$$\overline{C}_A(n) = \sum_{d \text{ instance de } A} \Pr(d) C_A(d)$$

où **Pr(d) est la probabilité** d'avoir en entrée une instance d parmi toutes les données de taille n.

Récursivité

Définition 1.3:
Lorsqu'un algorithme contient un **appel à lui-même**, on dit qu'il est **récursif.**
Lorsque deux algorithmes s'appellent l'un l'autre on dit que la **récursivité** est **croisée**

Complexité
Un algorithme récursif nécessite de **conserver les contextes récursifs des appels**. La récursivité peut donc conduire à une complexité mémoire plus grande qu'un algorithme itératif.

Complexité

Cas Particulier : Problème **NP-complet**
C'est un problème pour lequel on ne connait pas d'algorithme correct efficace : réalisable en temps et en mémoire.
L'ensemble des problèmes NP-complets ont les propriétés suivantes :

- Si on trouve un algorithme efficace pour un problème NP complet alors il existe des algorithmes efficaces pour tous,
- Personne n'a jamais trouvé un algorithme efficace pour un problème NP-complet,
- Personne n'a jamais prouvé qu'il ne peut pas exister d'algorithme efficace pour un problème NP-complet particulier.

Le plus célèbre est le **problème du voyageur de commerce**.

Récursivité

Exemple : calcul de la fonction factorielle :

```
fonction facRecur(val :entier):entier;
  début
    si n==0 alors
      retourner(1)
    sinon
      retourner(n*facRec(n-1))
    finsi
  fin
finfonction
```

Récursivité

Exemple : calcul de la fonction factorielle

```
fonction facIter(val n:entier):entier;
  début
    var i,p:entier;
    p=1:
    pour i=2 à n faire
      p=p*i;
    finpour
    retourner(p)
  fin
Finfonction
```

La fonction **factIter** est meilleure en temps et mémoire (voir).

Récursivité Terminale

L'algorithme facRecur ne présente pas de récursivité terminale.

```
fonction facRecur(val n:entier):entier;
  début
    si n==0 alors
      retourner(1)
    sinon
      retourner(n*facRec(n-1))
    finsi
  fin
finfonction
```

Récursivité Terminale

Définition 1.4 : Un algorithme récursif présente une **récursivité terminale** si et seulement si la valeur retournée par cet algorithme est une valeur fixe, ou une valeur calculée par cet algorithme.

L'algorithme *facRecur* ne présente pas de récursivité terminale.

Récursivité Terminale

Exemple

```
fonction facRecurTerm(val n:entier;
                      val res:entier):entier;
  début
    si n==0 alors
      retourner(res)
    sinon
      retourner(facRecurTerm(n-1,n*res))
    finsi
  fin
finfonction
```

L'algorithme facRecurTerm présente une récursivité terminale.
La factorielle se calcule par l'appel facRecurTerm(n,1)

Récursivité Terminale

Intérêt : les compilateurs détectent cette propriété et optimisent le stockage de l'environnement de la fonction.

Ainsi `facRecurTerm` aura une complexité identique à `facIter`.

ATTENTION.
Dans le cas d'un algorithme présentant deux appels récursifs, rendre la récursivité terminale ne permet pas obligatoirement au compilateur d'obtenir une complexité inférieure.

Type abstrait

Exemple
Les nombres complexes ne sont pas des types de bases. On peut les définir comme un type abstrait :

- **nom** : *nombreComplexe*
- **ensemble de valeur** : *réel×réel*
- **primitives :**
  - **multiplication** :
    `(nombreComplexe × nombreComplexe) → nombreComplexe`
  - **addition** :
    `(nombreComplexe × nombreComplexe) → nombreComplexe`
  - **module** :
    `nombreComplexe → réel`

Type abstrait

Définition 1.5 : Un **type abstrait** est un triplet composé :
- d'un **nom**,
- d'un **ensemble de valeurs**,
- d'un **ensemble d'opérations** (souvent appelé primitives) définies sur ces valeurs.

D'un point de vue complexité, on considère que les **primitives** (à part celle d'initialisation si elle existe) ont une **complexité en temps et en mémoire en O(1).**
Pour désigner un type abstrait on utilise une chaine de caractères.

Containeur

Définition 1.6 : Un **containeur** est un type abstrait permettant de représenter des collections d'objets ainsi que les opérations sur ces objets.

Les collections que l'on veut représenter peuvent être ordonnées ou non, numériques ou non.
L'ordre est parfois fourni par un évènement extérieur.
Les collections d'objets peuvent parfois contenir des éléments identiques.

Containeur

Primitives :

- **Accès**
  - **valeur** : containeur → objet
- **Modification**
  - **creerContaineur**: containeur → vide
  - **ajouter** : containeur X objet → vide
  - **supprimer** : containeur X objet → vide
  - **detruireContaineur** : containeur → vide

Exemple

Un ensemble de nombres complexes peut être défini par un containeur dont les objets sont des nombreComplexe.

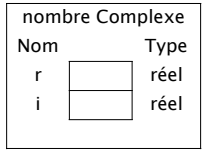
Implémentation

Exemple

Le type abstrait nombreComplexe peut être implémenté de la manière suivante :

```
nombreComplexe=structure
    r:réel;
    i:réel;
Finstructure

var c : nombreComplexe;
```



Nom variable	Type
c	nombreComplexe
c.r	réel
c.i	réel

Implémentation

Définition 1.7 : L'implémentation consiste à choisir une structure de données et les algorithmes associés pour réaliser un type abstrait

La structure de données utilisée pour l'implémentation peut elle-même être un type abstrait.

L'implémentation doit respecter la complexité des primitives à part celle d'initialisation (celle-ci ne s'exécutera qu'une fois).

Implémentation

Exemple

Le type abstrait nombreComplexe peut être implémenté de la manière suivante :

```
nombreComplexe=structure
      r:réel;
      i:réel;
Finstructure

fonction op::*(val a,b:nombreComplexe):nombreComplexe;
  var c:nombreComplexe;
  début
    c.r=a.r*b.r-a.i*b.i;
    c.i=a.r*b.i+a.i*b.r;
    retourner(c)
```

Implémentation

Exemple

```
fonction op::+(val a,b:nombreComplexe):nombreComplexe;
  var c:nombreComplexe;
  début
    c.r=a.r+b.r;
    c.i=a.i+b.i;
    retourner(c)
  fin

fonction module(val a:nombreComplexe):réel;
  début
    retourner(sqrt(a.r*a.r+a.i*a.i))
  fin
```

Implémentation

Modification

```
fonction creerContaineur(ref C:containeur de
                nombreComplexe):vide;
  var i:entier;
  début
    pour i allant de 1 à N faire
      C[i].b=faux;
    finPour;
  fin
```

C

	1	2	3	4	5	…		N	
v									nombreComplexe
b	F	F	F	F	F	F…F	F	F	booléen

Complexité : ?

Implémentation

Exemple

Un containeur de nombreComplexe peut être implémenté par un tableau de nombreComplexe.

```
containeur d'objet=tableau[1..N]de structure
     v:objet;
     b:booleen;
Finstructure
```

T

	1	2	3	4	5	6		N	
v									objet
b									booléen

Implémentation

Modification

```
fonction ajouter(ref C: containeur de
          nombreComplexe;val x:nombreComplexe):vide
  var i:entier;
  début
    i=1;
    tant que i<n et C[i].b faire
      i=i+1;
    fintantque
    si i<=n alors
      C[i].v=x;
      C[i].b=vrai
    finsi
  fin
```

C

	1	2	3	4	5	…		N	
v	c1	c21	c5	c2					nombreComplexe
b	V	V	V	V	F	F…F	F	F	booléen

Complexité : ?

Implémentation

Modification

```
fonction supprimer(ref C: containeur de
          nombreComplexe;val x:nombreComplexe):vide
  var i:entier;
  début
    i=1;
    tant que i<=n et C[i].v!=x faire
      i=i+1;
    fintantque
    si i<=n alors
      C[i].b=faux
    finsi
  fin
```

C

	1	2	3	4	5…		N	
v	c1	c21	c5	c2				nombreComplexe
b	V	V	F	V	F…F	F	F	booléen

Complexité : ?

Implémentation

Accès

```
fonction valeur(ref C : containeur de
nombreComplexe):nombreComplexe;
/* retourne le 1er nombre complexe présent/*
  var i:entier;
  début
    i=1;
    tant que i<=n et !C[i].b faire
      i=i+1;
    fintantque
    si i<=n alors
      retourner(C[i].v)
    sinon
      retourner(NULL)
    finsi
  fin
```

2 conditions d'arrêt

Test en sortie de boucle

Complexité : ?

Implémentation

Modification

```
fonction detruireContaineur(ref C : containeur
de nombreComplexe):vide
  début
    pour i allant de 1 à N faire
      C[i].b=faux;
    finPour;
  fin
```

Complexité : ?