

Structures de données (LU2IN006)

Nawal Benabbou

Licence Informatique - Sorbonne Université

2022-2023



Présentation et organisation de l'UE

Contenu de l'UE

Présenter de manière synthétique les structures de données couramment utilisées en informatique afin de connaître les clefs permettant de choisir optimalement les structures les plus adaptées pour résoudre un problème donné.

Quelques références bibliographiques :

- C. Cormen, C. Leiserson, R. Rivest. "Introduction à l'algorithmique", Dunod, 1994.
- A. Aho, J. Ullman. "Concepts fondamentaux de l'informatique", Dunod, 1993.
- M. Divay. "Algorithmes et structures de données génériques", Dunod, 1999.
- R. Malgouyres, R. Zrour, F. Feschet. "Initiation à l'algorithmique et aux structures de données en C", 2008.

Déroulement de l'UE

Cours : 11 séances de 1h45 (mercredi 16h-17h45)

TD/TME :

- 11 séances, en décalage d'une semaine avec le cours
- Chaque séance : TD de 1h45 et TME de 1h45 (en langage C)

Documents et informations : sur Moodle.

	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
LUNDI																								
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
MARDI																								
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
MERCREDI																								
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
JEUDI																								
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
VENDREDI																								
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	
	8h30	9h00	9h30	10h00	10h30	11h00	11h30	12h00	12h30	13h00	13h30	14h00	14h30	15h00	15h30	16h00	16h30	17h00	17h30	18h00	18h30	19h00	19h30	

Salles

Groupe	TD	TME
TD/TME1	24.34.101 du 25/01 au 19/04	1415-302 du 16/01 au 09/05
TD/TME2	24.34.104 du 24/01 au 18/04	1415-408 du 16/01 au 09/05
TD/TME3	14.24.110 du 25/01 au 19/04	2425-304 du 16/01 au 09/05
TD/TME4	ATRIUM.425 le 26/01 ATRIUM.523 du 02/02 au 20/04	1415-301 du 16/01 au 09/05
TD/TME5	23.24.105 du 24/01 au 18/04	1415-301 du 16/01 au 09/05
TD/TME6	14.24.205 du 23/01 au 17/04 24.25.102 le mardi 09/05	1415-307 (305 en janvier) ? le mardi 09/05
TD/TME7	23.24.212 du 23/01 au 17/04 14.15.103 le mardi 09/05	1415-408 du 16/01 au 09/05
TD/TME8	54.55.204 du 25/01 au 19/04	1415-307 du 16/01 au 09/05
TD/TME9	14.24.203 du 25/01 au 19/04	1415-401 du 16/01 au 09/05
TD/TME10	23.24.203 du 27/01 au 21/04	2425-303 du 16/01 au 09/05

Équipe pédagogique

Responsable : Nawal Benabbou

Chargés de cours :

- Nawal Benabbou (séances 1,2,7,8,9 et 10)
- Pierre-Henri Wuillemin (séances 3,4,5,6 et 11)

Groupes	Chargé de TD	Doublure(s) de TME
1	Hadrien Notarantonio	Ayush Pandey
2	Gabriel Le Bouder	Syrine Saidi
3	Mohamed Ouaguenouni	Mathieu Pont
4	Charles Meyer-Hilfiger	Quentin Ferro
5	Adel Nabli	Jean-Gabriel Ganascia, Jeremie Pantin
6	Paul Hermouet	Louis Serrano
7	Adel Nabli	Syrine Saidi, Hadrien Notarantonio
8	Jeremie Pantin	Quentin Ferro
9	Nawal Benabbou	Tristan Luiggi, Alper Er
10	Gabriel Le Bouder	Kim Vallée

Contacts : nawal.benabbou@lip6.fr, pierre-henri.wuillemin@lip6.fr, nabli@isir.upmc.fr, hadrien.notarantonio@lip6.fr, gabriel.le-bouder@lip6.fr, jean-gabriel.ganascia@lip6.fr, mohamed.ouaguenouni@lip6.fr, quentin.ferro@lip6.fr, jeremie.pantin@lip6.fr, paul.hermouet@lip6.fr, kim.vallee@lip6.fr, saidi@newmail.lip6.fr, mathieu.pont@lip6.fr, ayush.pandey@lip6.fr, charles.meyer-hilfiger@inria.fr, luiggi@isir.upmc.fr, alper.er@sorbonne-universite.fr, serrano@isir.upmc.fr.

Évaluation

Contrôle des connaissances

- Examen final : 50%
- Un projet : 40%
- Contrôle continu : 10%

Sur le projet

Le projet est à réaliser pendant les séances 5 à 10 de TME, avec une soutenance prévue en séance 11.

Attention :

- La note du projet est conservée en deuxième session.
- Le projet doit obligatoirement être fait **en binôme**.

Sur le contrôle continu

- Pas de partiel
- Une interrogation écrite réalisée pendant une séance de TD

L'abstraction

Une structure de données est une mise en œuvre concrète d'un type abstrait de données. Qu'est-ce que l'abstraction ?

Définition générale (Larousse)

Opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment de ses autres caractères.

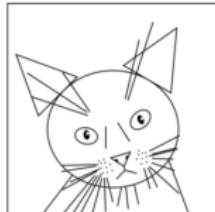
Quelques types d'abstraction :

- **L'abstraction par simplification** permet de réduire la description d'un objet en ne conservant que les informations utiles ou importantes.

Exemples : un plan de métro, une fiche personnage.

- **L'abstraction par généralisation** permet d'aller du particulier vers le général, d'un objet à une classe d'objets plus générale.

Exemples : les hommes sont des vertébrés, un bureau est un meuble.



L'abstraction



La trahison des images. René Magritte (1929)

L'abstraction en informatique

Abstraction informatique

L'abstraction en informatique consiste à synthétiser des caractéristiques et traitements communs, applicables à des entités ou concepts variés, afin de simplifier et d'unifier leur manipulation.

Exemples

- Un ordinateur peut se définir comme un ensemble de composants électroniques, et ce sont les couches progressives d'abstraction (noyau, système d'exploitation...) qui permettent à l'utilisateur de ne pas avoir à gérer directement les aspects électroniques lorsqu'il manipule l'ordinateur.
- Les langages informatiques sont des abstractions, faisant correspondre le langage machine à un langage plus compréhensible par l'homme.
- Les variables sont des abstractions, permettant de représenter par un symbole la valeur qu'elle contient, qui peut être une structure complexe.
- Les procédures sont des abstractions, permettant de résumer en une instruction des choses qui peuvent être complexes à être réaliser.

Attention : l'abstraction est une bonne chose (codes généraux, faciles à lire), mais cela ne doit pas rendre le programme plus lent !

Type abstrait de données

Définition

Un *type abstrait de données* est une spécification mathématique d'un ensemble d'objets et d'un ensemble d'opérations applicables à ces objets.

Exemples :

- les nombres entiers, munis des opérations $+, *$ etc.
- les chaînes de caractères, munis des opérations de concaténation, d'insertion, etc.
- les listes, munis des opérations d'insertion, d'accès à des éléments, etc.
- les ensembles, munis des opérations d'union, d'intersection, etc.

Remarque :

Un type abstrait de données ne définit pas la façon dont les données sont stockées, ni la manière d'implémenter les méthodes. Définir un type abstrait permet de manipuler des objets sans connaître leur représentation dans la mémoire de l'ordinateur (codage, implémentation), et donc d'élaborer des algorithmes de manière abstraite, conduisant à des programmes compréhensibles et réutilisables.

Structure de données

Définition

Une structure de données est une représentation concrète dans la mémoire d'un ordinateur des objets décrits par un type abstrait. Elle est accompagnée de l'implémentation des opérations sur cette représentation.

Exemple : un booléen est représenté par 1 octet en C++ contre 1 bit en java...

Coût associé au choix d'une structure de données

Le choix de la structure de données a plusieurs conséquences sur les performances des programmes :

- La déclaration d'un objet a un coût en mémoire (en octets). On parlera de complexité spatiale.
- Les opérations sur les objets ont un coût en temps de calcul. On parlera de complexité temporelle (mesurée en nombre d'opérations élémentaires).

⇒ Toutes les structures de données n'ont pas les mêmes performances, ni les mêmes cas d'utilisation.

Cette UE a pour but d'apporter une connaissance des principales structures de données, afin d'apprendre dans quel cas il convient d'utiliser une structure de données plutôt qu'une autre, et de savoir comment les implémenter.

Qu'est-ce qu'un programme ?

Un programme est composé de :

- un ensemble fini de variables, dont des variables d'entrée
- un ensemble fini d'opérations élémentaires
- un ensemble fini de structures de contrôle

Variable

Une variable est une "boîte" ayant un nom et contenant une valeur appartenant à un domaine. Une variable peut correspondre à un type simple (entier, flottant), ou à une structure plus élaborée.

Opération élémentaire

Une opération élémentaire est une commande informatique d'un langage portant sur les variables et réalisant une opération simple (arithmétique, logique, affectation, etc.) qu'un ordinateur réalise en un nombre connu d'appels au processeur (CPU). **Exemples :** $a = a + b$; $i > j$; $b = 2$;

Structure de contrôle

Une structure de contrôle est une instruction pouvant dévier le flux de contrôle du programme lors de son exécution. En programmation impérative, elle est construite à partir de structures élémentaires (tests, boucles...).

Qu'est-ce qu'un algorithme ?

Un algorithme est un programme associé à un problème et vérifiant certaines propriétés (terminaison, validité).

Problème

Un problème se définit à partir de paramètres, qui prennent des valeurs dans un domaine, et d'une question.

Exemples :

- À partir d'un tableau, construire un tableau trié.
- Trouver le plus court chemin dans une carte routière.
- Découper du tissu pour faire des vêtements en minimisant les chutes.
- Identifier les contours d'un visage sur une photo.

Définition : algorithme

Un algorithme est un programme résolvant un problème donné

- en un nombre fini d'opérations élémentaires (terminaison).
- en donnant une réponse juste à la question du problème, quelque soit les valeurs des variables d'entrée (validité).

Performances d'un algorithme

Complexité temporelle

La complexité temporelle d'un algorithme décrit le temps nécessaire à son exécution. Elle est estimée en comptant le nombre d'opérations élémentaires effectuées lors d'une exécution de l'algorithme.

Pour la plupart des algorithmes, le nombre d'opérations élémentaires exécutées dépend de la valeur des paramètres d'entrée.

“Pire des cas” et “Meilleur des cas”

Le cas où un algorithme effectue le nombre maximum d'opérations élémentaires possibles est communément appelé le “pire des cas”. Le “meilleur des cas” correspond quant à lui au cas où il effectue le moins d'opérations élémentaires possibles.

Illustration sur un exemple

Prenons l'exemple de la recherche d'une valeur dans un tableau.

```
1 int recherche_tab(int val, int* tab, int n){  
2     int i=0;  
3     while ((i<n) && (tab[i]!=val)){  
4         i=i+1; //ou encore i++;  
5     }  
6     if (i<n){  
7         return 1;  
8     }else{  
9         return 0;  
10    }  
11 }
```

Quel est le meilleur des cas ?

Le meilleur des cas correspond au cas où l'élément recherché est le premier élément du tableau. On réalise alors une initialisation (ligne 2), deux tests (lignes 3 et 6) et le retour de la fonction (ligne 7) \Rightarrow 4 opérations élémentaires.

Quel est le pire des cas ?

Le pire des cas correspond à la recherche d'un entier non présent dans le tableau. Dans ce cas, on réalise une initialisation (ligne 2), $n+2$ tests (lignes 3 et 6), n additions (ligne 4) et le retour de la fonction (ligne 9) \Rightarrow $2n+4$ opérations élémentaires.

Complexité temporelle et notations de Landau

Ce n'est pas toujours possible de donner précisément la complexité d'un algorithme (car le nombre d'opérations effectuées peut dépendre de beaucoup de paramètres). Dans ce cas, on cherche à fournir des bornes sur la complexité temporelle, et on utilise les notations de Landau.

Définition : notation de Landau O (borne supérieure)

Soit $f(n)$ et $g(n)$ deux fonctions positives définies sur \mathbb{N} . On dit que $f(n)$ est en $O(g(n))$ si la fonction f est asymptotiquement bornée supérieurement par la fonction g à un facteur près. Plus formellement, $f(n)$ est en $O(g(n))$ s'il existe un entier n_0 et un réel $k > 0$ tels que pour tout $n \geq n_0$:

$$f(n) \leq kg(n)$$

Retour à l'exemple de recherche dans un tableau

Pour cet algorithme, notons $f(n)$ le nombre d'opérations effectuées pour un tableau de taille n . Nous avons vu que, dans le pire des cas, l'algorithme effectue (environ) $2n+4$ opérations élémentaires. Donc, en posant $g(n) = n$, et par exemple $k = 6$ et $n_0 = 1$, on a bien $f(n) \leq k \times g(n)$ pour tout $n \geq n_0$. On en conclut que l'algorithme a une complexité temporelle en $O(n)$.

Complexité temporelle et notations de Landau

Définition : notation de Landau Ω (borne inférieure)

Soit $f(n)$ et $g(n)$ deux fonctions positives définies sur \mathbb{N} . On dit que $f(n)$ est en $\Omega(g(n))$ si la fonction f est asymptotiquement bornée inférieurement par la fonction g à un facteur près. Plus formellement, $f(n)$ est en $\Omega(g(n))$ s'il existe un entier n_0 et un réel $k > 0$ tels que pour tout $n \geq n_0$:

$$kg(n) \leq f(n)$$

Retour à l'exemple de recherche dans un tableau

Notons $f(n)$ le nombre d'opérations élémentaires effectuées pour un tableau de taille n . Nous avons vu que, dans le meilleur des cas, l'algorithme effectue (environ) 4 opérations élémentaires. Donc, en posant $g(n) = 1$, $k = 1$ et $n_0 = 0$, on a $k \times g(n) \leq f(n)$ pour tout $n \geq n_0$, ce qui signifie que la complexité temporelle de cet algorithme est en $\Omega(1)$.

Complexité temporelle et notations de Landau

Définition : notation de Landau Θ (borne inférieure et supérieure)

Soit $f(n)$ et $g(n)$ deux fonctions positives définies sur \mathbb{N} . On dit que $f(n)$ est en $\Theta(g(n))$ si la fonction f est dominée et soumise à g asymptotiquement. Plus formellement, $f(n)$ est en $\Theta(g(n))$ s'il existe un entier n_0 et deux réels $k_1, k_2 > 0$ tels que pour tout $n \geq n_0$:

$$k_1 g(n) \leq f(n) \leq k_2 g(n)$$

Remarque :

Quand on évalue la complexité temporelle d'un algorithme avec des notations de Landau, on peut en pratique ignorer les :

- opérations élémentaires hors des boucles car elles sont en nombre fini,
- tests de boucle car ils ajoutent un nombre fini d'opérations par tour.

Exemple

```
1  double mystere(int n){  
2      int i,j;  
3      double f,s;  
4      s=0;  
5      for (i=1;i<=n;i++){  
6          f=1;  
7          for (j=1;j<=i;j++){  
8              f=f*j;  
9          }  
10         s=s+f;  
11     }  
12     return s;  
13 }
```

Que fait cet algorithme ?

Il permet de calculer la somme des factorielles, c'est-à-dire : $\sum_{i=1}^n i!$

Quelle est sa complexité temporelle ?

La complexité de cet algorithme est $\Theta(n^2)$ car, dans tous les cas, le nombre d'opérations élémentaires réalisées est de l'ordre de :

$$\sum_{i=1}^n \left(2 + \sum_{j=1}^i 1\right) = \sum_{i=1}^n (2+i) = 2n + \frac{n \times (n+1)}{2} = \frac{1}{2}n^2 + \frac{5}{2}n \Rightarrow \Theta(n^2)$$

Une autre exemple

```
1 double mystere2(int n){  
2     int j;  
3     double f, s;  
4     s=0; f=1;  
5  
6     for (i=1; i<=n; i++){  
7         f=f*i;  
8         s=s+f;  
9     }  
10    return s;  
11 }
```

Que fait cet algorithme ?

Cet algorithme permet lui aussi de calculer la somme des factorielles.

Quelle est sa complexité temporelle ?

Sa complexité est en $\Theta(n)$ car son nombre d'opérations élémentaires est dans tous les cas de l'ordre de $2n$ (2 opérations élémentaires par tour de boucle).

⇒ Comme ces deux algorithmes permettent de résoudre le même problème, on préférera le deuxième au premier car il a une meilleure complexité temporelle.

Performances d'un algorithme

Pourquoi est-ce que la complexité temporelle est importante ?

Un petit exemple

Considérons un ordinateur capable de réaliser 10^9 opérations élémentaires par seconde (processeur avec une fréquence de 1 GigaHertz). Notons $f(n)$ le nombre d'opérations élémentaires réalisées par un programme, où n est la taille de l'instance (par exemple, la taille d'un tableau). Quand $n = 1000$, combien de temps faut-il pour exécuter un programme de complexité :

- $f(n) = \Theta(n^2)$? 1×10^{-3} seconde
- $f(n) = \Theta(n^3)$? 1 seconde
- $f(n) = \Theta(n^4)$? 16.7 secondes
- $f(n) = \Theta(n^5)$? 11.6 jours
- $f(n) = \Theta(2^n)$? 3.39×10^{282} siècles

Quand on effectue des opérations sur des données, le choix des structures de données a un impact sur le nombre d'opérations élémentaires effectuées. Il est primordial de savoir comment choisir les structures de données selon l'application considérée, pour réduire les temps de calcul des programmes.

Une autre mesure de performance : la complexité spatiale

On peut aussi vouloir évaluer l'efficacité d'un algorithme selon son occupation mémoire. En effet, il faut pouvoir déterminer une implémentation permettant de ne pas dépasser l'espace mémoire d'un ordinateur.

Définition : complexité spatiale

La complexité spatiale d'un algorithme est une estimation de l'espace mémoire occupé lors de son exécution.

⇒ La complexité spatiale peut être indiquée avec les notations de Landau par rapport à la valeur des variables d'entrée (comme la complexité temporelle).

Remarque

Les complexités temporelles et spatiales sont fréquemment en opposition, et dans ce cas, il faut déterminer un compromis entre les deux.

Exemple

```
1 double mystere3(int n){  
2     int f=1;  
3     for (i=2;i<=n;i++){  
4         f=f*i;  
5     }  
6     return f;  
7 }
```

```
1 double mystere4(int n){  
2     if (n <= 1){  
3         return 1;  
4     }else{  
5         return n*mystere4(n-1);  
6     }  
7 }
```

Que font ces algorithmes ?

Ces deux algorithmes permettent de calculer la factorielle d'un nombre ($n!$). Le premier algorithme (`mystere3`) donne une version itérative, alors que le deuxième (`mystere4`) donne une version récursive.

Quelles sont leurs complexités temporelles et spatiales ?

Ces deux algorithmes ont la même complexité temporelle : $\Theta(n)$. Cependant, leur complexité spatiale est différente :

- La version itérative utilise un nombre constant de cases mémoire (au plus 4) \Rightarrow Complexité spatiale : $\Theta(1)$
- La version récursive a besoin de 2 cases mémoire par appel, donc $2n$ au total \Rightarrow Complexité spatiale : $\Theta(n)$.

Pour résumer

Objectif de l'UE

L'objectif de ce module est tout à la fois :

- Étudier les différents types abstraits et structures de données.
- Comprendre leur vitesse et leur occupation mémoire.
- Apprendre à les implémenter correctement (en langage C).
- Savoir identifier dans quelles situations une structure de données est préférable à une autre.

Remarque

Cette étude est réalisée avec le langage C, mais elle pourrait être faite dans d'autres langages, et elle vous sera utile quel que soit le langage !

⇒ Avant la première séance de TD/TME, lisez les rappels de C mis en ligne sur moodle !