Remise à niveau
Algorithmique et programmation

INSA de Toulouse, 3 MIC/IMACS

Antonin Lentz

2023

Table des matières

I Notions fondamentales **5**

1 Variables et types **6**
1.1 Définition 6
1.2 Implémentation 6

2 Instructions **8**
2.1 Définition 8
2.2 Implémentation 9
2.3 Exercices 10

3 Branchement conditionnel **11**
3.1 Définition 11
3.2 Implémentation 12
3.3 Exercices 13

4 Boucle **14**
4.1 Définition 14
4.2 Implémentation 15
4.3 Exercices 16

5 Sous algorithme **17**
5.1 Définition 17
5.2 Implémentation 18
5.3 Exercices 18

II Notions avancées **19**

6 Approche ascendante et descendante **20**
6.1 Définition 20
6.2 Exercices 21

7 Récursivité **23**
7.1 Définition 23
7.2 Implémentation 24

7.3 Exercices 24

8 Tableaux **25**
8.1 Définition 25
8.2 Exemples de parcours de tableau. 25
8.3 Implémentation 27
8.3.1 En Python 27
8.3.2 En C 28
8.4 Exercices 29

9 Complexité **30**
9.1 Définition 30
9.2 Ordres de grandeur 32
9.3 Exercices 32

10 Passage par valeur ou par référence **33**
10.1 Définition 33
10.2 Implémentation 34

III Structures de données **36**

11 Types composés **37**
11.1 Définition 37
11.2 Implémentation 37

12 Types abstraits **39**
12.1 Opérations élémentaires 39
12.1.1 Premier exemple : le type abstrait Rationnel. 39
12.1.2 Deuxième exemple : le type abstrait Tableau. 39
12.2 Implémentation 40
12.3 Algorithmique 40
12.4 Exercices 40

13 Piles et files **41**
13.1 Piles 41
13.1.1 Type abstrait 41
13.1.2 Implémentation des piles par chaînage 41
13.1.3 Implémentation des piles à partir de tableaux 43
13.1.4 Algorithmique 44
13.2 Files 46
13.2.1 Type abstrait 46
13.2.2 Implémentation des files par chaînage 46
13.2.3 Des files avec des piles 47
13.2.4 Algorithmique 49
13.3 Exercices 49

14 Listes chaînées **51**
14.1 Définition 51
14.2 Algorithmique 52
14.3 Implémentation 52
14.4 Exercices 54

15 Structures arborescentes **56**
15.1 Arbres binaires 56
15.2 Arbres binaires étiquetés 57
15.3 Arbres binaires de recherche 58
15.4 Exercices 58

Introduction

Dans ce cours, on commence par présenter les concepts algorithmiques à l'aide d'un pseudo langage. Puis on développe les problématiques liées à l'implémentation. Cette dernière est généralement faite en C et en Python, mais ce cours n'est pas une remise à niveau dans ces langages. Il n'est pas nécessaire de comprendre toutes les subtilités du C, vous aurez un cours prochainement. De plus, les exemples fournis ne sont pas des programmes, i.e. des codes complets que l'on pourrait exécuter. Des éléments sont omis pour rester clair et succinct.

Dans une première partie, on commence par détailler les notions permettant de définir des algorithmes simples. Ces notions n'étant pas suffisantes pour résoudre des problèmes complexes, la deuxième partie introduit des techniques plus avancées. Elle introduit entre autre la manipulation de collections de données à partir de tableaux. Cette structure étant limitée, la troisième partie en introduit d'autres et formalise la notion de structure de données.

Quelques références afin de compléter votre apprentissage :

- Cours à l'INSA en première année : lien.
- Cours à l'INSA en 2MIC : lien du semestre 1 et lien du semestre 2. En 2IMACS, le programme est le même que le semestre 1 de la 2MIC.
- Cours à l'INSA de Python : lien
- *Introduction à l'algorithmique*, Cormen, Leiserson, Rivest, Stein. La référence incontournable de l'algorithmique, un livre très complet, sûrement disponible à la bibliothèque.

Première partie

Notions fondamentales

Chapitre 1

Variables et types

L'informatique est la science du traitement automatique d'informations. Afin de représenter ces informations, on introduit la notion de variable.

1.1 Définition

On se donne quelques types de base :
— des valeurs numériques (entiers naturels, relatifs, réels, etc...),
— des caractères,
— des booléens (vrai ou faux).
On peut ensuite définir des variables d'un type donné. Cela revient à créer une case pouvant contenir un élément du type donné.

1.2 Implémentation

Selon le langage de programmation, ces types sont représentés de différentes façons. Un type est :
— un nombre d'octets : c'est l'espace utilisé,
— une représentation en mémoire : c'est la façon d'interpréter les 0 et les 1 qui sont en mémoire.

Langage C. On a (entre autres) les types suivants :

int : un entier relatif, généralement sur 4 octets (de -2^{31} à $2^{31}-1$). Le premier bit représente le signe (0 pour +, 1 pour -, le reste pour la valeur).

long : un entier relatif généralement sur 8 octets (de -2^{63} à $2^{63}-1$).

unsigned int / unsigned long : un entier naturel. Plus besoin d'utiliser le premier bit pour le signe, on peut l'utiliser pour représenter deux fois plus de valeurs (unsigned int de 0 à $2^{32}-1$).

float : un nombre flottant sur 4 octets. Grossièrement, la représentation est constitué d'un bit de signe, de plusieurs bit d'ordre de grandeur (où placer la virgule), et du reste pour la valeur. Cette représentation est similaire à l'écriture scientifique, en remplçant 10 par 2.

double : un nombre flottant sur 8 octets. Même principe que les floats mais avec une meilleur précision.

char : un caractère. Il s'agit d'un entier sur un octet, entre -128 et 127.

Remarque. *Pas de booléen :* 0 *représente False, toute autre valeur True. Vous pourrez néanmoins introduire des booléens avec stdbool pour une meilleure lisibilité de votre code.*

Remarque. *Ne pouvant différencier l'infinité des nombres réels, on les stocke avec une certaine précision. Attention aux erreurs d'arrondis ! Certains langages dits* formels *permettent aussi de stocker des valeurs exactes pour tout fraction et racine : aucune simplification n'est opérée s'il y a une perte de précision.*

Langage Python. Les types de bases sont les suivants :

int : un entier relatif de taille variable en mémoire.

float : un nombre flottant de taille fixe. Attention, même problèmes d'arrondis qu'en C : $0.1 + 0.1 + 0.1 \neq 0.3$.

str : une chaîne de caractère (string). Pas de type spécifique pour un seul caractère.

Chapitre 2

Instructions

Afin de manipuler les variables, on va opérer des ensembles d'instructions qui seront exécutées séquentiellement. On qualifie cette approche algorithmique d'*impérative*.

2.1 Définition

Forme d'une instruction. On peut lire et écrire dans une variable :
- écriture : $X \leftarrow 1$
- lecture : `Afficher`(X)
- les deux : $Y \leftarrow X$

Affectations. La première et la troisième forme sont ce qu'on appelle des affectations. La première affecte X à 1. Dans une affectation, tout ce qui est à droite de la flèche est évalué, puis placé dans ce qui est à gauche (il faut que ça soit une variable !). Par exemple, si X contient 1, pour l'instruction $X \leftarrow 3 * (X + 2) + X$, on évalue $3 * (1 + 2) + 1$ qui fait 10 et on écrit 10 dans X. Repensez-y quand vos instructions seront plus complexes.

Opérations. On se donne (au moins) les opérations suivantes : addition, soustraction, multiplication, division flottante, division entière et modulo. La division flottante est la division classique. Ce que l'on appelle division entière (resp. modulo) est le quotient (resp. le reste) de la division euclidienne. Par exemple, la division entière de 9 par 4 est 2 et le modulo est 1 car $9 = 4 * 2 + 1$. Pensez à préciser quel division vous utilisez si il y a ambigüité.

Séquence d'instructions. En algorithmique impérative, un algorithme est une séquence d'instruction qui s'éxecute dans l'ordre.

Algorithme 1 : Séquence d'affectations

1 X : *Entier*
2 Y : *Entier*
3 $X \leftarrow 1$
4 $X \leftarrow X + 1$
5 $Y \leftarrow X$
6 $Y \leftarrow 8 * Y$
7 $X \leftarrow X + 1$
8 $X \leftarrow 10$

Remarque. *Notez qu'on ne peut pas remonter en arrière. Après avoir opéré la dernière instruction, comment déterminer ce qui était précédemment dans X ? Il faut reprendre depuis le début.*

2.2 Implémentation

Affectation. Selon les langages, la flèche vers la gauche est remplacée par un signe différent. En C et en Python, c'est un "=". L'instruction $X \leftarrow X + 1$ s'écrit donc $X = X + 1$. Ce n'est pas une égalité au sens mathématique. En Ada, l'affectation s'écrit " :=".

En *C*, la fin d'une instruction est représentée par un point virgule. Il est très fréquent de l'oublier au début (et même après...). En Python, c'est le saut à la ligne qui représente une fin d'instruction.

Opérations. Pour l'addition, la soustraction et la multiplication, la plupart des langages utilisent les signes $+, -$ et $*$. Pour la division, il faut être vigilant !

En Python, / est la division flottante et renverra toujours un flottant. Par exemple, $7/2$ renvoie 3.5 et $6/2$ renvoie 2.0. Pour la division entière, on peut utiliser // : par exemple, $7//2$ renvoie 3.

En *C*, seul le signe / est disponible. Entre deux int, c'est la division entière qui sera faite, mais entre deux floats, c'est la division flottante. Evitez de faire des calculs entre des variables de types différents. Des conversions implicites sont faites mais êtes vous sûr de savoir lesquelles ?

Séquence. L'algorithme 1 peut s'implémenter en C :

```
int x;
int y;
x = 1;
x = x+1;
y = x;
y = 8*y;
x = x+1;
x = 10;
```

En Python, pas besoin de déclarer les variables x et y.

2.3 Exercices

Exercice 1. Ecrire une suite d'instruction qui permute le contenu de deux variables réelles. Comment éviter d'utiliser une variable temporaire ?

Chapitre 3

Branchement conditionnel

3.1 Définition

Le comportement d'un algorithme peut être variable, selon l'état de la mémoire. On peut tester une condition sur cet état, puis adapter le comportement de l'algorithme selon l'évaluation de la condition.

Algorithme 2 : Branchement conditionnel

```
1 si Condition alors
2 |  Comportement si Condition Vraie
3 sinon
4 |  Comportement si Condition Fausse
```

Condition. L'évaluation d'une condition doit renvoyer un booléen. Ce peut être directement un booléen : on regarde le contenu d'une variable booléenne. On peut aussi définir des formules logiques inductivement avec les opérateurs suivants :

- $=$: *égalité*. $X = 1$ est Vraie si et seulement si la variable X contient la valeur 1.
- **non** : *inversion* d'un booléen. Si on se donne A une variable booléenne, la formule "non A" est vraie si et seulement A contient Faux.
- **ou** : *disjonction*. Le "ou" logique est inclusif. Si on se donne A et B deux variables booléennes, la formule "A ou B" est vraie si et seulement si au moins l'une des deux variables contient Vrai.
- **et** : *conjonction*. Si on se donne A et B deux variables booléennes, la formule "A et B" est vraie si et seulement si les deux variables contiennent Vrai.

Ces opérateurs peuvent être utilisés sur des formules à leur tour.

Exemple. On fait un pile ou face et l'algorithme affiche en texte le résultat. Il est impossible de prédire le comportement, le branchement conditionnel est donc nécessaire.

Algorithme 3 : Pile ou Face

```
1 P : pièce de monnaie
2 P ← Tirage aléatoire pile ou face
3 si P=Pile alors
4 |  Afficher("Pile") ;
5 sinon
6 |  Afficher("Face") ;
```

3.2 Implémentation

Les opérateurs ont des syntaxes variables selon les langages.

Langage	C	Python	Ada	Ocaml
Egalité	==	==	=	=
Différence	!=	!=	/=	<>
Inversion	!	not	not	not
Disjonction	||	or	or	||
Conjonction	&&	and	and	&&

Remarque. *Attention, en C et en Python, l'égalité dans une condition est avec deux ==. Un seul = est pour l'affectation. C'est un choix arbitraire, fait pour avoir des codes plus courts, l'affectation étant beaucoup plus souvent utilisée que le test d'égalité.*

En Ada, on préfère la lisibilité : un seul = pour l'égalité, comme en mathématiques, et := pour l'affectation, un signe dissymétrique pour une opération asymétrique.

Pile ou Face en C.

```c
int P;
P = rand()%2;
if(P==0){
   printf("Pile");
}
else{
  printf("Face");
}
```

Pile ou Face en Python.

```python
P = random.randint(0,1)
if(P==0):
   print("Pile")
else:
  print("Face")
```

3.3 Exercices

Exercice 2. Evaluer les formules logiques suivantes donne t'il toujours Vrai ? Faux ? Ou ça dépend ? Dans ce dernier cas, préciser quand on obtient Vrai.

1. $Vrai\ ou\ Faux$.
2. $X = 1\ et\ (non\ Y)$ avec X qui contient 1 et Y qui contient $Vrai$.
3. $(X\ ou\ (nonY))\ et\ ((Z == X)\ ou\ (X\ et\ Y))$.
4. $((\pi = 3)\, ou\ non(X > Y))\, ou\ non\left(Z > Xet\left(\frac{3}{2} = 1\right)\right)$.

Exercice 3. On suppose qu'une variable *Prix* contient le prix d'une commande faite en ligne. La livraison est gratuite si le prix dépasse les 500€, coûte 20% du prix si la commande est inférieure à 150€ et 10% si elle est entre 150€ et 500€. Proposer une séquence d'instruction qui calcule le coût total (prix + livraison).

Chapitre 4

Boucle

4.1 Définition

Afin de répéter plusieurs fois un même bloc d'instruction, on utilise des boucles.

Boucle Tant que. La boucle *Tant que* est la plus générique. Elle prend la forme suivante :

Algorithme 4 : Boucle Tant que

```
1 Bloc d'instruction 1
2 tant que Condition faire
3 |  Bloc d'instruction 2
4 Bloc d'instruction 3
```

Le bloc d'instruction 1 est fait en premier. Puis on teste la condition de la boucle. Si elle est vraie, on fait le bloc 2. Puis on teste à nouveau la condition. Si elle est toujours vraie, on refait le bloc 2, etc... Dès qu'on teste cette condition et qu'elle est fausse, on ne fait plus le bloc 2, on dit qu'on sort de la boucle. On fait alors le bloc 3.

Remarque. *Une condition de boucle n'est testée qu'avant de faire tout le bloc 2 et pas pendant. De plus, il faut que la condition devienne fausse à un moment, sinon on est bloqué dans une boucle infinie. Il faut donc s'assurer à minima que le bloc 2 soit parfois amené à modifier la condition.*

On prend l'exemple suivant pour un compte à rebours :

Algorithme 5 : Compte à rebours v1

```
1 N : Entier naturel
2 N ← 10
3 tant que N > 0 faire
4 |  Afficher(N)
5 |  N ← N − 1
6 Afficher("Boom")
```

Le programme va afficher 10, 9, ..., 1 puis "Boom".

Dans cet exemple, on peut prévoir le nombre d'itération à l'avance. Mais une boucle *Tant que* fonctionne aussi si on ne le sait pas. Par exemple, si la valeur initiale dans N est aléatoire. On en verra un exemple plus loin.

Boucle Pour. Si on connaît le nombre d'étape, on peut utiliser une boucle *Pour*.

Algorithme 6 : Compte à rebours v2

```
1 N : Entier relatif
2 N ← 10
3 for i de 0 à N − 1 do
4 |   Afficher(N)
5 |   N ← N − 1
6 Afficher("Boom")
```

Quel interêt me direz vous ? On peut utiliser le i dans la boucle ce qui évite d'avoir à modifier N.

Algorithme 7 : Compte à rebours v3

```
1 N : Entier relatif
2 N ← 10
3 for i de 0 à N − 1 do
4 |   Afficher(N − i)
5 Afficher("Boom")
```

Remarque. *Une boucle* Pour *peut être transformée en boucle* Tant que. *La réciproque est fausse. Lorsque vous apprenez un nouveau langage, il est donc prioritaire d'apprendre à écrire une boucle* Tant que.

4.2 Implémentation

Boucle Tant que

Exemple en C.

```c
int N = 10; //On peut declarer et affecter sur un meme ligne
while(N>0){
  printf("%d",N); // %d pour un entier, N le remplacera
  N=N-1; //N-=1 et N-- font la meme chose
}
printf("Boom");
```

Exemple en Python.

```python
N=10
while(N>0):
    print(N)
    N=N-1
}
print("Boom")
```

Boucle Pour

Exemple en C.

```
int N = 10;
int i;
for(i=0;i<N;i=i+1){
  printf("%d",N-i);
}
printf("Boom");
```

Le for utilise trois informations : l'initialisation de l'indice, la condition pour continuer la boucle et l'itération (comment passer d'une étape à la suivante). On peut définir l'indice directement dans la boucle : `for(int i=n;i>0;i--)`. Cet exemple illustre aussi comment parcourir les indices à l'envers.

Exemple en Python.

```
N=10
for i in range(N):
    print(N-i)
print("Boom")
```

En Python, *range*(N) correspond à l'intervalle $[\![0, N-1]\!]$. On peut aussi utiliser `range(a,b)` pour $[\![a, b-1]\!]$ et même ajouter un troisième paramètre pour régler le pas : `range(a,b,p)` pour l'ensemble $\{a + p.k | k \in \mathbb{N}, a + pk < b\}$.

4.3 Exercices

Exercice 4. Ecrire un algorithme qui prend en entrée $a \in \mathbb{Z}$ et $n \in \mathbb{N}$ et qui renvoie a^n. Pour améliorer une première version naïve, on pourra chercher des ressources sur l'exponentiation rapide, méthode très classique.

Exercice 5. Ecrire un algorithme qui prend en entrée $n \in \mathbb{N}$ et qui renvoie $n!$. Quel problème risque t'on de rencontrer en l'implémentant ?

Exercice 6. Pour un paramètre C entier strictement positif, on définit la suite de Syracuse par :

- $u_0 = C$
- $u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$

On suppose que pour tout $C \geq 1$, il existe $n \geq 0$ tel que $u_n = 1$. Soit S_C le premier rang n tel que $u_n = 1$. A partir de ce rang, on remarque que les termes de la suite $(u_n)_{ngeqS_C}$ sont périodiques en parcourant en boucle les valeurs $4, 2, 1$. C'est la conjecture de Syracuse.

1. Ecrire un algorithme qui calcule S_C.
2. Reprendre votre algorithme précédent afin de maintenant calculer la hauteur en vol, c'est à dire la valeur maximale prise par la suite.
3. Ecrire un algorithme qui prend en entrée $N \in \mathbb{N}^*$ et qui affiche les valeurs de S_C pour $C \in [\![1; N]\!]$.

Exercice 7. Ecrire un algorithme qui prend en entrée $n \in \mathbb{N}$ et qui renvoie la somme des chiffres apparaissant dans la décomposition de n en base 10. Montrer la correction de votre algorithme.

Chapitre 5

Sous algorithme

5.1 Définition

Quand un problème est complexe, il est déraisonnable de vouloir le résoudre avec un seul algorithme, en un seul morceau. On décompose une solution en sous algorithme, chacun résolvant une partie du problème. Les avantages sont multiples : organisation, clareté, dynamisme (si on veut changer un petit morceau), réutilisation d'un sous algorithme plusieurs fois, etc...

Spécification. Afin d'utiliser un sous algorithme, il faut être capable de le spécifier rigourement. Il faut nécessairement :

— un nom,
— la description de l'entrée (types et noms des variables que l'on donne à l'algorithme),
— la description de la sortie (au moins le type, voire le nom de ce que renvoie l'algorithme).

Parfois, on rajoute d'autres informations : conditions sur les paramètres, description, etc...

Utilisation d'un sous algorithme. Un algorithme peut en appeler un autre :

Algorithme 8 : Algorithme Principal

```
   Entrées : X, Y, Z : Valeurs numériques
   Sorties : X
1  tant que Z < X faire
2      si X < Y alors
3          X ← Algorithme1(Y, Z)
4      sinon
5          Y ← Algorithme2(X, Y)
6  renvoyer X
```

Il faut donc que les deux sous-algorithmiques prennent deux valeurs numériques en entrée et en renvoient une.

5.2 Implémentation

En C.

```c
int algoPrincipal(int x, int y, int z){
  while(z<x){
    if(x<y){
      x = algo1(y,z);
    }
    else{
      y = algo2(x,y);
    }
  }
  return x;
}
```

En Python.

```python
def algoPrincipal(x,y,z):
  while(z<x):
    if(x<y):
      x = algo1(y,z)
    else:
      y = algo2(x,y)
  return x;
```

5.3 Exercices

Exercice 8. Ecrire un algorithme qui calcule le maximum entre deux valeurs. Puis entre trois valeurs. Enfin entre quatre.

Exercice 9. Ecrire un algorithme qui détermine si $|x-y| < z$ ou [$(x+z)^2 \geq y$ et $|z|^5 - 4*(x-4)^4$] en prenant x, y et z trois réels en paramètre. On ne dispose pas de l'opération valeur absolue, ni puissance.

Deuxième partie

Notions avancées

Chapitre 6

Approche ascendante et descendante

6.1 Définition

Généralement, une bonne solution à un problème complexe est constituée d'un algorithme principal qui en appelle plusieurs intermédiaires, qui eux même en appellent des plus petits, etc... Souvent, l'approche la plus naturelle est de se concentrer sur les détails, de préciser les petits algorithmes, puis de les assembler en remontant jusqu'à l'algorithme principal. Il s'agit de l'approche ascendante.

Cependant, cette méthode est souvent inefficace : on perd du temps à réfléchir aux détails, à spécifier un petit algorithme technique, puis de se rendre compte à l'assemblage qu'il est inutile ! Il est donc fortement conseillé de faire l'inverse. On rédige d'abord l'algorithme principal en précisant qu'on appellera certains sous algorithmes en précisant seulement leurs spécifications mais pas leurs contenus. Dit autrement, on dit ce que chaque sous algorithme doit calculer mais on ne dit pas comment. Ensuite, on s'intéresse au contenu de ces algo, etc... jusqu'à descendre aux algorithmes de bases (qui n'en appellent pas d'autre). C'est l'approche descendante.

Pour illustrer avec un exemple, on corrige la première question de l'exercice 6.

Algorithme 9 : `Syracuse`

Entrées : C : Entier naturel
Sorties : n premier rang tel que $u_n = 1$ avec $(u_n)_{n\in\mathbb{N}}$ suite de Syracuse de paramètre C.

```
1 u, n : Entiers naturels
2 u ← C
3 n ← 0
4 tant que u ≠ 1 faire
5 |  u ← TermeSuivant(u)
6 |  n = n + 1
7 renvoyer n
```

Algorithme 10 : `TermeSuivant`

Entrées : $courant, param$: Entiers naturels
Sorties : *suivant* le terme suivant *courant* dans la suite de Syracuse.

```
1 suivant : Entier
2 si EstPair(courant) alors
3 |  suivant ← courant/2
4 sinon
5 |  suivant ← 3 ∗ courant + 1
6 renvoyer suivant
```

Algorithme 11 : `EstPair`

Entrées : k : Entier naturel
Sorties : $estPair$ booléen Vrai si et seulement si k est pair.

```
1 estPair : Booléen
2 si k mod 2 = 0 alors
3 |  estPair ← Vrai
4 sinon
5 |  estPair ← Faux
6 renvoyer estPair
```

Remarque. *En implémentant, on utilisera plutôt l'approche ascendante afin de pouvoir compiler progressivement. De même pour tester le code. En effet, si on observe une erreur en testant la fonction principale, bon courage pour trouver l'erreur : elle peut être n'importe où. Si vous tester d'abord les fonctions de base, puis leurs surfonctions, etc... l'enquête sera bien plus simple.*

6.2 Exercices

Exercice 10. Soit φ l'indicatrice d'Euler, i.e.

$$\forall n \in \mathbb{N}^*, \varphi(n) = Card\{k \in \mathbb{N}^*, k \leq n, k \wedge n = 1\}|.$$

avec $k \wedge n$ le pgcd (plus grand commun diviseur) de k et n.

1. Proposer un découpage en sous problèmes pour calculer φ.
2. Détailler le contenu de chaque algorithme.

Exercice 11. Ecrire un algorithme qui prend en entrée $n \in \mathbb{N}^*$ et affiche la moyenne de tous les entiers palindromes entre 1 et n.

Chapitre 7

Récursivité

7.1 Définition

Un algorithme peut s'appeler lui même ! Mais il risque à nouveau de s'appeller, à l'infini. Informellement, il faut deux choses :

- Des cas de base (où l'algorithme ne s'appelle pas lui-même).
- Des appels récursifs dans les autres cas, mais avec des paramètres qui "se rapprochent des cas de base".

Remarque. *Formellement, vous devez considérer des paramètres dans un ensemble muni d'un ordre bien fondé (pas de suite strictement décroissante). Par exemple, on peut considérer les entiers naturels muni de l'ordre naturel. Les cas de base sont (au moins) les éléments minimaux. Un appel est soit fait sur un cas de base, soit il ne fait des appels récursifs que sur des paramètres strictement plus petits.*

Exemple. On prend l'exemple classique du calcul de la factorielle : $\forall n \in \mathbb{N}, n! = \prod_{k=1}^{n} k$.

Algorithme 12 : Algorithme `Factorielle`

Entrées : n : Entier naturel
Sorties : $n!$: Entier naturel

```
1 si n > 1 alors
2     renvoyer n * Factorielle(n − 1)
3 sinon
4     renvoyer 1
```

Ici, on a deux cas de base : 0 et 1, pour lesquels la factorielle vaut 1. Pour tout n qui n'est pas un cas de base, on a la formule récursive suivante : $n! = n * (n-1)!$. L'appel récursif est donc fait sur un paramètre plus petit. Le déroulement sera le suivant : `Fact(n)` appelle `Fact(n-1)` qui appelle `Fact(n-2)`, etc... jusqu'à tomber sur `Fact(1)`. Puis on remonte en faisant les multiplications.

Remarque. *On a écrit des* `Renvoyer` *au milieu de l'algorithme et non à la fin. Trois points :*

- *Une fois un* `Renvoyer` *effectué, on sort de l'algorithme et on ignore la suite. Donc toute instruction qui suit le* `Renvoyer`*, y compris dans le if, sera ignorée.*

— *Dans un programme complexe, oublier de mettre un* `Renvoyer` *pour un cas ou se tromper de cas devient fréquent. Vérifier la correction de l'algorithme devient laborieuse, tout comme le débug de son implémentation. En C, le compilateur vous avertira si il manque un* `Renvoyer`*. Mais Python ne vous dira rien...*
— *Cette écriture est parfois pratique. Sur cet exemple simple, elle permet une syntaxe plus concise. On peut aussi éliminer les cas simples au début d'un programme, puis traiter les cas complexes.*

7.2 Implémentation

Aucune difficulté dans les langages C et Python : une fonction peut s'appeler elle même. Dans d'autres langages, il faut une syntaxe particulière. Par exemple, il faut rajouter le mot clé *rec* en Ocaml.

Soyez quand même vigilants au dépassement de pile (stack overflow), si il y a trop d'appels récursifs. En C, la limite par défaut est généralement haute mais ce n'est souvent pas le cas en Python.

Remarque. *Une technique pour éviter un dépassement de pile consiste à utiliser de la récursivité terminale : il n'y a qu'un seul appel récursif et c'est la dernière opération à effectuer. Ce n'est pas le cas de l'algorithme 12 (factorielle), où la multiplication est la dernière opération et non l'appel récursif. Ce mécanisme n'est pas proposé dans tous les langages : par défaut en Ocaml, nécessite une option en C, impossible en Python et en Java.*

7.3 Exercices

Exercice 12. On veut inverser les chiffres d'un entier écrit en décimal : si on se donne 8934, on doit renvoyer 4398.

1. Ecrire une formule de récurrence et en déduire un algorithme (vous avez le droit d'utiliser la fonction $\log_{10}$).
2. Se passer de l'utilisation du logarithme si ce n'est déjà fait. On veut une complexité linéaire en le nombre de chiffre de l'entier donné. Si vous ne savez pas ce que veut dire linéaire, lisez le chapitre 9 puis revenez sur cette question.

Exercice 13. Coefficients binomiaux. On note $\binom{n}{p}$ ou C_n^p le nombre de parties à p éléments dans un ensemble à n éléments. L'inversion n/p entre les deux notations n'est pas une typo.

1. On peut montrer que $\binom{n}{p} = \frac{n!}{!(n-p)!}$. Quel problème peut-on avoir en pratique si on utilise cette formule pour calculer des coefficients binomiaux ? $\binom{100}{2} = 4950$ mais calculer cette petite valeur peut poser des problèmes.
2. Ecrire une formule de récurrence pour $\binom{n}{p}$.
3. Proposer un algorithme basé sur cette formule.
4. Dessiner les axes N et P et illustrer le calcul récursif, en montrant de quel point du plan dépend l'appel au point (n, p). Quel défaut, flagrant sur le dessin, a cette méthode ?
5. Proposer une autre méthode de calcul plus efficace.

Chapitre 8

Tableaux

8.1 Définition

Un tableau est une séquence de cases contenant chacune un élément. Généralement, ces éléments sont du même type (tableau d'entiers, tableau de caractères, chaîne de caractères, ...). Chaque case a un indice unique. Si le tableau contient n cases, les indices de ses cases vont de 0 à $n-1$. On dit que les cases sont contigües, les indices se suivent : case 0, puis case 1, etc... Parfois, on trouve plus naturel que les indices aillent de 1 à n. Ou n'importe quel intervalle discret à n éléments. Il faut juste que les cases soient contigües (indices qui se suivent).

On peut accéder à n'importe quelle case directement pour la lire ou y écrire. Ici, on notera $T[i]$ le contenu de la case d'indice i dans le tableau T. Par contre, on ne peut en général pas augmenter sa taille : il faut alors définir un nouveau tableau et y copier l'ancien.

8.2 Exemples de parcours de tableau.

On propose deux exemples : la recherche de minimum et la recherche d'un élément donné.

Premier exemple : minimum d'un tableau. On se donne un tableau d'entiers et on veut en connaître le plus petit élément.

Algorithme 13 : Minimum d'un tableau (version avec Tant que)

Entrées : T : tableau d'entiers de taille n // par défaut, indicé de 0 à $n-1$
Sorties : min le minimum du tableau

```
1 min ← +∞
2 i ← 0
3 tant que i < n faire
4 |   si T[i] < min alors
5 |   |   min ← T[i]
6 |   i ← i + 1
7 renvoyer min
```

On peut aussi utiliser une boucle Pour puisqu'il faudra nécessairement parcourir tout le tableau, et on connaît d'entrée de jeu le nombre de case à consulter.

Algorithme 14 : Minimum d'un tableau (version avec Pour)

Entrées : T : tableau d'entiers de taille n // par défaut, indicé de 0 à $n-1$
Sorties : min le minimum du tableau

```
1 min ← +∞
2 for i de 0 à n − 1 do
3 |   si T[i] < min alors
4 |   |   min ← T[i]
5 renvoyer min
```

Deuxième exemple : recherche d'un élément. Ici, on se donne une chaîne de caractère T et un caractère c et l'on veut savoir si c apparaît dans T. Puisque la boucle Pour semblait plus simple pour l'exemple précédent, utilisons la.

Algorithme 15 : Recherche d'un élément (version avec Pour)

Entrées : T : tableau de caractères de taille n, c un caractère
Sorties : vu booléen indiquant si c est présent

```
1 vu : Booléen
2 vu ← Faux
3 for i de 0 à n − 1 do
4 |   si T[i] = c alors
5 |   |   vu ← Vrai
6 renvoyer vu
```

Pour se convaincre du bon fonctionnement de cet algorithme, on peut remarquer qu'après la i-ème étape, le booléen indique si l'élément est dans les i premiers éléments. Formellement, vous pouvez le montrer par récurrence.

Remarque. *Attention à ne pas remettre le booléen à Faux si l'élément courant n'est pas le bon. Sinon, l'algorithme ne renverrait comme information que le fait que le dernier élément de la chaîne soit le caractère recherché ou non.*

Une fois l'élément trouvé, il est inutile de continuer la recherche. Ce serait particulièrement dommage si on le trouve dès le début d'un très grand tableau.

La boucle Pour ne permet pas de s'arrêter à mi-chemin, ou alors avec un `Renvoyer` un peu sale au milieu, à éviter dans un premier temps. Si vous connaissez l'instruction `break`, attention : elle n'est pas disponible dans tous les langages. Et ce n'est pas toujours une bonne pratique, sur des codes de tailles importantes, avec des boucles imbriquées en particulier.

On va donc utiliser une boucle Tant que.

Algorithme 16 : Recherche d'un élément (version avec Tant que)

Entrées : T : tableau de caractères de taille n, c un caractère
Sorties : vu booléen indiquant si c est présent

```
1 vu : Booléen
2 vu ← Faux
3 i ← 0
4 tant que i < n et (non vu) faire
5 |   si T[i] = c alors
6 |   |   vu ← Vrai
7 |   i ← i + 1
8 renvoyer vu
```

Cette fois-ci, dès que l'on voit l'élément recherché, vu passe à Vrai et la boucle s'arrête : c'est plus efficace.

8.3 Implémentation

En algorithmique, déclarer un tableau ne pose aucun problème. Mais lors de l'implémentation, la tâche devient plus ou moins complexe selon le langage de programmation utilisé. Commençons par le plus simple.

8.3.1 En Python

En Python, manipuler un tableau est assez simple. Le type correspondant est appelé *list* (à ne pas confondre avec les listes chaînées) :

```python
def minArray(T):
  minVu = math.inf #represente + infini en Python
  for i in range(len(T)): #len(T) renvoie la taille de T, len pour length
    if T[i] < minVu :
      minVu = T[i]
  return minVu

T = [4,2,9]
T.append(4)
minVu = minArray(T)
print(minVu)
#min(T) donne deja le mininum de T, la fonction min existant deja
```

En Python, le programme principal est ce qui est en dehors toute définition de fonction. Les fonctions (comme minArray) sont simplement définies, pas exécutées tant qu'elles ne sont pas appelées par le programme principal ou par une autre fonction elle même appelée. Pour tester ses fonctions, il est donc nécessaire de les appeler : exécuter des définitions ne teste rien, à part les grosses fautes de syntaxe.

Remarque. *Si vous utilisez un langage qui n'a pas de ∞, vous serez tenté de mettre une grande valeur. Mais laquelle ? 1000 ? 10000 ? On n'est jamais sûr que ça va suffire. On pourrait mettre le plus grand entier possible mais il y a une solution moins bancale et plus simple. Il suffit d'initialiser min avec n'importe quelle valeur du tableau : elle sera forcément supérieure ou égale au minimum. On peut par exemple initialiser avec la première case et commencer la boucle par la deuxième.*

La solution précédente parcourt les indices du tableau mais on peut directement parcourir les éléments.

```
def minArray(T):
  minVu = math.inf
  for x in T:
    if x < minVu :
      minVu = x
  return minVu
```

Remarque. *En Python, on peut rajouter des éléments à un tableau, avec la méthode* append. *Exemple : T.append(4) rajoute 4 à la fin du tableau. Le type* list *est ce que l'on appelle un tableau dynamique.*

8.3.2 En C

En C, ça devient plus compliqué. Un tableau contenant des éléments de type `obj` est de type `obj[]`. Par exemple, un tableau d'entiers est de type `int[]`. On ne peut pas les agrandir, il faut donner la taille dès la déclaration.

```
int searchArray(int tab[], int tailleTab, int x){
  int vu = 0;
  int i = 0;
  while(i<tailleTab && vu==0){
    if(T[i]==x) vu = 1; //instruction sur la meme ligne que le if
    else i++; //pas besoin d'accolade si une seule instruction
  }
  return vu;
}

int main(){
  int T[4] = [5,2,9,4];
  int vu = searchArray(T,4,2);
  printf("%d",vu);
  return 0;
}
```

Remarque. *En C, le programme principal est une fonction toujours appelée* main. *Tout programme doit contenir une unique fonction main. C'est logique : sans cela, qu'exécuterait la machine pour commencer ?*

Remarque. *En C, pour donner un tableau en paramètre d'une fonction, il faut lui donner aussi*

sa taille. En pratique, la seule information qui passe avec tab *est l'adresse de la première case (qui permet de retrouver toutes les autres). C'est pourquoi on doit rajouter* tailleTab *dans les paramètres.*

8.4 Exercices

Exercice 14. Ecrire un algorithme qui inverse l'ordre des éléments d'un tableau.

Exercice 15. Ecrire un algorithme qui insère un élément dans un tableau trié. L'élément maximal sera perdu.

Exercice 16. Somme des éléments d'un tableau.

1. Proposer un algorithme itératif (boucle) qui calcule la somme des éléments d'un tableau d'entier.
2. Proposer une version récursive.

Exercice 17. On se donne un tableau d'entiers.

1. Proposer un algorithme qui cherche si un entier donné appartient au tableau.
2. Pour un tableau à n éléments, combien fait-on de comparaisons ?
3. On suppose maintenant que le tableau est trié. Proposer un algorithme qui tire profit de ce tri.

Exercice 18. Ecrire un algorithme qui prend T un tableau d'entiers en entrée et détermine si il existe un indice séparateur, i.e. si il existe un indice i_s tel que $\sum_{i<i_s} T[i] = \sum_{i>i_s} T[i]$. Essayer de trouver une version linéaire.

Exercice 19. Ecrire un algorithme qui prend en entrée T un tableau d'entiers et s un entier. L'algorithme doit déterminer si il existe deux éléments de T tels que leur somme est égal à s. Autrement dit, il existe $i \neq j, T[i] + T[j] = s$. Si le tableau est trié, améliorer votre algorithme.

Exercice 20. Ecrire un algorithme qui tri un tableau par ordre croissant. Que faire maintenant si on veut trier par nombre de 1 croissants dans l'écriture binaire ? (en cas d'égalité, ordre naturel)

Exercice 21. Ecrire un algorithme qui calcule la moyenne des éléments d'un tableau d'entiers naturels en excluant les deux plus grandes valeurs.

Chapitre 9

Complexité

9.1 Définition

La complexité d'un algorithme est une façon d'exprimer ses performances et de les comparer avec celles d'autres algorithmes. Il y a deux grands types de complexités algorithmiques : en *temps* ou en *mémoire*. En général, on s'intéresse plus à la complexité temporelle car c'est le facteur le plus souvant limitant. Attention cependant à des cas comme l'embarqué où la mémoire peut être un facteur critique.

Modèle RAM. Plusieurs modèles de calculs existent : modèle RAM, machine de Turing, fonctions calculables, lambda calcul, ... On va s'intéresser au plus courant, souvent choisi pour sa simplicité et sa ressemblance avec une vraie machine. Dans le modèle RAM, la complexité temporelle d'un algorithme est le nombre d'opérations élémentaires que celui-ci fait :

— opérations arithmétiques (addition, soustraction, multiplication, division et modulo),
— affectation et lecture,
— comparaisons.

En général, le nombre d'opérations élémentaires à effectuer dépend des entrées. On va donc l'exprimer en fonction des valeurs en entrée, ou de paramètres sur les entrées si elles sont trop complexes. Par exemple, si une entrée est un tableau, on va généralement exprimer la complexité en fonction de la taille du tableau et non des éléments le composant.

Meilleur/moyen/pire cas. Cependant, le contenu d'un tableau peut influencer fortement la complexité : il peut être plus facile de trier un tableau déjà trié. On peut donc chercher à exprimer les complexités de pire, moyen et meilleur cas. Pour une taille donnée de tableau, on peut exprimer le nombre d'étapes maximum/moyen/minimum. Attention, le cas moyen est souvent beaucoup plus dur que les deux autres : pire et meilleur demandent d'identifier un tableau spécifique, alors que moyen demande une étude sur l'ensemble des tableaux.

Méthode. Pour un branchement conditionnel, on peut exprimer la complexité pire cas en prenant le maximum d'opérations faites par chaque branchement. La complexité pire cas de l'algorithme 17

est le maximum de la complexité de Bloc 1 et de celle de Bloc 2.

Algorithme 17 : Branchement conditionnel

```
1 si Condition alors
2 |  Bloc 1
3 sinon
4 |  Bloc 2
```

Pour une boucle, il faut exprimer la complexité du bloc interne et la sommer sur l'ensemble des itérations. Pour l'algorithme 18, si la complexité de Bloc 1 ne dépend pas de i, il suffit la multiplier par N pour avoir la complexité globale.

Algorithme 18 : Boucle Pour

```
1 for i de 0 à N − 1 do
2 |  Bloc 1
```

Attention, elle peut parfois dépendre de i comme on peut le voir avec l'algorithme 19 : Bloc 1 est lui-même une boucle Pour dont le nombre d'itérations dépend de i.

Algorithme 19 : Boucles Pour imbriquées

```
1 for i de 0 à N − 1 do
2 |  for j de i à N − 1 do
3 |  |  Bloc 2
```

Exemple. On reprend l'algorithme 13 dont on redonne le pseudo-code :

Algorithme 20 : Minimum d'un tableau (version avec Tant que)

```
   Entrées : T : tableau d'entiers de taille n // par défaut, indicé de 0 à n − 1
   Sorties : min le minimum du tableau
1  min ← +∞
2  i ← 0
3  tant que i < n faire
4  |  si T[i] < min alors
5  |  |  min ← T[i]
6  |  i ← i + 1
7  renvoyer min
```

Pour un tableau de taille n en entrée, commençons par étudier le pire cas du bloc interne à la boucle :

— 2 pour le Si : une affectation dedans et une comparaison pour savoir si il faut rentrer dedans : pire cas, on considère qu'on rentre toujours dedans (ce cas est-il réaliste ?),

— 2 opérations pour la ligne 6 : incrément et affectation de i,

— on peut considérer qu'aller à la i-ème case d'un tableau pour lire/écrire est une opération.

La boucle contient donc 6 opérations.

Globalement :

— 2 affectations au début,

— n itérations donc $6n$ opérations internes à la boucle,

— n itérations donc $n+1$ comparaisons pour la condition de boucle (ne pas oublier le dernier test qui permet de sortir de la boucle).

Donc en tout, l'algorithme 13 fait au plus $7n+3$ opérations élémentaires pour un tableau de taille n en entrée.

9.2 Ordres de grandeur

La complexité exacte peut être trop complexe à exprimer et variable : certains ne compteront que les comparaisons par exemple. On va donc exprimer des ordres de grandeurs à la place. On donne les notations suivantes :

— $f(n) = \mathcal{O}(g(n))$ si $\exists C > 0, \exists N \in \mathbb{N}, \forall n > N, f(n) \leq C \cdot g(n)$,
— $f(n) = \Omega(g(n))$ si $\exists C > 0, \exists N \in \mathbb{N}, \forall n > N, f(n) \geq C \cdot g(n)$,
— $f(n) = \Theta(g(n))$ si $f(n) = \mathcal{O}(g(n))$ et $f(n) = \Omega(g(n))$.

Cela permet d'exprimer des complexité asymptotiques : ce n'est pertinent que pour n grand.

On obtient plusieurs catégories classiques de complexités :

— si la complexité est en $\mathcal{O}(n)$, on dit qu'elle est linéaire,
— si la complexité est en $\mathcal{O}(n^2)$, on dit qu'elle est quadratique,
— si la complexité est en $\mathcal{O}(a^n)$ avec $a > 1$, on dit qu'elle est exponentielle,
— si la complexité est en $\mathcal{O}(\log n)$, on dit qu'elle est logarithmique.

Remarque. *La complexité exacte n'est cependant pas inutile : la constante a une importance secondaire face à l'ordre de grandeur mais n'est néanmoins pas négligeable en pratique. On préférera un algorithme en n^2 qu'un algorithme en $10^{1000}n$. De plus, son étude est parfois un passage obligé pour justifier une complexité en $\mathcal{O}$.*

9.3 Exercices

Exercice 22. On se donne n pièces de monnaie. On sait que l'une d'entre elles (et une seule) est fausse. La seule différence facilement mesurable est qu'elle ne fait pas le même poids. On dispose d'une balance à plateaux, permettant de comparer le poids de deux pièces.

1. Proposer un algorithme permettant de trouver la pièce fausse. Minimiser le nombre de comparaisons. Objectif : $\lfloor (n+1)/2 \rfloor$.
2. Montrer l'optimalité de votre algorithme.
3. Vous savez maintenant que la fausse pièce est plus légère que les autres. Cela change t'il la complexité optimale ?
4. On suppose maintenant qu'on peut mettre un nombre arbitrairement grand de pièces sur chaque plateau. Pouvez vous réduire le nombre de comparaisons ?

Exercice 23. Proposer un algorithme permettant de calculer le minimum et le maximum d'un tableau d'entiers. Minimiser le nombre de comparaisons. Objectif : $\lceil 3n/2 \rceil - 2$.

Chapitre 10

Passage par valeur ou par référence

10.1 Définition

Il y a deux façon de donner des paramètres à un algorithme :

Par valeur : le contenu de la variable est copié. La copie est donnée à l'algorithme qui peut la lire et la modifier. La variable originale n'est pas modifiée.

Par référence : on donne à l'algorithme une référence de la variable. Une référence est un moyen d'accéder à la variable. L'algorithme peut lire et modifier la variable originale à partir de la référence.

Le passage par copie est plus sûre : dans un algorithme, seules les instructions de ce dernier peuvent modifier les variables locales. Les appels à des sous algorithmes ne modifient rien. On maîtrise mieux ce qui se passe, on débug plus facilement.

Le passage par référence peut simplifier certains cas avec des variables de base, mais il devient primordial pour des variables complexes. Par exemple, si un tableau est donné en paramètre d'un algorithme, le copier peut être très coûteux. On préfèrera souvent le faire passer par référence. Les modifications dans le sous algorithme se répercutent dans l'algorithme appelant : on appelle cela un effet de bord (prudence !).

On prend l'exemple classique de l'échange du contenu de deux variables (*swap*).

Algorithme 21 : SwapByReference

Entrées : x, y référence sur deux variables entières.

1 $tmp : Entier$
2 $tmp \leftarrow x$
3 $x \leftarrow y$
4 $y \leftarrow tmp$

Algorithme 22 : Algorithme principal

1 $x, y : Entiers$
2 $x \leftarrow 1$
3 $y \leftarrow 2$
4 `SwapByReference`(x, y)

10.2 Implémentation

En Python. En Python, une vision simplifiée consiste à dire que :

- Les variables de types de bases (entiers, flottants,booléen,...) et de certains types complexes (tuple, string) passent par valeur. Les modifier dans une sous fonction ne les modifie pas dans la fonction appelante.
- Certains types complexes (list, dict) passent par référence. Ils ne sont pas copiés. Donc les modifier dans une sous fonction les modifie dans la fonction appelante.

Remarque. *En vrai, tout passe par référence. La copie d'une variable de base n'est faite que dans la fonction, si on veut la modifier. C'est pratique pour les fonctions qui ne font que lire le contenu pour des types comme les strings et les tuples. On évite de copier une très longue chaîne de caractères si il n'est pas nécessaire de la modifier.*

Reprenons l'exemple précédent :

```
def swap(x,y):
  tmp = x
  x = y
  y = tmp

  a=1
  b=2
  swap(a,b) #a et b ne seront pas modifies
```

Un autre exemple avec un passage par référence :

```
def inc(T):
  for i in range(len(T)):
  T[i] += 1

monTab = [6,3,8,1]
inc(monTab) #monTab sera modifie
print(T) # on verra [7,4,9,2]
```

En C. La gestion des références en C est plus technique mais beaucoup moins opaque. Les références sont implémentées avec ce que l'on appelle des pointeurs : ce sont des adresses. La référence à une variable est simplement son adresse en mémoire. Trois points de syntaxe :

- Pour une variable de type *int*, son adresse est de type *int** (on parle de pointeur). De même pour les autres types.
- Pour récupérer l'adresse d'une variable x, il suffit d'écrire $\&x$. On parle de référencer.
- Si ptr_x est d'adresse d'une variable x, on peut récupérer le contenu de x avec $*x$. On parle de déréférencer.

Remarque. *Attention à ne pas confondre les deux utilisations de l'étoile. La première est pour déclarer un pointeur, la deuxième est pour déréférencer et donne donc une variable.*

On reprend l'exemple de l'échange :

```c
void swap(int * x, int * y){
  int tmp = *x;
  *x = *y;
  *y = *x;
}

int main(){
  int a = 1, b = 2;
  swap(&a,&b); // cette fois les contenus de a et b sont echanges
  return 0;
}
```

Puis l'exemple du tableau :

```c
void inc(int * T, int size){
  for(int i=0;i<size;i++) T[i]++;
}

int main(){
  int tab[4] = {6,3,8,1};
  inc(tab,4);
  return 0;
}
```

Troisième partie

Structures de données

Chapitre 11

Types composés

11.1 Définition

On peut vouloir agglomérer des types de base ensemble pour former un nouveau type. Par exemple, en mettant deux entiers ensemble, on forme un couple d'entiers. En algorithmique, pas besoin de le préciser si le type aggloméré est standard : tout le monde sait ce qu'est un couple d'entiers.

Si vous voulez utiliser un type personnel, il est en général plus pratique de le définir au préalable même si il s'agit simplement d'un produit cartésien. Par exemple, si on veut représenter un élève par un nom, un prénom, un numéro d'étudiant et un tableau de notes, on peut écrire $student = (name, id, grades)$ avec $name$ une chaîne de caractères, id un entier naturel et $grades$ un tableau de réels.

11.2 Implémentation

En Python, le type *tuple* permet de manipuler des types composés mais en introduire proprement nécessite des notions d'orienté objet. Vous le verrez avec des langages beaucoup plus propres : C++ ou Java selon votre parcours.

Pour le C, c'est très simple. L'exemple précédent s'implémente de la façon suivante :

```
struct student{
  char[100] name;
  unsigned int id;
  int * grades;
  unsigned int nbGrades;
}; //attention au point virgule ici !!!

  int main(){
    struct student myStudent; // declaration d'une variable de type struct student
    myStudent.id = 42;
    struct student * ptr_myStudent = &myStudent; //on recupere son adresse
    *(ptr_myStudent).id = 42; //equivalent a l'affectation deux lignes au dessus
    ptr_myStudent->id = 42; //syntaxe purement equivalente a la ligne precedente
```

```
    return 0;
}
```

Le mot clé `struct` permet d'introduire un type composé. Ce nouveau type sera appelé une `struct student`.

Remarque. *Si on en a marre d'écrire* `struct` *partout, on peut rajouter la ligne de code suivante :*

```
typedef struct student student;
```

Il suffira ensuite d'écrire `student myStudent;` *pour déclarer une variable du nouveau type. La syntaxe* `typedef A B;` *remplace les occurences de B par A. Ici A vaut* `struct student` *et B vaut* `student`*.*

On a rajouté une variable pour la taille du tableau de notes (obligatoire sinon on ne peut pas la retrouver). En revanche, on ne l'a pas fait pour les noms et surnoms. Pourquoi ? En C, les chaînes de caractères sont des tableaux de char qui conventionnellement finissent pas un caractère spécial : '\0'. Si il est présent (à vous de faire attention que l'écriture dans ces variables le place), le parcours de *surname* s'arrête dès qu'on le croise.

Remarque. *On donne deux façons de représenter les tableaux :* `char[100] surname` *pour un tableau de taille fixée (il est stocké dans la variable de type struct student). Ou* `int * grades` *: ce n'est pas un tableau mais l'adresse de la première case.*

Chapitre 12

Types abstraits

12.1 Opérations élémentaires

Un type abstrait est :
— Un ensemble d'éléments représenté par ce type.
— Un ensemble d'opérations élémentaires (autrement appelées primitives). Ces opérations servent à manipuler une variable de ce type.

On peut penser un type abstrait comme une boîte noire : elle contient des objets mais on ne peut pas aller voir à l'intérieur comment sont agencés les rouages. Des boutons à sa surface (opérations élémentaires) permettent à l'utilisateur de manipuler la boîte.

12.1.1 Premier exemple : le type abstrait Rationnel.

L'ensemble représenté par ce type est $\mathbb{Q}$. Un ensemble d'opérations élémentaires intéressantes peut être :

CreerRationnel(num,denum) : créer un rationnel de numérateur *num* et de dénominateur *denum* sous forme irréductible,

Numérateur(r) : renvoie le numérateur du rationnel r,

Dénumérateur(r) : renvoie le dénumérateur du rationnel r,

Addition(r1,r2) : renvoie le rationnel $r = r1 + r2$ sous forme irréductible,

Multiplication(r1,r2) : renvoie le rationnel $r = r1.r2$ sous forme irréductible.

Vous penserez sûrement à d'autres opérations qu'il serait utile d'avoir. Il n'y a pas une seule façon de définir un type abstrait. Tout dépend des besoins.

12.1.2 Deuxième exemple : le type abstrait Tableau.

Les éléments pouvant être stockés peuvent eux-mêmes être des ensembles. Par exemple, un tableau d'entiers contient un ensemble fini d'entiers. Le type abstrait *Tableau* peut être muni des opérations suivantes :

CreerTableau(n) : créer un tableau de taille n d'indices de 0 à $n-1$ et le renvoie,

Taille(T) : renvoie la taille du tableau T,

RecupererValeur(T,i) : renvoie le contenu de la i-ème case du tableau T,

ModifierValeur(T,i,x) : écrit x dans la i-ème case du tableau T.

S'ensuivent deux grandes problématiques : implémenter le type abstrait dans un langage de programmation et définir des algorithmes n'utilisant que les opérations élémentaires, sans reposer sur l'implémentation.

12.2 Implémentation

Pour reprendre l'image de la boîte, ici, on met en place les rouages internes qui s'activent en appuyant sur les boutons externes.

Pour implémenter un type abstrait, il faut :

— Choisir comment stocker les informations. Par exemple, si on veut une collection d'entiers, on peut utiliser un tableau, une liste chaînée, un arbre binaire, une table de hachage, ... La disponibilité de ces types dépend du langage, il faut parfois les définir eux-mêmes.

— Implémenter les opérations élémentaires (fonctions).

— *Tester les opérations élémentaires.* On n'insistera jamais assez sur l'importance de cette étape. Pour manipuler notre type à partir de ces opérations (cf Section 12.3), il faut être sûr que nos opérations sont robustes à tous les cas d'utilisation.

Le choix d'une implémentation dépend de deux critères : la simplicité de développement (en particulier dans le langage choisi) et l'efficacité des opérations élémentaires.

Remarque. *Afin de permettre une certaine simplicité d'implémentation, on essaie, en définissant un type abstrait, de réduire le nombre d'opérations élémentaires au minimum. On évite en particulier les opérations facultatives (faisables à partir des autres).*

12.3 Algorithmique

Pour manipuler notre type abstrait dans un algorithme, on n'utilise que les opérations élémentaires. Il est important de ne pas se reposer sur un choix d'implémentation du type. Vos algorithmes seront plus généraux : si quelqu'un veut les coder, il est libre de choisir son implémentation du type préférée.

De plus, la correction de l'algorithme ne dépend pas non plus de l'implémentation ; il suffit que cette dernière soit robuste.

Enfin, en écrivant un algorithme complexe, utiliser les détails d'implémentation est la meilleure façon d'introduire des erreurs. Les langages orientés objets permettent justement d'éviter ce genre d'écueil en l'interdisant. Lors de la définition du type abstrait, il est donc nécessaire de se donner assez d'opérations élémentaires pour pouvoir le manipuler. Ici, on ne touche donc pas aux rouages internes mais seulement aux boutons externes du type abstrait.

12.4 Exercices

Exercice 24. Proposer la définition du type abstrait *complexe*. Dans les grandes lignes, comment implémenter ce type ? Mêmes questions pour le type abstrait *ensemble*.

Chapitre 13

Piles et files

13.1 Piles

13.1.1 Type abstrait

Une pile est un type abstrait qui permet de stocker une collection d'objets du même type et qui est muni des opérations élémentaires suivantes :

CreerPile() : renvoie une pile vide nouvellement créee,

EstVide(P) : renvoie un booléen qui est vrai si et seulement si la pile P est vide,

Empiler(P,x) : ajoute x à la pile P (passage par référence),

Dépiler(P) : retire de P l'élément le plus récemment inséré avec `Empiler` (par référence aussi),

Sommet(P) : renvoie l'élément de la pile le plus récemment inséré.

La pile, comme son nom l'indique, peut être visualisée comme un empilement des éléments. Les éléments sont empilés par ordre d'insertion, avec en bas, le plus anciennement inséré. L'insertion (`Empiler`) et la suppression (`Dépiler`) se font en haut de la pile. On parle d'une structure LIFO : last in, first out.

Remarque. *Souvent, en pratique, l'opération* `Dépiler` *permet aussi de renvoyer l'élément au sommet. La fonction* `Sommet` *devient alors facultative : on pourrait dépiler, puis empiler à nouveau la valeur dépilée. Mais la séquence de ces deux instructions peut se révéler absurdement plus coûteuse en pratique pour certaines implémentations, en cachant de l'allocation dynamique par exemple.*

Le type abstrait est donc défini avec l'opération `Sommet` *la plupart du temps, et toute implémentation sérieuse la contient, facultative ou non.*

13.1.2 Implémentation des piles par chaînage

Une première implémentation classique utilise un chaînage entre les éléments de la pile.

Les types. Dans la représentation chaînée, une séquence d'éléments est une séquence de cellules. Chaque cellule contient un élement, ainsi qu'une référence à la cellule suivante. En C, on va définir

le type `struct cell` pour une cellule. Ce type contient un élément (`int` ici) et un pointeur vers une `struct cell` (la cellule suivante). La dernière cellule pointe vers `NULL`.

Une pile est l'accès à la première cellule, le sommet. On rajoute aussi un champ `size` facultatif.

```
struct cell{ //cellule
  int value;
  struct cell * next;
};

struct stack{ //pile
  struct cell * top; //adresse de la premiere cellule
  int size;
};
```

Les opérations élémentaires. On va passer les piles par valeur : il faut donc modifier la spécification de certaines opérations élémentaires. Lorsque la pile est modifiée, on renvoie la nouvelle. Il faudra donc faire attention à la récupérer dans la fonction mère : `P = depiler(P);` par exemple. Seule la structure de type `struct stack` est passé par valeur, l'ensemble des cellules ne sont pas copiées.

```
struct stack creerPile(){
  struct stack P;
  P.top = NULL; //puisque la pile est vide
  P.size = 0;
  return P;
}

bool estVide(struct stack P){
  return P.top==NULL; //ou P.size==0
}

struct stack sommet(struct stack P){
  if(estVide(P)) error();
  return P->value;
}

struct stack empiler(struct stack P, int x){
  struct cell * newCell = (struct cell *)malloc(sizeof(struct cell));
  if(newCell==NULL) error();
  newCell->value = x;
  newCell->suiv = P.top;
  P.top = newCell;
  return P;
}


struct stack depiler(struct stack P){
  if(estVide(P)) error();
  struct cell * tmp = P.top;
```

```
    P.top = P.top->next;
    free(tmp);
    return P;
  }
```

13.1.3 Implémentation des piles à partir de tableaux

On présente une deuxième technique très similaire au type `list` de Python.

Le type. On commence avec un tableau de taille fixe. On a des cases réservées en mémoire (de quantité *capacity*), et à un instant donné, on n'en utilise qu'une partie pour la pile. Le nombre d'éléments de la pile est *size*. On utilise alors les *size* premières cases du tableau (le bas de la pile étant d'indice 0).

Une version très similaire consiste à remplacer le nombre d'éléments par l'indice du sommet (*top*) de la pile. On a $top = size - 1$, il faudra donc penser à adapter selon le choix que vous faites.

Au passage, on introduit une valeur de pointeur particulière : `NULL`. C'est une adresse illicite. Pensez **toujours** à vérifier qu'un pointeur ne contient pas cette valeur avant de le déréférencer, i.e. avant de faire `*ptr` ou `ptr->champ`. Quand la fonction `malloc` n'arrive pas à allouer la mémoire demandée, elle renvoie `NULL` pour spécifier l'échec. On se donne une fonction `error` pour gérer les mauvaises utilisations des opérations élémentaires.

```
#define TAILLE_INIT 8 //toutes les occurences de TAILLE_INIT seront remplacees par 8

struct stack{
  int * values;
  int size;
  int capacity;
};
```

Les opérations élémentaires. Cette fois, on va passer la pile par référence pour illustrer la différence de fonctionnement. On doit passer en paramètre des `struct stack *`. Plus besoin de renvoyer quoique ce soit pour `empiler` et `depiler`. Pour simplifier le code, on va utiliser un `typedef` : `stack` sera remplacé par `struct stack *` par le compilateur.

```
typedef struct stack * stack; //stack est donc un pointeur sur une structure

stack creerPile(){
  stack P = (stack)malloc(sizeof(struct stack));
  if(P==NULL) error(); //fonction a coder, traitement d'erreur
  P->values = (int*)malloc(sizeof(int)*TAILLE_INIT);
  if(P->values==NULL) error();
  P->size = 0;
  P->capacity = TAILLE_INIT;
  return P;
}
```

```c
int estVide(stack P){
  if(P==NULL) error();
  return P->size==0;
}

int sommet(stack P){
  if(P==NULL || size==0) error();
  return P->values[size-1];
}

void depiler(stack P){
  if(P==NULL || P->size==0) error();
  P->size--;
}

void empiler(stack P, int x){
  if(P==NULL || P->size >= P->capacity) error();
  P->values[size] = x;
  P->size++;
}
```

Si vous voulez rendre vos tableaux dynamiques, pensez rajouter une réallocation de la mémoire quand *size* dépasse *capacity*. Attention, une réallocation afin d'agrandir un tableau consiste à demander au système d'exploitation si les cases qui suivent le tableau sont disponibles. Si ce n'est pas le cas, le tableau est copié dans un espace plus grand. Il faut donc éviter de le faire à chaque étape en demandant de rajouter une seule case au tableau. Au lieu de cela, on peut multiplier par deux la capacité du tableau à chaque réallocation.

La complexité amortie est alors constante. La complexité amortie est la complexité totale d'une séquence d'étapes ramenée sur le nombre d'étapes. Grossièrement, pensez à une facture mensuelle. Vous ne payez pas tous les jours, mais à la fin du mois, vous payez d'un coup une grosse somme pour tous les jours du mois. On reverra cette idée dans la Section 13.2.3.

```c
stack empiler(stack P, int x){
  if(P==NULL) error();
  if(P->size==P->capacity){
    P->capacity *= 2; //on multiplie par 2 la capacite
    P->values = (int*)realloc(P->values,sizeof(int)*P->capacity);
    if(P->values==NULL) error();
  }
  P->values[size] = x;
  P->size++;
}
```

13.1.4 Algorithmique

Déplacer. On se donne une pile P et on veut en déplacer les éléments dans une autre pile P', dans un ordre arbitraire. Pour cela, on récupère le sommet de P et on le place dans P'. Ensuite,

on dépile P. Notez que l'on n'impose pas que P' soit vide.

Algorithme 23 : Déplacer

```
Entrées : P pile à déplacer dans P'
tant que non estVide(P) faire
    v ← Sommet(P)
    Empiler(P', v)
    Dépiler(P)
```

On propose aussi une version récursive.

Algorithme 24 : DéplacerRec

```
Entrées : P pile à déplacer dans P'
si non estVide(P) alors
    v ← Sommet(P)
    DéplacerRec(depiler(P), empiler(P', v))
```

Les éléments de P sont ainsi placés en haut de la pile P', mais dans l'ordre inverse. Le dernier élément de P se retrouve donc en haut de P'. Si on veut déplacer une pile en conservant l'ordre, il faut utiliser l'algorithme précédent deux fois. Inverser l'ordre deux fois revient à l'ordre initial (on appelle cette opération une involution).

Complexité. Pour étudier la complexité d'un algorithme manipulant un type abstrait, on associe une complexité à chaque opération élémentaire. Pour les piles, il est raisonnable de dire que chaque opération est en temps constant, i.e. en $\mathcal{O}(1)$ puisque les implémentations classiques le garantissent (en moyenne pour le dépilement avec tableau). L'algorithme 23 est linéaire en le nombre d'élément de P, i.e. $O(n)$ si n est le nombre d'éléments dans P :

— chaque étape de la boucle *while* est en temps constant,

— la boucle itère n fois puisque P contient n éléments.

Copier. On a perdu la pile P en la dépilant. Si on veut copier, il faut remplir une deuxième pile en parallèle quand on dépile.

Algorithme 25 : Copier

```
Entrées : P pile
P'' ← creerPile()
P'' ← Deplacer(P, P'')
tant que non estVide(P'') faire
    v ← valeur(P'')
    P'' ← depiler(P'')
    P ← empiler(P, v)
    P' ← empiler(P', v)
renvoyer P'
```

Cet algorithme renvoie une nouvelle pile P' identique à P, tout en conservant intacte la pile P. Sa complexité est aussi linéaire :

— `Déplacer` est linéaire,

— la boucle *while* est linéaire par les mêmes arguments que l'étude de complexité de `Déplacer`.

13.2 Files

13.2.1 Type abstrait

De même, une file permet de stocker une collection d'objets. Mais cette fois, les opérations élémentaires sont :

CreerFile() : renvoie une file vide nouvellement créee.

EstVide(F) : renvoie un booléen qui est vrai si et seulement si la file F est vide,

Enfiler(F,x) : insère x à la file F,

Défiler(F) : supprime l'élément le plus anciennement inséré avec `Enfiler`,

Valeur(F) : renvoie l'élément de la file F le plus anciennement inséré.

Cette fois, il faut visualiser une file d'attente. Donc, à l'inverse de la pile, la file est une structure FIFO : first in, first out. Le premier élément à être arrivé dans la file (avec `Enfiler`) sera le premier à en sortir (avec `Défiler`).

13.2.2 Implémentation des files par chaînage

Les données sont encore stockées dans une chaîne de cellules. On stocke dans une structure `queue` (*file* en anglais) l' adresse de la première et de la dernière cellule. La première représente la tête de la file et la dernière sa queue : il est facile de retirer une cellule au début et d'en rajouter une à la fin. L'inverse aurait demandé de parcourir la chaîne depuis le début pour supprimer à la fin.

```
struct queue{
  struct cell * begin;
  struct cell * end;
}
typedef struct queue queue; //pas d'etoile cette fois
//queue est un alias pour struct queue

queue creerFile(){
  queue newQ;
  newQ.begin = NULL;
  newQ.end = NULL;
  return newQ;
}

int estVide(queue Q){
  return Q.begin == NULL; //Q.end==NULL fonctionne aussi
}

queue defiler(queue Q){ //meme code que depiler, on passe par valeur
  if(estVide(Q)) error();
  struct cell * tmp = Q.begin;
  Q.begin = Q.begin->next;
  free(tmp);
  return Q;
}
```

```
queue enfiler(queue Q, int x){
  struct cell * newCell = (struct cell *)malloc(sizeof(struct cell));
  if(newCell==NULL) error();
  newCell->value = x;
  newCell->next = NULL;
  Q.end->next = newCell;
  Q.end = Q.end->next;
  return Q;
}

int valeur(queue Q){
  if(estVide(Q)) error();
  return Q.begin->value;
}
```

13.2.3 Des files avec des piles

Méchanisme. Une file peut être implémentée avec deux piles :

— une première nommée *entrée*, qui représente la queue de la file,

— une seconde nommée *sortie*, qui représente la tête de la file.

Un élément de la file est dans une seule des deux piles. Il suit les étapes suivantes :

1. on l'enfile, ce qui l'empile au sommet de la pile *entrée*,
2. on le déplacera à un moment de la pile *entrée* à la pile *sortie*,
3. on le défile lorsqu'il est au sommet de la pile *sortie* en dépilant cette dernière.

Le déplacement d'un élément de la pile *entrée* à la pile *sortie* se fait lorsque la pile *sortie* est vide : on déplace entièrement *entrée* dans *sortie*.

Cette opération inverse l'ordre des éléments et c'est nécessaire. La pile *entrée* est dans l'ordre inverse de la file. En effet, les éléments en tête de la pile *entrée* sont les derniers ajoutés donc les derniers de la file. Pour pouvoir défiler dans l'ordre, il faut que la pile *sortie* soit dans le même ordre que la file. C'est pour cela qu'on inverse l'ordre.

Opérations élémentaires. On exprime les opérations de la file à partir de celles de la pile.

Algorithme 26 : creerFile

```
1 F : File
2 F.entree ← creerPile()
3 F.sortie ← creerPile()
4 renvoyer F
```

Algorithme 27 : estVide

```
   Entrées : F une file
1 renvoyer estVide(F.entree) et estVide(F.sortie)
```

Algorithme 28 : valeur

```
   Entrées : F une file
1 si estVide(F.sortie) alors
2 |  si estVide(F.entree) alors
3 |  |  #erreur
4 |  déplacer(F.entree, F.sortie)
5 renvoyer valeur(F.sortie)
```

Algorithme 29 : défiler

```
   Entrées : F une file
1 si estVide(F.sortie) alors
2 |  si estVide(F.entree) alors
3 |  |  #erreur
4 |  sinon
5 |  |  déplacer(F.entree, F.sortie)
6 dépiler(F.sortie)
```

Algorithme 30 : enfiler

```
   Entrées : F une file, x un objet
1 empiler(F.entree, x)
```

Complexité. Les opérations `creerFile`, `estVide` et `enfiler` sont en temps constant. En revanche, `valeur` et `depiler` sont potentiellement linéaire puisqu'elle peuvent nécessiter un déplacement d'une pile sur l'autre. Heureusement, leur complexité amortie est constante.

Ici, en faisant transiter n éléments par une file, certaines étapes seront linéaires mais il y a en a peu, ce qui permet d'obtenir une complexité totale qui est elle-même linéaire. Si on divise par le nombre d'éléments, la complexité amortie est donc constante.

Pour le justifier, on étudie le parcours d'un élément. Chaque élément ne demande que cinq appels (donc un nombre constant) à des opérations élémentaires de la pile :

— `empiler` pour le rajouter à *entrée*,
— `sommet` pour le récupérer dans la pile *entrée*,
— `sommet` pour le récupérer dans la pile *entrée*,

— `depiler` pour le supprimer d'*entrée*,
— `empiler` pour le rajouter à *sortie*,
— `depiler` pour le supprimer de *sortie*.

Pour être plus complet, on peut aussi compter les appels à `estVide` : au plus quatre par élément. Deux sont explicitement écrits dans `Défiler`, un autre dans la boucle de `Déplacer` et un en fin de boucle (si l'élément est toujours seul par exemple).

13.2.4 Algorithmique

Palindrome. On se donne une chaîne de caractères s de longueur n et on cherche à vérifier si elle représente un palindrome, c'est dire si elle est symétrique par rapport à son milieu. Plus formellement, on veut vérifier si $\forall i \in [\![0; n-1]\!], s[i] = s[n-i-1]$.

On va placer notre chaîne de caractères dans une pile et dans une file en parallèle en la parcourant par indices croissants. En dépilant la pile obtenue, notre chaîne de caractères arrive dans l'ordre inverse, tandis qu'en défilant la file, elle est dans le bon ordre. Si on prend deux valeurs de même rang dans chaque structure, ce sont deux caractères à des positions symétriques dans la chaîne de caractères. Il faut donc vérifier qu'ils soient égaux.

Algorithme 31 : Palindrome

```
Entrées : s une chaîne de caractères de taille n
1  P ← creerPile()
2  F ← creerFile()
3  for i de 0 à n − 1 do
4  |  P ← empiler(P, s[i])
5  |  P ← enfiler(F, s[i])
6  tant que non estVide(P) faire
7  |  si valeur(P) ≠ valeur(F) alors
8  |  |  renvoyer Faux
9  |  P ← depiler(P)
10 |  F ← defiler(F)
11 renvoyer Vrai
```

La complexité est linéaire puisqu'on copie une chaîne de caractère dans une pile et une file, Donc une boucle à n étapes dont chaque étape est en $O(1)$. Ensuite on dépile et on défile deux structures de taille n, ce qui ajoute une complexité linéaire.

13.3 Exercices

Exercice 25. Pour une pile, une rotation place l'élément du haut tout en bas. Proposer un algorithme qui fait n rotations. Quelle est la complexité de votre algorithme ? Essayez linéaire.

Exercice 26. Manipulation d'une pile.

1. On veut séparer une pile d'entiers en deux piles : les pairs et les impairs. Proposer un algorithme récursif.
2. Proposer un algorithme qui détermine la plus grande somme de deux valeurs consécutives dans une pile.

Exercice 27. On se donne une chaîne de caractères et on souhaite savoir si elle est bien parenthésée. Pour cela, il faut que toute parenthèse ouverte soit refermée, une et une seule fois. Ainsi, si on parcourt la chaîne de caractère de gauche à droite, il faut qu'à tout instant on ait vu au moins autant de parenthèses ouvrantes que fermantes. Il faut aussi qu'à la fin du parcours, on en ait vu exactement autant de chaque.

1. Proposer un algorithme basé sur un compteur afin de vérifier si une chaîne de caractères est bien parenthésée.
2. Maintenant, on ne se limite plus aux parenthèses : il y a aussi des accolades '{', '}', des crochets '[', ']', etc... On se donne les fonctions suivantes :
   — `ouvrant`(c) qui renvoie Vrai si et seulement si le caractère c est une parenthèse/accolade/crochet/... ouvrant,
   — `fermant`(c) qui renvoie Vrai si et seulement si le caractère c est une parenthèse/accolade/crochet/... fermant,
   — `inverse`(c) qui pour un signe fermant, renvoie l'ouvrant correspondant. `inverse`('}') = '{'.

   Il faut cette fois être plus vigilant : on ne peut refermer une accolade que si le dernier signe ouvrant étant aussi une accolade. Par exemple, `"(abc{de)fg}"` n'est pas correct ! Proposer un algorithme vérifiant si une chaîne de caractères est correcte.

Exercice 28. Soit deux positions p_1, p_2 sur un échiquier de taille $n \times n$. En au moins combien de coups peut-on déplacer un cavalier de p_1 à p_2 ?

Exercice 29. Trier des piles et des files.

1. Proposer un algorithme qui trie une pile sans utiliser d'autre structure de données que des piles. Quelle est la complexité de votre algorithme ?
2. On veut rajouter une opération aux piles : `Min`. Que rajouter à une pile pour pouvoir répondre à cette requête en temps constant, en supposant que toutes les opérations élémentaires sont elles mêmes en temps constant. Indice : utiliser une pile alternative.
3. Comment se passer de la pile alternative ?
4. Proposer un algorithme qui tri une file. Vous n'avez pas le droit d'utiliser une autre structure de donnée, juste un nombre fixé de variables.

Chapitre 14

Listes chaînées

Afin de manipuler une séquence de données, on peut définir un type abstrait appelé *liste* ou *vecteur*. Les données sont placées dans des cellules. Ce type abstrait se dérive en deux variantes : une disposition des cellules contigüe (tableaux), ou chaînée (listes chaînées). On va étudier la version chaînée.

Remarque. *En Python, ce type* list *est implémenté avec des tableaux, tandis qu'en OCaml, ce sont des listes chaînées.*

14.1 Définition

Une liste simplement chaînée est une collection d'éléments constituée d'une séquence de cellule, chacune contenant un élément. Chaque cellule se voit associer une unique position, que l'on peut penser comme une référence. On suppose qu'il existe une position spécifiant la fin de la liste, strictement après la dernière cellule.

Le type abstrait est muni des opérations élémentaires suivantes :

CreerListeChainee() : créer une liste chaînée vide et la renvoie,

EstVide(L) : renvoie Vrai si et seulement si la liste chaînée L est vide,

Début(L) : renvoie la position de la première cellule de la liste chaînée L,

Valeur(L,pos) : renvoie le contenu de la cellule de la liste chaînée L à la position *pos*,

Suivant(L,pos) : renvoie la position suivant *pos* dans la liste chaînée L,

PositionFin(L,pos) : renvoie Vrai si et seulement si la position *pos* est la fin de la liste,

InsererDebut(L,x) : insère l'élément x au début de la liste chaînée L,

SupprimerDebut(L) : supprime la première cellule de la liste chaînée L,

InsererApres(L,pos,x) : insère l'élément x après la cellule de position *pos* dans la liste L,

SupprimerApres(L,pos) : insère l'élément après la cellule de position *pos* dans la liste L.

Les opérations élémentaires basées sur une position prennent aussi la liste en paramètre : les positions peuvent être relatives ! Par exemple, dans un tableau, on peut utiliser les indices ou les adresses des cases. Le premier cas nécessite de spécifier aussi la structure.

Remarque. *Ici, on a choisi de ne pas avoir d'opérations élémentaires pour modifier le contenu d'une cellule. On peut simuler cette opération avec une suppression et un ajout. De même que pour l'accès au sommet d'une pile, il peut être pertinent de l'ajouter.*

D'un autre côté, on a choisi d'ajouter l'opération élémentaire `EstVide` *par cohérence avec les types abstraits précédemment définis. Mais cette opération est facultative : il suffit de composer* `Début` *et* `PositionFin`*.*

Vous pourrez rencontrer plusieurs variantes des listes simplement chaînées, notamment :

— Les listes doublement chaînées. L'idée est qu'on peut revenir en arrière. On se donne pour cela l'opération élémentaire `Précédent`.

— Les listes circulaires : l'élément suivant le dernier est le premier. Attention aux conditions d'arrêt de vos boucles de parcours.

14.2 Algorithmique

En guise d'exemple, on va décrire un algorithme qui affiche le contenu d'une liste chaînée.

Algorithme 32 : Affichage d'une liste chaînée itérativement

Entrées : L liste chaînée d'entiers.

```
1 pos ← Début(L)
2 tant que non PositionFin(L, pos) faire
3 |   Afficher(valeur(L, pos))
4 |   pos ← suivant(L, pos)
```

Une version récursive serait la suivante :

Algorithme 33 : AffichageAux pour afficher une liste chaînée récursivement

Entrées : L liste chaînée d'entiers, pos une position de L.

```
1 si non PositionFin(L, pos) alors
2 |   Afficher(valeur(L, pos))
3 |   AffichageAux(L, Suivant(L, pos))
```

Algorithme 34 : AffichageRec pour le premier appel à la fonction récursive

Entrées : L liste chaînée d'entiers.

```
1 pos ← Début(L)
2 AffichageAux(L, pos)
```

14.3 Implémentation

En Python

Vous pouvez utiliser *Collections.deque* qui sont des listes doublement chaînées. Pour les implémenter à la main, des notions de programmation orientée objet sont nécessaire. On ne traitera pas ce sujet ici.

En C

L'implémentation des listes chaînées peut se faire de manière très similaire à celle proposée pour les piles avec chaînage. On peut reprendre ce qui a déjà été fait de la sorte :

```
typedef stack list; //finalement une liste, c'est une pile plus complexe

list creerListe(){
  return creerPile();
}

list supprimerDebut(list L){
  return depiler(L);
}

list insererDebut(list L, int x){
  return empiler(L,x);
}
```

Structure. On va proposer une version alternative qui n'utilise que des cellules. Comme vu avec les piles, une cellule contient un couple constitué d'un élément et d'un pointeur vers la cellule suivante. Une liste peut alors être considérée comme un pointeur sur la première cellule.

```
  struct cell{
    int value;
    struct cell * next;
  };

  typedef struct cell * list;
```

Il suffit ensuite de manipuler le type *list* qui est un pointeur sur une cellule. La liste vide est un pointeur égal à NULL. Il est alors plus simple d'écrire des programmes récursifs : `suivant` renvoie la sous liste privée du premier élément.

Opérations élémentaires. Si on donne à une fonction une variable de type `list`, bien qu'il s'agisse d'un pointeur, le passage peut être considéré par valeur : on fait une copie de l'adresse de la première case. Cette adresse ne peut donc être modifiée à l'extérieur de la fonction. On peut alors soit passer par référence en manipulant des `list*`, soit renvoyer la nouvelle adresse lorsqu'elle change. On propose d'implémenter cette dernière méthode.

Remarque. *Passer les listes globales en paramètre devient facultatif dans cette implémentation lorsque l'on a une position donnée. Les positions sont des adresses donc absolues. Modifier une cellule à partir de sa position la modifie dans la liste globale.*

```
    list creerList(){
      return NULL; //puisque la liste est vide
    }
```

```
int estVide(list L){
  return L==NULL;
}

int valeur(list pos){
  if(estVide(pos)) error();
  return pos->value;
}

list suivant(list pos){
  if(estVide(pos)) error();
  return pos->next;
}

list insererDebut(list L, int x){
  list newCell = (list)malloc(sizeof(struct list));
  newCell->value = x;
  newCell->next = L; //l'ancienne premiere cellule devient la deuxieme cellule
  return newCell; //on renvoie la nouvelle premiere
}

list supprimerDebut(list L){
  if(estVide(L)) error();
  list newBegin = L->next; //met de cote la deuxieme case
  free(L); //supprime la premiere case
  return newBegin;
}

void insererApres(list pos, int x){
  if(estVide(pos)) error();
  pos->next = insererDebut(pos->next,x);
}

void supprimerApres(list pos){
  if(estVide(pos) || estVide(pos->next)) error();
  pos->next = supprimerDebut(pos->next);
}
```

14.4 Exercices

Exercice 30. Position dans une liste chaînée.

1. Ecrire une formule de récurrence exprimant la position d'un élément dans une liste ($-\infty$ si l'élément donné n'y est pas).
2. En déduire un algorithme récursif.

Exercice 31. Insertion en fin de liste chaînée.

1. Ecrire un algorithme qui permet d'insérer un élément à la fin d'une liste simplement chaînée.

2. Ecrire un algorithme qui permet de copier une liste chaînée dans le même ordre. Deux versions : une avec la question précédente, une sans.
3. Quelles sont les complexités de vos deux algorithmes ?

Exercice 32. On représente des ensembles avec des listes simplement chaînées.

1. Ecrire une fonction d'insertion. Doublons interdits !
2. Ecrire une fonction qui prend en paramètre une liste et renvoie l'ensemble correspondant (i.e. retire les doublons). Quelle est sa complexité ?
3. Ecrire une fonction qui fait l'union entre deux ensembles. Quelle est la complexité ? Proposer une optimisation si les éléments sont des entiers.

Exercice 33. Ecrire un algorithme qui permet d'insérer un élément au milieu d'une liste simplement chaînée.

Exercice 34. Ecrire un algorithme qui calcule le plus long plateau d'une liste chaînée. Un plateau est une séquence de cellules consécutives contenant une même valeur.

Chapitre 15

Structures arborescentes

On donne une présentation rapide des arbres.

15.1 Arbres binaires

Définition. On se donne un objet que l'on appelle Feuille. On définit un arbre inductivement de la façon suivante :

— soit c'est une Feuille,

— soit c'est un couple (g, d) avec g et d des arbres binaires. On dit que g est le fils gauche de l'arbre et que d est le fils droit.

Les arbres sont représentés à l'envers. On appelle le haut la racine.

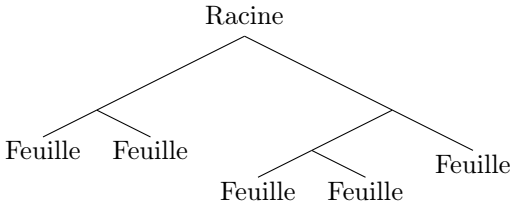


FIGURE 15.1 – Représentation d'arbre binaire.

Remarque. *De par leur construction inductive, les arbres sont plus facilement (voire obligatoirement) manipulés avec des algorithmes récursifs.*

Type abstrait. On introduit le type abstrait correspondant à l'objet défini ci-dessus. On se donne les opérations élémentaires suivantes :

CreerFeuille() : renvoie une feuille.

EstFeuille(A) **:** renvoie Vrai si et seulement si l'arbre A est une feuille.

CreerArbre(A_g, A_d) **:** renvoie l'arbre dont le fils gauche est A_g et le fils droit A_d.

Gauche(A) **:** renvoie le fils gauche de A.

Droit(A) **:** renvoie le fils droit de A.

Algorithmique. Un exemple simple : le calcul de la hauteur (profondeur maximale d'une feuille par rapport à la racine).

Algorithme 35 : Hauteur

Entrées : A un arbre binaire

```
1 si EstFeuille(A) alors
2     renvoyer 0
3 sinon
4     renvoyer 1 + max{Hauteur(Gauche(A)), Hauteur(Droit(A))}
```

On peut prouver par induction que cet algorithme est linéaire en le nombre de feuilles. Intuitivement, il faut aller visiter toutes les feuilles et chaque feuille ne demande qu'un nombre constant d'opération.

15.2 Arbres binaires étiquetés

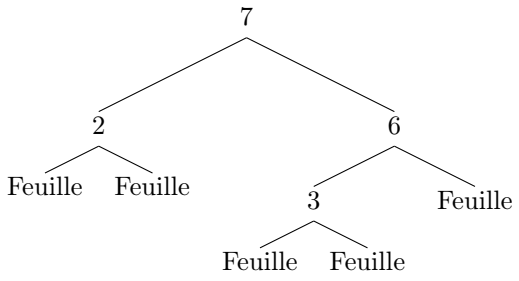
Afin de stocker des éléments dans ce type, on peut rajouter des étiquettes. Soit E un ensemble d'étiquettes. Un arbre binaire étiqueté est :

— soit une Feuille,

— soit un triplet (e, g, d) avec $e \in E$ et g, d deux arbres binaires étiquetés. On dit que e est l'étiquette (de la racine) de l'arbre. On appelle ce triplet un noeud interne.

Un noeud est soit une feuille, soit un noeud interne.

Visuellement, un arbre binaire étiqueté est représenté comme sur la figure 15.2



On se donne les opérations élémentaires suivantes :

CreerFeuille() : renvoie une feuille.

EstFeuille(A) : renvoie Vrai si et seulement si l'arbre A est une feuille.

CreerArbre(x, A_g, A_d) : renvoie l'arbre d'étiquette x, de fils gauche A_g et de fils droit A_d.

Etiquette(A) : renvoie l'étiquette de A.

Gauche(A) : renvoie le fils gauche de A.

Droit(A) : renvoie le fils droit de A.

Algorithme 36 : Recherche

Entrées : A un arbre binaire étiqueté sur E, $x \in E$

```
si EstFeuille(A) alors
    renvoyer Faux
sinon
    renvoyer Etiquette(A) = x ou Recherche(Gauche(A, x)) ou Recherche(Droit(A, x))
```

Cet algorithme est linéaire en le nombre de noeud.

Remarque. *Pour optimiser cet algorithme, on peut préciser que les* ou *sont paresseux : si l'élément est trouvé à gauche, pas besoin de chercher à droite. Cela ne change pas la complexité en pire cas.*

15.3 Arbres binaires de recherche

On se donne un ordre total sur E (i.e. on peut comparer tous les éléments). Un arbre binaire de recherche est un arbre binaire étiqueté sur E tel que, si $A = (e, g, d)$ un arbre n'étant pas une Feuille, alors :

— g et d sont des arbres binaires de recherche,
— si g est un arbre qui n'est pas une feuille, et d'étiquette e', alors $e' \leq e$,
— si d est un arbre qui n'est pas une feuille, et d'étiquette e', alors $e' \geq e$,

Remarque. *Par récurrence, on peut montrer que toutes les étiquettes du côté gauche sont inférieures à e et toutes celles à droites sont supérieures. En "aplatissant" l'arbre, les étiquettes sont dans l'ordre.*

Exercice 35. L'algorithme 36 fonctionne pour les arbres binaires étiquetés donc en particulier pour les arbres binaires de recherche. Mais pouvez vous trouver mieux ? Quelle complexité pire cas obtenez vous ?

15.4 Exercices

Exercice 36. Proposer un algorithme calculant le cardinal d'un arbre binaire.

Exercice 37. Proposer un algorithme qui vérifie si un arbre binaire est parfait (si toutes les feuilles ont la même profondeur).

Exercice 38. Proposer un algorithme qui calcule la liste chaînée des étiquettes paires d'un arbre binaire dont les étiquettes sont des entiers. Eviter si possible toute concaténation/renversement de liste chaînée.

Exercice 39. Comment calculer la hauteur maximal d'un arbre non enraciné (la racine peut être n'importe qui) ? On veut un algorithme linéaire.

Exercice 40. On dit qu'un arbre binaire est équilibré si la hauteur de ses deux sous arbres diffère au plus de 1 et qu'ils sont tout deux équilibrés. On appelle ces arbres des AVL. Proposer un algorithme calculant le nombre d'arbres binaires équilibrés à n noeuds. On veut un algorithme linéaire.