Informatique MP2I

Hector Dang-Nhu

2023-2024

Table des matières

Chapitre I

Le C, une meilleure granularité dans la gestion des ressources

Le C [1] est l'autre langage de programmation principal au programme de MP2I. Il est l'outil de choix pour comprendre et manipuler les ressources de la machine avec précision.

Le standard en vigueur dans le programme est la norme C99.

1 Présentation et utilisation de base du C

1.1 Un premier programme en C

On s'intéresse à un premier programme en C.

```c
#include <stdio.h>
int main() {
    printf("Bonjour au monde !");
    return 0;
}
```

Afin de pouvoir exécuter ce programme, nous avons besoin d'un compilateur qui s'occupe de transformer le programme (c'est-à-dire le fichier de texte) en un fichier exécutable par la machine (toujours un fichier, mais binaire).

Définition 1.1 : Compilation

La **compilation** est le processus de transformation d'un programme d'un langage vers un autre, usuellement de plus bas niveau, c'est-à-dire plus proche de la machine et donc plus facile à exécuter. Le logiciel qui réalise la compilation est appelé **compilateur**.

Le compilateur le plus commun pour programmer en C est `gcc` (*GNU Compiler Collection* [2]). La commande du même nom dans un terminal unix permet de compiler le fichier.

Par exemple, si l'on suppose que le programme précédent a été enregistré dans le fichier `bonjour.c`, il faudra réaliser la commande :

```
gcc bonjour.c
```

Cela crée un fichier `a.out` que l'on peut ensuite exécuter avec la commande `./a.out`. Bien sûr, on peut modifier le nom de sortie du fichier grâce à l'option `-o` de la commande `gcc`.

1. https://fr.wikipedia.org/wiki/C_(langage)
2. https://fr.wikipedia.org/wiki/GNU_Compiler_Collection

```
gcc -o bonjour bonjour.c
```

Pour utiliser le code, il faut donc exécuter le fichier bonjour (qui n'a pas d'extension), à l'aide de la commande ./bonjour.

gcc dispose de nombreuses d'options, et en particuliers des options d'optimisation du code produit que nous n'aurons pas l'occasion d'étudier.

En pratique nous pouvons utiliser un interpréteur en ligne [3] pour le C, comme ce que nous avons fait pour OCaml. Dans tous les cas, le programme, très poliment, nous donne la sortie suivante :

```
./bonjour
```

```
Bonjour au monde !
```

Nous allons désormais nous intéresser à l'anatomie de notre premier programme. La première ligne est la suivante :

```
#include <stdio.h>
```

Il s'agit d'une inclusion d'un fichier d'entête, d'extension .h (pour *Header*, entête) qui permet d'utiliser des fonctions de la bibliothèque standar. Dans ce cas, stdio (pour *STanDard Input Output*), qui contient la plupart des fonctions de gestion des entrées et sorties classiques.

Il existe de nombreux entêtes classiques.

Exemple 1.1 : Entêtes pour l'écrit

À l'écrit, on suppose toujours travailler sous les hypothèses que les entêtes suivants :

- assert.h : principalement la macro assert qui permet de faire des assertions à l'exécution du code ;
- stdbool.h : les définitions qui permettent de définir et manipuler les booléens ;
- stddef.h : types size_t et ptrdiff_t qui servent pour la manipulation mémoire ;
- stdint.h : différents types d'entiers (signés et non signés, taille en place mémoire...) ;
- stdio.h : des fonctions liées aux entrées et aux sorties de programme ;
- stdlib.h : la bibliothèque standard avec beaucoup de fonctions utiles dont malloc et free.

Nous nous intéresserons à la plupart de ces entêtes ultérieurement.

La suite est la définition d'une fonction avec un nom spécifique :

```
int main () {
    ...
}
```

En C, les fonctions sont définies par leur type de retour (ici int, un entier), le nom de la fonction (ici main), les arguments entre parenthèses (ici, aucun).

Le corps de la fonction, comme tous les blocs en C, est délimité par des accolades.

La fonction main a un rôle spécifique : c'est la fonction qui sera appelée si au début de l'exécution du programme. Il s'agit donc du *point d'entrée* du programme.

Il est possible de définir la fonction main avec plus d'arguments, mais nous verrons cela plus en détail avec les arguments d'un programme.

Dans le corps de la fonction, on peut trouver la première instruction :

```
printf("Bonjour au monde !");
```

Définition 1.2 : Expression

Une **expression** est un morceau de code qui est réduit à une valeur lors de l'exécution du programme.
Une expression est *évaluée* lorsque la valeur est calculée.

3. https://www.w3schools.com/c/tryc.php?filename=demo_compiler

Définition 1.3 : Instruction

Une **instruction** est un morceau de code qui forme une commande, un ordre à effectuer qui modifie l'état actuel de la machine.
Une instruction est *exécutée* lorsque cette ordre est réalisé.

Les instructions se terminent toujours par des un point-virgule ; en C. C'est une erreur commune que d'oublier les points-virgule.

Les points virgules permettent de délimiter les instructions, et donc d'en écrire plusieurs sur une même ligne, ou à cheval sur plusieurs lignes.

```
a = 1; b = 2;
c = 3; d = 4; e =
5;
```

En effet, en C, tout comme en OCaml, les sauts de lignes sont ignorées lors de l'analyse lexicale du code par le compilateur ou l'interpréteur.

Exemple 1.2 : Expressions et instructions

- `1 + 2` est une expression ;
- `f(1)` est une expression qui est le résultat de l'appel de `f` ;
- `f(1);` est une instruction qui a l'effet de l'appel à la fonction `f` ;
- `a = 1 + 2;` est une instruction qui affecte à la variable `a` la valeur de l'expression `1 + 2` ;
- `1 + 2;` est une instruction (qui fait un calcul, mais ne fait rien du résultat).

L'instruction en elle-même est un appel de fonction. L'appel de fonction est réalisé en C par le nom de la fonction (ici `printf`) suivi de ses arguments entre parenthèse (ici, une unique chaîne de caractère : `"Bonjour au monde !"`).

Cette fonction `printf` est issue de l'entête que nous avons ajouté à notre code et permet d'afficher une chaîne de caractère sur la sortie standard.

Enfin, le programme termine avec une dernière instruction :

```
return 0;
```

Celle-ci permet de mettre fin à l'exécution de la fonction `main`, et donc au programme, en renvoyant un code de retour qui indique si le programme s'est exécuté avec succès ou non. Un code de retour de 0 est souvent considéré comme une exécution sans erreurs.

Notre premier programme a donc bien la forme que nous attendions : un programme qui affiche une chaîne de caractère, puis termine.

1.2 Variables

En C, il est nécessaire de *déclarer* toutes les variables qui seront utilisées. Pour ce faire, on utilise la syntaxe suivante :

```
int a;
```

La première partie est le type de la variable, ici `int`, et la seconde partie est le nom de la variable `a`.

On peut ensuite *définir* cette variable en lui affectant une valeur (du bon type) :

```
a = 1;
```

Définition 1.4 : Déclaration

La **déclaration** est la spécification du nom, du type d'une variable, ainsi que d'autres propriétés selon les cas. Il s'agit d'une information pour le compilateur.

Définition 1.5 : Définition

La **définition** d'une variable est l'affectation initiale d'une variable. On parle parfois d' *initialisation*.

Bien sûr, on peut réaliser la déclaration et la définition au même moment :

```
int a = 1;
```

On peut de plus définir des constantes grâce à la syntaxe :

```
const int a = 1;
```

Il ne sera pas possible de modifier cette variable où que ça soit dans le code. Cela a deux avantages : le premier est d'être sûr que même une erreur dans le programme ne pourra pas modifier cette variable ; et par ailleurs le compilateur pourra réaliser des optimisations variées dessus.

Exemple 1.3 : Déclaration et définition

- `int i, j;` déclare deux variables `i` et `j` sans les définir ;
- `const int a = 0;` déclare un constante entière égale à 0 ;
- `float a = 0.;` déclare un flottant dont la valeur est 0 initialement.

On ne peut pas accéder à chaque variable partout dans le code : elles sont restreintes pour la plupart.

Définition 1.6 : Portée

La **portée** d'une variable est l'étendue du programme au sein de laquelle cette variable est accessible.

La portée est une caractéristique *statique*, c'est-à-dire que l'on peut calculer la portée d'une variable sans exécuter le code.

En C, la portée d'une variable ne peut pas plus grande que le bloc dans lequel elle est définie.

Par ailleurs, comme dans la plupart des langages, dans le cas d'un conflit (quand deux variables de même nom sont dans l'environnement), on prend la variable la plus spécifique, c'est-à-dire celle qui a été définie en dernier.

Définition 1.7 : Variable globale

Une variable est **globale** quand elle est déclarée à l'extérieur de tout bloc ou fonction du programme.
Potentiellement, la portée de cette variable peut-être tous le programme.

Attention !

Cela ne veut pas dire qu'on peut accéder à la variable partout dans le programme. Si par exemple une variable locale de même nom est définie, c'est cette dernière qui aura la précédence sur le nom et on ne pourra plus accéder à la variable globale.

Exemple 1.4 : Différentes variables et leurs portées

```c
const int a = 0;
const int b = 0;

int f(int b){}

int main(){
    {
        int a = 1;
    }
    return a;
}
```

La variable globale `a` est accessible dans le corps de la fonction `main` sauf le bloc défini à l'intérieur de la fonction `main`.
La variable globale `b` est accessible dans tous le code, sauf dans la fonction `f` où l'argument prend la précédence.

Exercice 1.1. Quelle est la sortie du code suivant ?

```c
#include <stdio.h>

const int a = 4;
const int b = 0;

int f(int b){
    printf("%d\n", b);
    {
        int b = 2;
        return b;
    }
}

int main() {
    {
        int a = 1;
        a = f(3);
    }
    return a;
}
```

1.3 Booléens

De sorte à pouvoir avoir des instructions conditionnelles, on a besoin d'un type particulier qui est composé des booléens.

Définition 1.8 : Booléen

Les **booléens** sont un type de variable qui peut prendre deux valeurs, soit Vrai, soit Faux.

Il existe de nombreuses manières de nommer le Vrai et le Faux. On utilise parfois des entiers (1 et 0), des lettres (V et F) ou des symboles logiques ($\top$ et $\bot$).
On utilise par ailleurs parfois les notations anglo-saxonnes (T et F).
Si la convention n'est pas évidente, elle sera explicitée à chaque fois.

En C, ces valeurs sont `true` et `false`. Elles sont définies dans `stdbool.h`. Nous aurons besoin de cet entête la plupart du temps quand l'on aura besoin de manipuler des variables booléennes.

Les booléens apparaissent naturellement quand l'on fait des comparaisons de nombres.

Exemple 1.5 : Opération de comparaisons en C

Il existe plusieurs opérations de comparaisons en C.
- `a==b` vérifie si a est égal à b ;
- `a!=b` vérifie si a est différent à b ;
- `a>b` vérifie si a est supérieur strictement à b ;
- `a>=b` vérifie si a est supérieur ou égal à b ;
- `a<b` vérifie si a est inférieur strictement à b ;
- `a<=b` vérifie si a est inférieur ou égal à b.

Attention, il faut bien utiliser le double égal pour vérifier l'égalité. En effet le symbole égal simple est réservé à l'affectation ou à l'initialisation.

Contrairement au Python il n'y a *pas* d'erreurs de syntaxe dans le code suivant :

```
1 if (a = 0) {...}
```

Cependant, le code n'aura pas le comportement attendu : la condition affectera 0 à a en vérifiant si la valeur affectée est non nulle.

On dispose d'un certain nombre d'opération sur les booléens.

Étant donné que le nombres de possibilités pour un nombre d'opérandes fixés est fini, on peut expliciter la valeur sur chacune dans un tableau appelé *table de vérité*.

Définition 1.9 : OU logique

Le OU logique est une opération à deux opérandes dont la valeur est Vrai si et seulement si au moins l'une de ses opérandes l'est.

A	B	A OU B
V	V	V
V	F	V
F	V	V
F	F	F

En C, le OU logique est réalisé par le symbole `||`.

Attention, en informatique, le ou correspond bien à la véracité d'au moins un des deux opérandes. Il ne s'agit donc pas du « fromage *ou* dessert ».

Définition 1.10 : ET logique

Le ET logique est une opération à deux opérandes dont la valeur est Vrai si et seulement si ses deux opérandes le sont.

A	B	A ET B
V	V	V
V	F	F
F	V	F
F	F	F

En C, le ET logique est réalisé par le symbole `&&`.

Définition 1.11 : Négation

Le NON logique est une opération à une opérande dont la valeur est différente.

A	NON A
V	F
F	V

En C, le NON logique est réalisé par le symbole `!`.

Exercice 1.2. Définition du OU exclusif

Le OU exclusif est une opération binaire qui vérifie si exactement l'une de ses opérandes vaut Vrai.

A	B	A XOR B
V	V	F
V	F	V
F	V	V
F	F	F

Comment implémenter le XOR à partir des autres opérations logiques ?

Cela peut nous permettre d'introduire une fonction `xor` :

```
bool xor(bool a, bool b) {
    /* Votre Code */
}
```

Exemple 1.6 : Utilisation du OU exclusif

L'utilisation du OU exclusif permet de simplifier quelques expressions logiques.
Ainsi, on peut définir deux fonctions qui calculent la sortie et la retenue d'une addition sur deux bits avec une retenue à l'aide des fonctions suivantes :

```
bool sortie(bool a, bool b, bool r)
{
    return xor(a, xor(b, r));
}
bool retenue(bool a, bool b, bool r)
{
    return (a && b)||(r &&(b || c));
}
```

Les opérations logiques `&&` et `||` sont dites paresseuses (ou séquentielle gauche-droite). Le calcul de la première opérande est réalisé dans tous les cas, mais le calcul de la seconde opérande n'est réalisé que si la première ne permettait pas de conclure.

Par exemple, dans le code suivant, la fonction `f` n'est pas exécutée, et le programme n'affiche donc rien.

```
bool f(){
    printf("Cette fonction s'execute");
    return true;
}
...
a = true || f();
...
```

Nous verrons dans le chapitre sur la logique qu'il y d'autres opérations sur les booléens que nous pouvons réaliser.

1.4 Structures conditionnelles

Les booléens nous permettent d'utiliser des structures conditionnelles. La plus simple étant la structure conditionnelle simple. Sa syntaxe en C est la suivante :

```
if (cond) {
    // Code a executer si la condition cond est verifiee
}
```

On peut par ailleurs rajouter le mot-clef `else` pour déterminer le code à réaliser dans le cas contraire.

```c
if (cond) {
    // Code a executer si la condition cond est verifiee
}
else {
    // Code a executer sinon
}
```

Exemple 1.7 : Exemple de structure conditionnelle simple

On se donne une fonction scanf de l'entête stdio.h qui permet, à l'exécution du programme, de récupérer une entrée dans la console. On ne s'intéressera pas au symbole & pour l'instant.

```c
#include <stdio.h>
int main()
{
    int date;
    printf("En quelle annee a eu lieu la revolution francaise ?\n");
    scanf("%d", &date);
    if (date==1789)
    {
        printf("C'est vrai !");
    }
    else
    {
        printf("C'est faux !");
    }
}
```

La chaîne de caractère \n permet de faire un retour à la ligne avant d'afficher la réponse.

Il est absolument nécessaire de mettre des parenthèses autour de votre condition :

```c
if i == 0
{
    ...
}
```

Quand il n'y a qu'une instruction à réaliser, on peut se passer des délimiteurs de blocs :

```c
if (cond)
    printf("Bim");
else
    printf("Bam");
```

Il existe une ambiguité sur le code suivant. Est-ce que le else est associé au premier ou au second if ?

```c
if (cond1)
if (cond2)
    {...}
else
    {...}
```

Il s'avère que le C, comme la plupart des langages, associent le else avec le if le plus proche, et le code précédent peut être interprété de la manière suivante :

```c
if (cond1){
    if (cond2)
        {...}
    else
        {...}
}
```

On peut utiliser une boucle pour pouvoir répéter un nombre arbitraire de code.

```c
while (cond)
    {Code a executer tant que cond est vrai}
```

Exemple 1.8 : Exemple de boucle

Voici un code qui affiche tous les entiers de 0 à 99.

```c
int i = 0;
while (i<100){
    printf("%d\n", i);
    i = i+1;
}
```

Une erreur relativement classique est d'oublier de modifier les variables qui servent dans la conditions. Par exemple, le code suivant ne termine pas :

```c
int i = 0;
while (i<100){
    printf("%d\n", i);
}
```

Une piste de solution est l'utilisation d'autres structures

```c
for (init ; cond ; incr)
    {/* Corps de la boucle*/}
```

Ici, *init* est l'opération à réaliser une seul fois au début, *cond* est la condition à vérifier à chaque tour de boucle pour savoir s'il faut continuer, et enfin *incr* est l'opération à réaliser à la fin de chaque boucle.

Exemple 1.9 : Exemple de boucle for

```c
for (int i = 0 ; i<100; i++){
    printf("%d\n", i);
}
```

Souvent, dans une boucle for, ou bien de manière générale quand on veut incrémenter de 1 un nombre, on utilisera l'instruction `a++;` qui augmente la valeur contenue dans `a` de 1.
De même, il existe les instructions de la forme `a--` qui diminue la valeur contenue dans `a` de 1.

Il existe deux instructions spéciales pour les boucles `break` et `continue`.
`break` met fin à la boucle, et `continue` revient au début de la boucle.

Exercice 1.3. Proposer un programme C qui affiche tous les nombres de 0 à 100 qui sont des carré parfait, c'est-à-dire les nombres de la forme n^2 où n est un entier.

En C, les booléens sont en réalité des entiers non signés (c'est comme cela qu'ils sont définits dans `stdbool.h`).

Plus encore, on pourrait se passer de booléens et utiliser des entiers pour les conditions :

```c
int i;
if (i)
    {/*A faire si i != 0*/}
else
    {/*A faire si i == 0*/}
```

Cependant, nous utiliserons spécifiquement les booléens pour cet usage :

```c
int i;
if (i!=0)
    {...}
else
    {...}
```

1.5 Fonctions

Comme dans la plupart des langages de programmations, on dispose en C d'outils pour pouvoir regrouper des instructions de sorte à mieux organiser un programme.

Le principal de ces outils consiste en les fonctions.

Définition 1.12 : Fonction

Une **fonction** est un élément syntaxique qui regroupe des instructions qui peuvent être exécutées d'un seul bloc.
Les fonctions peuvent avoir des **arguments** nécessaire à l'appel.
Le plus souvent, les fonctions ont une **valeur de retour** qui est le résultat de la fonction.
Lorsque que ces instructions sont exécutées, on parle d'un **appel** à la fonction.

Les fonctions, par analogie avec les mathématiques, sont des objets qui ont des arguments et une valeur de retour.

On peut définir des fonctions en C à l'aide de la syntaxe suivante :

```
type_retour nom_fonction(type_argument1 nom_argument1, type_argument2 nom_argument2,
    ...){
    //Corps de la fonction
}
```

Les arguments dans la déclaration et dans l'appel sont séparés par des virgules. L'ordre des arguments est important.
Par ailleurs il n'y a *pas* de point-virgule à la fin de la définition.

On peut réaliser un appel à une fonction à l'aide de la syntaxe suivante :

```
nom_fonction(argument1, argument,2, ...)
```

La syntaxe est proche de celle de la définition, mais on fera attention à ne pas repréciser les types des arguments.

Définition 1.13 : Prototype

Le **prototype** d'une fonction sont les informations concernant le type de retour et des arguments de la fonction.
On parle parfois de *signature*. Par ailleurs, on rajoute souvent les noms des arguments dans le prototype.

Les prototypes seront fournis dans la plupart des cas quand l'implémentation d'une fonction sera donnée. Il est important de conserver le type de retour, le type et l'ordres des arguments, et, le cas échanant, le noms des arguments.

Tout comme une variable, on peut déclarer une fonction sans la définir pour pouvoir la définir plus tard.

```
type_retour nom_fonction(type_argument1 nom_argument1, type_argument2 nom_argument2,
    ...);

/*...*/

type_retour nom_fonction(type_argument1 nom_argument1, type_argument2 nom_argument2,
    ...){
    /*...*/
}
```

On peut interrompre une exécution avec l'instruction return. Cela peut nous permettre d'arrêter l'exécution d'une fonction dans une boucle.

```
int racine_entiere(int n){
    int k = 0;
    while (true){
        if (k*k >= n)
            return k;
        k++;
    }
}
```

Exercice 1.4. Proposer une fonction `bool est_premier(int n)` qui renvoie si un entier est premier.

On peut utiliser le type `void` pour définir le type de retour de fonctions qui ne renvoient rien.

```
void affichier_entiers_inferieurs(int n){
    for (int i = 0; i<n; i++)
    {
        printf("%d\n", i);
    }
}
```

En réalité, parler de type pour `void` est abusif. En effet, il n'existe pas de valeur de ce type, et l'on ne peut pas définir de variable de ce type :

```
void a;
```

Il est donc parfois plus pertinent de parler du *mot-clef* `void`, plutôt que du type `void`. Par ailleurs nous verrons une autre utilisation du mot-clef `void`, dans un autre contexte, mais qui porte un sens totalement différent.

On parle parfois de *procédure* pour une fonction qui ne renvoie rien : l'idée est de considérer qu'il ne peut pas s'agir d'une fonction au sens mathématiques car elle n'a pas de valeur de retour.

Les procédures modifient l'environnement, que ça soit en modifiant la mémoire, ou en gérant des périphériques à l'extérieur de l'espace mémoire du programme.

De la même manière, on veut parfois éviter de nommer une fonction qui ne renvoie rien une fonction pour éviter la confusion avec les fonctions mathématiques. On a alors un terme générique qui regroupe les fonctions et les procédures : les *routines*.

De manière générale, on utilisera le terme fonction pour routine par abus de langage.

Exercice 1.5. Proposer une fonction à 3 arguments qui renvoie le plus grand des entiers en argument. La fonction aura le prototype `int plus_grand(int a, int b, int c)`.

2 Les entiers

En C, beaucoup de types sont structurés comme des entiers, et la compréhension de la représentation des entiers est donc un outil important pour pouvoir manipuler ces entiers.

2.1 Entiers naturels

Pour des raisons matérielles, nous voulons représenter les données en binaire. Il s'avère que les entiers peuvent être facilement représentés dans une écriture qui n'utilise que deux valeurs possibles pour chaque chiffre : la base deux.

Proposition 1.1 : Décomposition en base deux

Pour tout $n \in \mathbb{N}*$, il existe un unique $N \in \mathbb{N}$, et un unique $N+1$-uplet d'entiers a_0, a_1, ..., a_N tels que :

$$\forall k \leqslant N, a_j \in \{0,1\}$$
$$a_k = 1$$
$$\sum_{k=0}^{N} a_k 2^k = n$$

On écrit alors :

$$n = \overline{a_N a_{N-1} \ldots a_1 a_0}^2$$

Pareillement, pour des raisons matérielles, la nombre maximal de chiffres que nous pouvons utiliser pour représenter est souvent fixé, et on représentera donc les entiers en mémoire sur un certains nombres de chiffres que l'on appelle *bit* (pour *Binary digIT*).

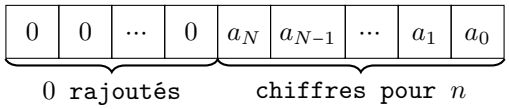
On peut décrire ces bits en fonction de l'importante numérique qu'ils ont dans le calcul de la valeur.

Définition 1.14 : Bit de poid fort, bit de poid faible

On parle de **bit de poid fort** pour le bit qui correspond au coefficient devant la plus grande puissance de 2.
On parle de **bit de poid faible** pour le bit qui correspond au coefficient devant la plus faible puissance de 2.

On peut parler des k bit de poids fort pour décrire les k bits qui correspondent aux coefficients devant les k plus grande puissance de 2.

Par exemple, sur N bits, on pourra représenter un entier n avec $k < N$ en mémoire de la manière suivante :

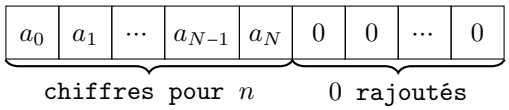


On a décider de faire apparaître les bits de poids les plus fort en premier, et de terminer par les bits de poids faible. Il s'agit d'une convention la plupart du temps utilisée.

Définition 1.15 : Boutisme

On parle de **boutisme** pour décrire l'ordre dans lequel les bits sont représentés.
On parle de **grand-boutisme** quand on représente les bits de poids fort en premier.
On parle de **petit-boutisme** quand on représente les bits de poids faible en premier.

Ainsi, avec l'orientation petit-boutiste, on obtient la représentation suivante :



Pour trouver la représentation en binaire d'un nombre, on peut procéder par division euclidienne successives.

Algorithme 1.1 : Calcul de la représentation binaire d'un entier

Pour calculer la représentation binaire d'un nombre, on peut utiliser l'algorithme suivant :
Entrée : n un entier, N le nombre de bit
Sortie : t le tableau représentant en binaire le nombre
  $t \leftarrow$ un tableau de taille N rempli de 0
  Pour Chaque $0 \leqslant i \leqslant N-1$ **Faire**
    $q, r \leftarrow$ les entiers q et r le quotient et le reste de n par 2
    $t[N-1-i] \leftarrow r$
    $n \leftarrow q$
  Renvoyer t
On a bien utilisé l'ordre grand-boutiste.

Bien sûr, en pratique, on n'aura pas besoin de calculer la représentation binaire d'un nombre car celle-ci est déjà la manière dont on représente les nombres dans la machine.

Exercice 1.6. Comment représenter l'entier 75 sur 8 bits ?

Exercice 1.7. Que représente l'écriture binaire suivante ?

0	1	1	0	0	1	1	0

Proposition 1.2 : Gamme des entiers représentables

Sur N bits, on peut représenter les entiers de 0 à $2^N - 1$.

Exemple 1.10 : Entiers positifs en C

En C, les entiers ne sont pas nécessairement positifs en C, et il faut donc utiliser différents types. Par défaut, on dispose du type `unsigned int` qui représente un entier de même nombre de bit que `int` mais qui est toujours positif. Le nombre de bits précis n'est pas une contrainte du langage, mais une contrainte de la machine.
Par ailleurs, l'entête <stdint.h> rajoute les types suivants :
- `uint8_t` : entiers positifs sur 8 bits ;
- `uint16_t` : entiers positifs sur 16 bits ;
- `uint32_t` : entiers positifs sur 32 bits ;
- `uint64_t` : entiers positifs sur 64 bits sur les machines qui le supporte.

2.2 Entiers relatifs

On peut représenter les entiers positifs, mais on cherche désormais à représenter les entiers relatifs.

Une des solutions qui serait naturelle serait de représenter les entiers à l'aide de leur distance à zéro et de leur signe, mais il s'avère que cela n'est pas le mieux : on peut prendre une représentation de nombres relatifs qui permet de conserver le même algorithme pour l'addition.

Définition 1.16 : Nombres en complément à deux

La **représentation par complément à deux** d'un nombre s'obtient de la manière suivante :
- Si le nombre est positif, on utilise la représentation binaire ;
- si le nombre est strictement négatif, on utilise la représentation binaire de l'opposé, on inverse les 0 et les 1 dans cette représentation, et puis on ajoute 1 à cette représentation comme s'il s'agissait d'un entier naturel.

Ainsi, pour représenter le nombre −99 sur 8 bits, on commence par représenter le nombre 99, ce qui nous donne la représentation suivante :

0	1	1	0	0	0	1	1

On inverse les 0 et les 1 :

1	0	0	1	1	1	0	0

Enfin, on rajoute 1 en incrémentant le résultat :

1	0	0	1	1	1	0	1

On remarque que si le nombre est plus grand que 2^{N-1} où N est le nombre de bit, il peut y aura une ambiguïté entre le nombre représenté et un nombre négatif si le nombre commence par un 1.

Par convention, quand le bit de poids fort est égal à 1, c'est la représentation du nombre négatif, et quand le bit de poid fort est égal à 0, c'est le nombre positif qui est représenté.

Proposition 1.3

En complément à deux, un nombre représenté par les bits suivant :

a_{N-1}	$\cdots$	a_1	a_0

est égal à :

$$-a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k$$

Réciproquement, si un nombre peut s'écrire de cette manière, alors sa représentation sera celle qui est donnée au dessus.

Demonstration : On remarque qu'en inversant le procédé, on ne peut avoir qu'un seul nombre qui a cette représentation. Il suffit donc de montrer le sens réciproque.

Si $n \geqslant 0$ (c'est-à-dire si $a_{N-1} = 0$), le résultat est équivalent à ce qu'on avait proposé pour les entiers non signés.

Si $n = -2^{N-1}$, on peut vérifier que sa représentation est 1 suivi de 0.

Soit n un nombre qui s'écrit de la manière suivante avec des a_i dans $\{0,1\}$:

$$n = -2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k$$

On remarque que :

$$\begin{aligned} -n-1 &= 2^{N-1} - 1 + \sum_{k=0}^{N-2} a_k 2^k \\ &= \sum_{k=0}^{N-2} 2^k - \sum_{k=0}^{N-2} a_k 2^k \\ &= \sum_{k=0}^{N-2} (1-a_k) 2^k \end{aligned}$$

Donc le nombre $-(n+1)$ se représente en binaire par le tableau :

0	$(1-a_{N-1})$	$\cdots$	$(1-a_1)$	$(1-a_0)$

Or on remarque que, pour $n \neq -2^{N-1}$ décrémenter une représentation binaire puis inverser les bits, revient à inverser les bits puis incrémenter.

Donc n est bien représenté de la manière suivante :

1	a_{N-2}	$\cdots$	a_1	a_0

Ainsi, on a bien l'équivalence.

Cette propriété est terriblement pratique, en réalité, on pourrait s'en servir pour définir les entiers par complément à deux.

Pour la plupart des résultats sur les nombres relatifs, il est en général plus facile de se servir de cette propriété pour les démontrer. Et par exemple la suivante :

Proposition 1.4 : Gamme des entiers relatifs sur N bits

Sur N bits, on peut représenter les entiers relatifs de -2^{N-1} à $2^{N-1}-1$ sans ambiguïté.

Une remarque intéressante est qu'il n'y a qu'une seule manière de représenter le nombre 0.

Exemple 1.11 : Entiers relatifs en C

Le type `int` par défaut correspond à des entiers qui peuvent être à la fois positifs ou négatifs.
De la même manière que pour les positifs, l'entête <stdint.h> rajoute les types suivants :
- `int8_t` : entiers relatifs sur 8 bits ;
- `int16_t` : entiers relatifs sur 16 bits ;
- `int32_t` : entiers relatifs sur 32 bits ;
- `int64_t` : entiers relatifs sur 64 bits sur les machines qui le supporte.

2.3 Opérations sur les représentation

Étant donné que les entiers sont représentés de cette manière, on veut pouvoir réaliser des opérations directements sur cette représentation. Heureusement, la plupart des opérations que l'on veut réaliser vont pouvoir être effectuées grâce à l'aide des algorithmes classiques sur les bases données.

Ainsi, on peut augmenter de 1 un entier sans que cela ne pose de soucis :

Algorithme 1.2 : Incrémenter en base deux sur les entiers naturels

Pour incrémenter un nombre en base deux, on pars du bit de poids faibles, et on ajoute un en propageant l'éventuelle retenue.
Entrée : La représentation t d'un entier n sur N bits
Sortie : La représentation de $n+1$ sur N bits.

```
k ← N − 1
Tant Que k ⩾ 0 Faire
    Si t[k] = 0 Alors
        t[k] ← 1
        Renvoyer t
    Sinon
        t[k] ← 0
    k ← k − 1
Renvoyer t
```

De la manière dont on a construit notre algorithme, si l'on incrémente l'entier maximal que l'on peut représenter en mémoire, on revient sur 0.

Notre résultat est donc le bon, mais seulement modulo 2^N.

Exercice 1.8. Comment réaliser l'addition de deux nombres naturels ?

On peut montrer que le résultat, encore une fois, est le bon modulo 2^N même dans le cas d'un dépassement de mémoire.

Pour l'instant, nous avons fait marcher notre algorithme sur les entiers naturels, mais la magie des nombres en complément à deux est très justement qu'avec le même algorithme, on obtient le bon résultat :

Proposition 1.5 : Compatibilité des additions et des incrémentations sur la représentation des entiers naturels et relatifs

Pour ajouter n et m deux entiers relatifs, on peut considérer leur représentation comme une représentation d'entiers positifs, faire l'addition de ces entiers avec les algorithmes précédent, et considérer le résultat en tant qu'entier relatif.

Pour des raisons similaires au cas de l'addition dans les positifs, nous aurons le bon résultat modulo 2^N.

On dispose par ailleurs d'un certain nombre d'opérations sur les entiers.

Exemple * : Opérations sur les entiers en C

— Les opérateurs classiques $+$, $-$ et $*$;
— La division entière `/`, le modulo `%`.

2.4 Opérations bit à bit

Indépendamment du type d'une donnée, on peut se donner des opérations que l'on peut réaliser sur les bits d'une valeur en mémoire.

Définition 1.17 : Opération bit à bit

Une **opération bit à bit** est une opération qui opère directement sur les bits de la mémoire.

Par exemple si on se donne une opération $\odot$ et deux nombres A et B dont les représentations sont les suivantes :

a_{N-1}	a_{N-2}	$\cdots$	a_1	a_0

b_{N-1}	b_{N-2}	$\cdots$	b_1	b_0

On obtient le résultat suivant :

$a_{N-1} \odot b_{N-1}$	$a_{N-2} \odot b_{N-2}$	$\cdots$	$a_1 \odot b_1$	$a_0 \odot b_0$

Bien sûr, dans le cas d'une opération unaire, on n'aura besoin que d'un seul opérande.

Les opérations bit à bit sont donc étroitement liées aux opérations booléens : à partir de chaque opération booléennes, on peut créer une opération bit à bit qui lui correspond.

Exemple 1.12 : Opérations bit à bit en C

On a les opérations bits à bits suivantes en C :
— `|` le ou bit à bit ;
— `&` le et bit à bit ;
— `~` la négation bit à bit ;
— `^` le ou exclusif bit à bit.

Attention !

Il ne faut pas confondre les opérateurs booléens (`&&` et `||`) et les opérateurs bit-à-bit (`&` et `|`).

Exercice 1.9. Combien vaut (234|125)&127 ?

2.5 Décalage Mémoire

Définition 1.18 : Opérations de décalage logique

Un **décalage** est une opération qui décale à gauche ou à droite les bits d'une représentation binaire en complétant par des 0.

Cependant, cette opération n'a pas le comportement attendu sur les entiers relatifs : on peut changer le signe en complétant avec des 0 à gauche dans le cas du décalage à droite.

Définition 1.19 : Opérations de décalage arithmétique

Un **décalage** est une opération qui décale à gauche ou à droite les bits d'une représentation binaire en complétant par par des 0 dans le décalage à gauche, ou par un bit de signe dans le cas du décalage à droite.

Exemple 1.13 : Opérations de décalage en C

Les opérations de décalage en C sont << et >>.
- a << k décale de k bits vers la gauche la représentation de a ;
- a >> k décale de k bits vers la droite la représentation de a.

Ces décalages sont arithmétiques : à la compilation, le compilateur se sers des informations de typage pour savoir comment compléter dans le cas d'un décalage à droite (0 dans le cas d'un nombre positif, 1 dans l'autre cas).

Il s'avère qu'il est beaucoup plus rapide de faire un simple décalage en binaire que de réaliser des multiplications et c'est pourquoi on préfère souvent utiliser des multiplications par 2 quand on a le choix de prendre un facteur arbitraire.

En pratique, il n'est pas nécessaire de modifier le code pour utiliser des décalages spécifiquement au lieu de multiplication par deux, gcc s'occupera de faire les optimizations pertinentes (qui ne sont pas nécessairement des décalages).

3 Gestion de la mémoire

3.1 Gestion de la mémoire par un programme

Le programme dispose d'un espace mémoire dans lequel il travaille. Cette mémoire est organisée en différentes parties qui ont chacune un rôle différent.

Définition 1.20 : Organisation de la mémoire d'un programme

La mémoire dont dispose un programme est organisé en plusieurs zones :
- Le **texte** : correspond à l'espace mémoire où le programme est chargé en binaire ;
- La **pile** : qui sers à réaliser les différents appels de fonctions ;
- Le **tas** : qui sers à stocker les données dynamiquement indépendamment de la pile ;
- Le **segment de données** : qui contient les variables globales, statiques et certaines chaînes de caractère.

Il y a une multitude de choses à dire sur l'organisation de la mémoire d'un programme, et beaucoup de subtilités qui sont par exemple liés au subterfuge qui permet à un programme de vivre dans l'illusion qu'il dispose de l'intégralité de la mémoire, mais l'important pour l'instant sont l'utilisation que nous ferons de la pile et du tas.

Grâce à la pile, nous pourrons gérer les différents appels imbriqués des fonctions ainsi que leurs arguments et leurs valeurs de retours, et grâce aux tas, nous pourrons allouer des zones mémoires qui survivent aux différents appels de fonctions.

Le texte et le segment de données ont des applications et permettent de mieux comprendre le fonctionnement d'un programme, mais nous ne les verrons pas dans beaucoup plus de détails.

L'idée principale derrière la gestion de la mémoire est qu'à chaque zone de la mémoire, on veut que cette mémoire soit libre, soit réservée à exactement une donnée.

Définition 1.21 : Allocation

L' **allocation mémoire** est le procédé qui permet de réserver des parties de la mémoire.

De manière générale, on peut donc *allouer* une zone mémoire ou une variable quand on procéde à la réservation de cette zone.

Cependant, il existe un procédé symmétrique qui est la *libération* de la mémoire quand on retire la réservation qui avait été faite.

Les zones de la mémoire se distinguent non seulement dans les données que l'on va stocker dans chacune d'entre elle, mais aussi dans la manière dont on procède à l'allocation et à la libération de la mémoire.

Il s'avère que les données allouées qui ne sont pas initialisées vont avoir la valeur donnée par la mémoire au moment de l'allocation : si l'on fait rien, nous n'aurons pas que des zéro nécessairement.

De sorte à pouvoir manipuler des objets sur le tas et dans la pile, il va nous falloir un des concepts importants de la gestion de la mémoire, les *pointeurs*.

3.2 Pointeurs

Lors du calcul des arguments d'une fonction, ceux-ci sont recopiés dans une variable qui peut être manipulée à l'intérieur de la fonction sans affecter les éventuelles variables qui était initialement utilisées.

```c
# include <stdio.h>

void f(int n){
    n++;
    printf("n vaut %d dans la fonction f\n", n);
}

int main(){
    int n = 0
    f(n);
    printf("n vaut toujours %d dans la fonction main\n", n);
    return 0;
}
```

On dit que les paramètres sont *passés par valeur*.

Définition 1.22 : Passage par valeur

On dit qu'il y a **passage par valeur** lorsque le code appelé dispose d'une copie de ses paramètres.

En C, les arguments des fonctions sont toujours passé par valeur. Le problème est qu'il n'est pas possible de modifier des variables extérieur de cette manière.

La solution principale en C est d'utiliser des *pointeurs*, c'est-à-dire une adresse (la localisation) vers un autre endroit de la mémoire. Ainsi, quand une fonction aura accès à l'adresse d'une valeur, cette adresse sera certes une copie, mais la valeur à cette adresse dans la mémoire sera bel et bien

Définition 1.23 : Adresse mémoire

Une **adresse mémoire** est un entier qui désigne une zone particulière de la mémoire.

Définition 1.24 : Pointeur

Un **pointeur** est une variable qui contient une adresse mémoire.

Exercice 1.10. Selon les processeurs, les adresses mémoires sont stockées sur 32 ou 64 bits.

En supposant que chaque octet (c'est-à-dire ensemble de 8 bits) doit posséder une adresse propre, quelle est le nombre maximal d'octets qui peuvent être utilisés avec une mémoire sur 32 bits ? Sur 64 bits ?

Le C nous donne des outils pour manipuler les pointeurs.

Exemple 1.14 : Utilisation des pointeurs en C

On peut indiquer le type d'un pointeur à l'aide du symbole $*$ dans le type :

```c
int *aptr; // aptr est un pointeur vers un entier
```

On peut accéder à l'adresse d'une variable à l'aide du symbole & :

```c
int a = 0;
    aptr = &a;// aptr contient l'adresse vers la variable a
```

Par ailleurs, on peut utiliser le symbole $*$ pour modifier valeur contenue dans un pointeur selon le contexte :

```c
*aptr = 1; // On mets 1 dans la case memoire pointee par aptr
```

On peut aussi s'en servir pour accéder à la valeur contenue dans un pointeur.

```c
printf("%d\n", *aptr); // On affiche la valeur contenue dans l'adresse aptr
```

On dit alors que $*$ est un opérateur de *déréférencement*.

Exemple 1.15 : Exemple d'utilisation des pointeurs

```c
# include <stdio.h>

void f(int *n){
    *n++;
    printf("n vaut %d dans la fonction f\n", *n);
}

int main(){
    int n = 0;
    f(&n);
    printf("n vaut maintenant %d dans la fonction main\n", n);
    return 0;
}
```

À cela s'ajoute un pointeur spécifique, le pointeur NULL qui ne fait référence à aucune valeur dans la mémoire. C'est une sorte de valeur par défaut pour les pointeurs, et cela pourra nous servir à indiquer qu'une quantité n'existe pas.

Exemple 1.16 : Pointeur NULL en C

Le pointeur NULL est défini dans plusieurs entêtes classiques : stddef.h, stdio.h, stdlib.h et string.h.

```c
int *a = NULL;
```

Exercice 1.11. Proposer une fonction de prototype `void echanger(int *a, int *b)` qui échange la valeur contenue dans deux pointeurs, et proposer un programme qui illustre ce code :

```c
# include <stdio.h
void echanger(int *aptr, int *bptr){
    ...
}

int main(){
    int a = 87;
    int b = 65;

    printf("a vaut %d et b vaut %d", a, b);
    echanger(...);
    printf("a vaut %d et b vaut %d", a, b);
}
```

Nous disposons maintenant d'un outil pour, à partir d'une donnée, avoir accès à un autre moment dans la mémoire, et donc à une autre donnée.

3.3 Conventions d'appel sur la pile

La pile est l'endroit dans lequel tous les appels de fonction en cours sont

Définition 1.25 : Bloc d'activation d'un appel

Un **bloc d'activation** d'une fonction est un segment de donnée qui se trouve dans la pile et qui est découpé de la manière suivante :
- Paramètres de la fonction : les paramètres de la fonctions ;
- Adresse de retour : l'adresse qui pointe vers un élément du texte pour savoir où revenir dans le code après l'exécution de la fonction ;
- Variables locale à la fonction : les différentes variables locales qui sont calculées et modifiés dans la pile.

Exemple 1.17 : Exemple de fonctions imbriquées

```c
int f(int a, int b){
    int c = 2;
    return 0;
}

int main(){
    int d = 0;
    int e = 1;
    f(d, e);
    return 0;
}
```

Définition 1.26 : Durée de vie d'une variable

La **durée de vie** d'une variable est la période temporelle durant laquelle elle est allouée dans la mémoire.

Contrairement à la portée, la durée de vie est une caractéristique *dynamique* : c'est au cours de l'exécution que l'on peut en connaître exactement la durée de vie.

Cependant, la durée de vie d'une variable *ne* correspond *pas* au temps durant laquelle le programme exécute des instructions dans sa portée.

Par exemple, si une variable locale de même nom qu'une variable globale est utilisée dans le bloc à l'intérieur, la variable globale est toujours en vie, mais l'intérieur du code ne fait pas partie de sa portée.

```c
int x;
{
    int x;
    /* Ici, nous ne sommes plus dans la portee de la premiere variable x
    mais celle-ci est encore en vie. */
}
```

Une remarque importante dans le fonctionnement de la pile est le fait que les données alouées sur la pile (variables locales à une fonction) ne survivent pas à leur bloc d'activation.

```c
int *f(){
    /* Cree une variable entiere et en renvoie l'adresse */
    int a = 0;
    return &a;
}

int main(){
    int *b;
    b = f();
    // L'adresse contenue dans b n'est pas valide : il s'agissait d'une variable
    locale a la fonction f
    printf("%d\n", *b);
    return 0;
}
```

Si l'on veut conserver des variables, il va falloir allouer des données sur le tas.

3.4 Allocation sur le tas

La pile nous permettait d'avoir une allocation automatique lors des différents appels de routines. Le *tas* est une zone mémoire à l'organisation plus libre que la pile, toutes les allocations doivent être gérées manuellement.

Définition 1.27 : Séquence d'utilisation de l'allocation dynamique de la mémoire sur le tas

1. Le programme demande au système d'exploitation de lui attribuer un espace mémoire ;
2. Si possible, le système d'exploitation renvoie l'adresse de début d'un bloc de cette taille là et la réserve ;
3. Le programme utilise l'espace alloué ;
4. Le programme informe le système d'exploitation que l'espace mémoire peut être libéré ;
5. Le système d'exploitation libére cet espace mémoire.

Le fonctionnement exacte de l'allocation mémoire côté système n'est pas nécessaire du côté de l'écriture de la programmation. Le coût exacte de l'allocation mémoire peut être relativement compliqué à estimer : cela peut dépendre du nombre de bloc de mémoire qui ont été alloué, et aussi de la taille de la mémoire qui est demandé.

En pratique, on essayera d'ignorer les soucis potentiels que peuvent lever cette allocation mémoire, ou on fera des hypothèses sur le temps nécessaire à l'allocation.

Par ailleurs, les programmes en temps réels avec des contraintes de réactivités peuvent vouloir éviter autant que possible la question de l'allocation dynamique sur le tas.

Par ailleurs, pour des soucis de performances, il s'avère qu'utiliser la pile pour de grandes tailles de données, ou bien pour des données dont la taille est décidée dynamiquement au cours du programme, n'est pas une option raisonnable.

Nous verrons donc que l'allocation sur le tas possède un autre intérêt quand nous voulons allouer un espace dont la taille est décidé à l'exécution.

De sorte à pouvoir réaliser les allocations sur le tas en C, nous allons avoir besoin de quelques subtilités de la mémoire en C. L'essence du problème est que pour allouer un espace mémoire sur le tas, nous avons besoin de savoir *combien* de bits nous avons besoin.

Définition 1.28 : Taille d'une donnée

La **taille d'une donnée** est la quantité de mémoire que la donnée occupe.

L'unité est souvent le bit, ou l'octet (8 bits). En pratique, nous n'aurons pas besoin de nous poser la question car nous utiliserons une fonction en C qui permet d'obtenir directement la taille des données d'un type donné de manière cohérente avec les fonctions d'allocation.

Exemple 1.18 : Obtenir la taille d'une donnée en C

En C, la fonction sizeof donne la taille en nombre d'octet du type passé en argument.
Par exemple :

```
sizeof(int8) (* 1 *)
```

Cela nous permet de connaître le nombre d'octet utilisé par un entier par défaut. Cette valeur peut dépendre du support, mais en pratique, il est souvent de 4.

```
sizeof(int) (* 4 *)
```

De sorte à pouvoir conserver un typage cohérent, nous avons besoin d'inquer au compilateur le type de l'espace réservé.

Définition 1.29 : Conversion

Une **conversion de type** est la transformation d'une donnée d'un type pour qu'elle devienne une donnée d'un autre type.

Exemple 1.19 : Conversion en C

Pour convertir en C, on utilise le type entre parenthèse devant l'objet à convertir :

```
int a = -1;
unsigned int b;
b = (unsigned int) a;
```

Nous pouvons désormais utiliser les fonctions de la bibliothèque standard stdlib.h pour manipuler la mémoire.

Exemple 1.20 : Fonctions malloc et free en C

La fonction malloc prend en argument une taille, et renvoie soit un pointeur vers le début d'une zone de la mémoire du tas qui devient réservée au moment de l'appel, soit le pointeur NULL si l'allocation est un échec.
La fonction free prend en argument un pointeur vers le début d'une zone réservée du tas et la libère.

Exemple 1.21 : Utilisation de malloc et free

```c
#include <stdlib.h>

int main(){
    int * ptr;

    ptr = (int *) (malloc(sizeof(int)));
    if (ptr==NULL){
        return 1;
    }
    /* ... */

    free(ptr);
    return 0;
}
```

Pour utiliser le tas en C, nous avons besoin de plusieurs étapes :

1. Dans un premier temps, il est nécessaire de calculer la taille de l'espace qui nous intéresse grâce à sizeof.
2. Nous devons ensuite appeler malloc pour pouvoir récupérer l'adresse donnée par le système.
3. Cette adresse est de type void * qui est le type d'un pointeur vers un type non spécifié, il faut donc le convertir vers le type qui nous intéresse.
4. Une bonne pratique est de vérifier que notre allocation n'a pas échouée en vérifiant que le résultat n'est pas le pointeur NULL. Cette étape est particulièrement importante dans les système embarqués où la capacité mémoire peut être un souci.
5. Nous pouvons alors utiliser notre espace mémoire grâce à ce pointeur.
6. Finalement, après utilisation, nous pouvons libérer cet espace.

Exemple 1.22 : Exemple d'utilisation

```c
int *allouer_entier(){
    int * ptr;

    ptr = (int *) (malloc(sizeof(int)));
    if (ptr==NULL){
        abort();
    }
    return ptr
}
void incrementer_entier(int * ptr){
    *ptr++;
}
void liberer_entier(int * ptr){
    free ptr;
}
```

Lors de l'allocation sur le tas, il est possible que la donnée ne soit pas égale à 0. Les données doivent donc être initialisées.

Allouer des entiers sur le tas n'est pas nécessairement très intéressant. De sorte à pouvoir utiliser le tas à son plein potentiel, nous allons avoir besoin de manipuler des structures de données plus intéressantes dans le C.

4 Structures de données et types construits en C

Définition 1.30 : Structure de donnée

Une **structure de donnée** est une manière d'organiser les données pour les traiter plus facilement.

4.1 Tableaux

Définition 1.31 : Tableau

Un **tableau** de données est une structure de données linéaire à taille fixe dans laquelle il est rapide d'accéder à un élément quelconque du tableau.

Exemple 1.23 : Tableaux en C

Pour définir un tableau, on doit en préciser la taille et le type des éléments à l'aide de la syntaxe suivante :

```
int tableau[10];
```

La taille entre crochet doit être un littéral (c'est-à-dire un nombre en toute lettre) ou une constante.
On peut ensuite accéder à un élément du tableau à l'aide de l'expression suivante :

```
tableau[k]
```

Enfin, on peut modifier un élément du tableau à l'aide de l'instruction suivante :

```
tableau[k] = nouveau;
```

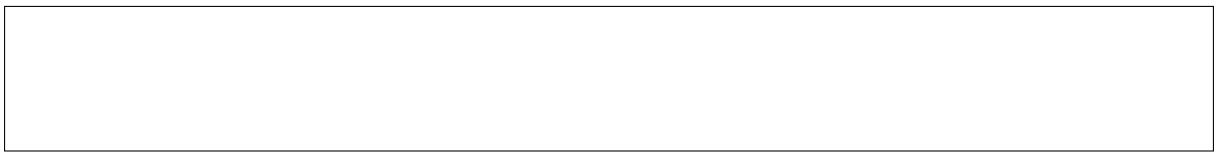
La contrainte de staticité de la taille du tableau, c'est-à-dire du fait que l'on peut déterminer la taille du tableau à la compilation, est indispensable.

Nous pouvons initialiser un tableau à l'aide de la syntaxe suivante :

Définition 1.32 : *Pointer Decay*

Quand nous manipulons le tableau en tant que valeur (par exemple en affectant le tableau à une variable, ou surtout en l'utilisant comme argument d'une fonction), celui-ci est transformé en pointeur vers son premier élément.
On appelle ce procédé ***Pointer Decay*** en anglais (que l'on pourrait traduire par *dégradation en pointeur*).
Les deux exceptions notables sont l'opérateur `&` et la fonction `sizeof`

En C, les tableaux sont des espaces contigus en mémoire, et il est donc facile d'accéder rapidement à une donnée grâce à l'adresse du premier élément : il suffit de multiplier la taille d'un élément par l'indice et d'ajouter le total à l'indice de départ.
On peut donc utiliser sans souci la syntaxe `t[k]` même avec un pointeur.



Le souci de la dégradation en pointeur est que l'on perd la taille du tableau, et que le compilateur ne fait donc plus les vérifications de taille :

```c
#include <stdio.h>

int main(){
    int t[10];
    printf("%d\n", t[10]); // Le compilateur accepte, mais le resultat est faux.
    return 0;
}
```

Il faut donc rajouter une taille dans les fonctions qui travaillent sur des tableaux.

```c
void afficher_tout(int * t, int taille){
    for(int i = 0; i<taille; i++)
    {
        printf("%d\n", t[i]);
    }
}
```

Pour l'instant, nous avons alloué des tableaux de taille *fixe*. Ceux-ci sont alloués sur la pile en tant que variables locales.

Nous pouvons être intéressés pour allouer des tableaux sur la pile à l'aide de malloc, d'une part pour avoir des tableaux dont la taille n'est pas connue à l'avance, et d'autres part pour pouvoir conserver ces données à travers les appels de fonctions.

Exemple 1.24 : Allocation d'un tableau sur le tas

Pour allouer un tableau sur le tas de taille n, nous devons allouer une taille n multiplié par la taille d'un élément.

```c
int *t = (int*) malloc(n * sizeof(int));
```

Ici, ce tableau est directement un pointeur, et nous devons donc le manipuler comme un pointeur.

Exercice 1.12. Proposer une fonction de prototype `int * recopier(int * t, int n)` qui, à partir d'un tableau (et donc d'un pointeur vers son premier élément) ainsi que sa taille, alloue un tableau de même taille sur le tas et recopie les valeurs du premier tableaux dans le second.

Dans le cas d'un tableau que l'on veut initialiser vide, on peut utiliser une fonction spécifique.

Exemple 1.25 : Fonction calloc

La fonction calloc prend en argument le nombre d'éléments à allouer, et la taille d'un élément, et renvoie un pointeur vers le premier élément d'une zone dans la mémoire initialisée à zéro réservée pour ces éléments.
Pour allouer un tableau de taille n d'entiers, il faut donc utiliser :

```c
(int*) calloc(n, sizeof(int))
```

Le souci de calloc est que la remise à zéro des valeur a un coût. Si le tableau doit être initialisé dans tous les cas, ou si les valeurs qu'il contient ne sont pas importantes, on préférera utiliser malloc.

Il s'avère qu'il est souvent *autorisé* par le compilateur d'allouer des tableaux à taille variable sur la pile.

```c
void f(int n){
    int t[n];
}
```

Cependant, cette affectation pose des soucis de performances, et ne respecte pas les standards. En particulier, il est précisé dans le programme que cette allocation est fautive.

4.2 Types produits

Un **type produit** de plusieurs types t_1, t_2, ..., t_n est un type dont chaque élément dispose de n composantes de types respectifs t_1, t_2, ..., t_n.

Exemple 1.26 : Type structuré en C

Une **type structuré** en C est une implémentation d'un type produit. Un tel type est composé de **champs** qui ont chacun un type.
Nous pouvons en définir un avec la syntaxe suivante :

```
struct nom_de_la_struct {
            type1 champs1 ;
            type2 champs2 ;
            ...
            typen champsn;
        };
```

On peut créer un élément de ce type classiquement en le déclarant :

```
struct nom_de_la_struct variable
```

Par la suite, on peut accéder à un champs de ce type à l'aide de la syntaxe :

```
variable.champs1
```

```
variable.champs1 = ...;
```

Exemple 1.27 : Un type structuré

```
struct couple  {
    int x;
    int y;
};
```

On peut simplifier un peu les déclarations de type à l'aide du mot-clef typedef. Cela nous permet de simplifier le code.

```
typedef struct couple couple_t;
```

On peut ensuite se servir du nouveau nom couple_t :

```
int projection_x(couple_t c){
    return c.x;
}
```

De manière générale, typedef permet de définir des alias pour des types.

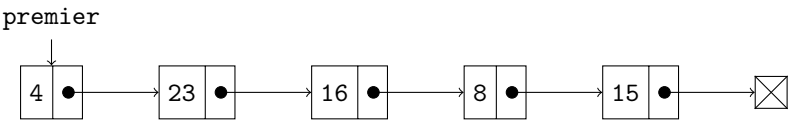
4.3 Listes chaînées

Une application importante des types structurés et des pointeurs sont les *listes chaînées*.

Définition 1.33 : Liste Chaînée

Une **liste chaînée** est une structure de données linéaire dont la représentation mémoire est une succession de **cellules** faites d'un contenu et d'un pointeur vers la cellule suivante si elle existe.

Chaque cellule permet d'avoir accès à la suivante. La liste est donc un pointeur vers le début à partir duquel nous pourrons parcourir la liste.



Pour définir une liste chaînée en C, nous avons besoin d'un type structuré de cellule :

```c
typedef struct _cell {
    int val ;
    struct _cell *suiv;
} cellule;
```

À partir de là, nous pouvons définir un type de liste chainée qui est un pointeur vers

```c
typedef cellule *chainee ;
```

On peut donc parcourir la liste chaînée en commençant par l'élément pointé par le pointeur, puis utiliser le champs suiv pour accéder à la cellule suivante à partir de chaque cellule.

Il subsiste les questions suivantes :

— Comment représenter une liste chaînée vide ?

— Quand s'arrête le parcours des éléments ?

La solution repose sur l'utilisantion du pointeur NULL.

Ansis, pour représenter une cellule qui n'a pas de suivant, on fera que le pointeurs suiv aura la valeur NULL, tandis que si on veut représenter la liste vide, on utilisera le pointeur NULL pour la valeur du premier élément.

Exemple 1.28 : Calculer la longueur d'une liste chaînée

```c
int longueur(chainee lc){
    int reponse = 0;
    cellule * actuel;
    actuel = lc;
    while (actuel != NULL){
        reponse++;
        actuel = (*actuel).suiv;
    }
    return reponse;
}
```

Proposition 1.6 : Temps d'accès dans une liste chaînée

Pour accéder à l'élément d'indice k, c'est-à-dire le $(k+1)$-ième élément en prenant les éléments dans l'ordre, il faut parcourir $k+1$ cellules.

C'est la principale limitation de la liste chaînée, pour atteindre à élément, même si on en connaît l'indice, il faut parcourir les éléments qui se trouvent avant.

Exercice 1.13. Proposer une fonction de prototype `int indice(chainee lc, int n)` qui renvoie l'indice de l'élément de valeur `n` dans la liste chaînée `lc`.

Il s'avère qu'il est assez récurent d'avoir besoin d'accéder à un champs dans un pointeur vers une structure. C nous donne un sucre syntaxique bien pratique qui nous permet de réaliser cette opération rapidement.

Ainsi le code

```c
(*c_ptr).suiv
```

peut s'écrire

```c
c_ptr->suiv
```

De manière générale, cela peut simplifier les codes, et il faut pouvoir lire du code qui utilise ce sucre syntaxique.

Si on veut modifier la liste chaînée, il va falloir modifier les différents pointeurs :

Exemple 1.29 : Ajouter une cellule en tête d'une liste chaînée

```c
void ajouter_debut(chainee *lc, int valeur){
    cellule *nouvelle;
    nouvelle = (cellule *) (malloc(sizeof(cellule)));
    if (nouvelle == NULL){
        abort();
    }
    nouvelle->suiv = *lc;
    nouvelle->val = valeur;
    *lc = nouvelle;
}
```

On remarque que le premier argument de notre fonction est un pointeur vers une liste chaînée, et donc un pointeur vers un pointeur vers une cellule.

Exercice 1.14. Comment retirer une cellule à la fin d'une liste chaînée ? La fonction aura le prototype :

```c
void retirer_fin(chainee * lc)
```

5 Autres types en C

5.1 Flottants

Proposition 1.7 : Écriture d'un réel en base deux

Pour tout réel positif x, il existe N, ainsi qu'un $N+1$-uplet a_N, a_{N-1}, ..., a_0, et une suite $(d_k)_{k\geqslant 1}$ tels que :

$$\forall 0 \leqslant k \leqslant N,\ a_k \in \{0,1\}$$
$$\forall k \in \mathbb{N},\ d_k \in \{0,1\}$$
$$x = \sum_{k=0}^{N} a_k 2^k + \sum_{k=1}^{+\infty} d_k 2^{-k}$$

On dit que $\sum_{k=1}^{+\infty} d_k 2^{-k}$ est le **développement décimale en base 2 de** x.

Ce résultat est un peu délicat à prouver (il s'agit du calcul d'une somme infinie et porte sur la définition des réels), et contient quelques subtilités ennuyeuse (par exemple cette représentation n'est pas unique), mais l'idée principale est qu'un nombre réel peut toujours être représenté, quitte à avoir un nombre avec un développement infini, en base deux.

Comme il existe une suite de d_k qui permette d'approcher de plus en plus le nombre x, nous pouvons nous intéresser à la méthode pour les trouver. Pour tout réel, cela peut donc nous donner une approximation de celui-ci.

L'idée principale est l'inverse de la décomposition entière en base : nous avions fait des divisions successives pour trouver les a_k, maintenant, nous devons faire des multiplications successives pour trouver les d_k.

Algorithme 1.3 : Développement décimal en binaire d'une partie décimale

Entrée : $0 \leqslant x \leqslant 1$ un nombre à représenter, N une précision voulue
Sortie : d_1, ..., d_N les chiffres du développement décimale de x.

```
resultat ← liste vide
Pour Chaque k de 1 à N Faire
    x ← x × 2
    d_k ← la partie entière de x
    x ← la partie décimale de x
    ajouter d_k à la fin de resultat
Renvoyer resultat
```

Exercice 1.15. Quelle est le développement décimale de 0.375 ? De $\frac{1}{3}$?

Cependant, on ne peut pas représenter tous les réels avec des structures discrètes finies, et nous avons donc besoin de nous intéresser à un sous ensemble des réels.

L'idée, quand on veut représenter un réel est de prendre une valeur approchée, à une précision donnée. Une première idée serait d'utiliser une précision fixe, c'est-à-dire un nombre de d_k donné, mais le souci est que l'on ne pourra représenter que des ordres de grandeurs relativement restreints.

La solution trouvée est d'utiliser des nombres à virgule flottante.

Définition 1.34 : Nombre à Virgule Flottante

Un **nombre à virgule flottante**, ou souvent un **nombre flottant** et parfois un **flottant**, est un nombre x pour lequel il existe trois nombres :

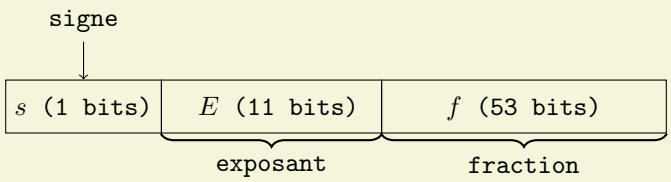
- son signe s égal à -1 ou 1 ;
- sa mantisse $m \in \mathbb{N}$;
- son exposant $e \in \mathbb{Z}$

Et ce, avec :

$$x = s \times m \times 2^e$$

L'intérêt principal des nombres à virgule flottante est de pouvoir représenter de nombreux ordres de grandeurs avec une même donnée.

Exemple 1.30 : Valeur d'un flottant sur 64 bits dans la norme IEEE 754



E est un entier positif sur 11 bits qui correspond à l'exposant, f est la fraction sur 53 bits, et s est le signe du flottant, sur 1 bit.
Si $1 \leqslant E < 2^{11} - 1$, le nombre est dit normal et sa valeur est :

$$(-1)^s \times 2^{E-1023} \times 1.f$$

Si $E = 0$, le nombre est dit subnormal et sa valeur est

$$(-1)^s \times 2^{1-1023} \times 0.f$$

Proposition 1.8 : Valeurs extrêmes

Le plus grand nombre que l'on peut représenter avec un tel flottant est :

$$2^{2^{11}-2-1023} 1.1.....1 \approx 2^{1023}$$

Le plus petit nombre strictement positif que l'on peut représenter avec un tel flottant est :

$$2^{-1074}$$

Cette gamme de valeurs possibles est particulièrement pratique en physique numérique ou en apprentissage statistique.

Il existe par ailleurs des valeurs spéciales pour $E = 2^{11} - 1$ qui permettent notamment de représenter les différents infinis, et le fameux `nan` qui correspond à *Not A Number*.

Exemple 1.31 : Les flottants en C

On dispose des types `float` et `double` qui permettent de manipuler des flottants en C.
Le premier de ces types est un flottant sur 32 bits, tandis que le second est un flottant sur 64 bits.

Il existe cependant quelques soucis avec les flottants que nous aurons l'occasion de voir en exercices.

5.2 Chaînes de caractères

Définition 1.35 : Caractère

Le **caractère** est un type de donnée qui permet de représenter une lettre, un chiffre, un symbole, un blanc (espaces, tabulations, retours à la ligne...), ou une opération spéciale de contrôle.

Il existe de nombreuses manière élaborées d'encoder des caractères variées, mais en C, nous n'avons pas cette marge de manœuvre.

Exemple 1.32 : Caractères en C

En C, le type `char` encode un caractère sur un entier sur 8 bits en ASCII [a] .
Les caractères sont représentés par des guillemets simples.

a. https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

Cela veut dire notamment que nous ne pourrons pas représenter de caractères accentuées. Cela n'est pas très pratique pour ce qui est de l'affichage, mais beaucoup plus pratique pour ce qui est de la manipulation mémoire : nous pouvons manipuler des caractères comme des entiers.
Il s'avère que la plupart des compilateurs nous autorise à utiliser de l'unicode, mais le comportement des fonctions de la bibliothèques standard peut être inattendu.

Par exemple, on remarque qu'il y a toujours un décalage de 32 entre une lettre en majuscule (`'A'` avec 65 par exemple, et `'a'` avec 97), et on peut donc passer d'un caractère minuscule à majuscule en retirant 32.

Les caractères peuvent se regrouper en *chaînes de caractères* (*string* en anglais).

Définition 1.36 : Chaîne de caractère

La **chaîne de caractère** est un type qui correspond à une suite ordonnée de caractères.

Exemple 1.33 : Chaîne de caractères en C

En C, une chaîne de caractère est un tableau de caractères de sentinelle nulle, c'est-à-dire un tableau de caractères dont le dernier élément est un caractère spécial de valeur '\0'.
Les chaînes peuvent être définie à l'aide de guillemets droits doubles :

```
char chaine[] = "Bonjour !"
```

Cela est équivalent à initialiser le tableau (en n'oubliant pas la sentinelle) :

```
char chaine[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', ' ', '!', '\0'}
```

Ainsi, on peut représenter la chaîne "Bonjour par le tableau :

'B'	'o'	'n'	'j'	'o'	'u'	'r'	' '	'!'	'\0'

Les chaînes ne se comportent donc pas exactement comme des tableaux car on a pu réserver un caractère spécial pour marquer le dernier élément.

Exemple 1.34 : Calcul de la longueur d'une chaîne de caractère

On nomme **longueur** d'une chaîne de caractère le nombre d'éléments dans cette chaîne de caractère. Par convention, en C, le caractère '\0' ne compte pas dans la longueur de la chaîne de caractère.
Pour trouver la longueur, on peut parcourir les éléments du tableau jusqu'à trouver le caractère nul.

```
int longueur(char * chaine){
    int indice = 0;
    while (chaine[indice]!='\0'){
        indice++;
    }
    return indice;
}
```

En tant que tableau, les chaînes de caractères sont mutables en C. Sur beaucoup de fonctions sur les chaînes de caractères, on préfère avoir l'assurance que la chaîne de caractère ne peut pas être modifiée. Pour s'en assurer, on peut indiquer `const char *` au lieu de `char *` pour les chaînes qu'on ne veut pas pouvoir modifier.
Ainsi, on pourrait changer le prototype de longueur pour qu'il soit `int longueur(const char * chaine)`.

Exercice 1.16. Proposer une fonction de prototype `int compter(const char * chaine, char c)` qui renvoie le nombre d'occurence de `c` dans `chaine`.

Les chaînes de caractère sont particulièrement utiles pour pouvoir réaliser un affichage. Le C nous donne par ailleurs une fonction particulière pour formater une chaîne de caractère de sorte à pouvoir afficher des valeurs d'autres types à l'intérieur.

Exemple 1.35 : Fonction printf

La fonction `printf` du module `stdio.h`permet d'afficher une chaîne de caractère sur la sortie standard.

```
printf("Bonjour !");
```

On peut se servir de cette fonction pour afficher des entiers à l'aide de %d dans la chaîne de caractère et un entier en arguemnt :

```
int a = 2;
printf("L'entier a vaut %d", a);
```

On peut rajouter des arguments et utiliser des types différents à l'aide d'autres lettres après le symbole % et d'autres arguments :

```
printf("Voici un entier %d et voici un flottant %f", 2, 1.0);
```

(Les suites de caractères %... sont remplacés dans l'ordre par les arguments en plus du premier dans l'ordre)

On dispose d'un tableau qui fait correspondre une lettre à un type pour l'affichage :

Caractères	%d, %i	%u	%f, %F	%c	%s
Type	Entier signé	Entier non signé	Flottant de type double	Caractère	Chaîne de caractère

Exercice 1.17. Proposer une fonction de prototype `void afficher_espace(const char * chaine)` qui affiche les caractères de `chaine` en les séparant par des espaces.

Exemple 1.36 : Fonction scanf

La fonction `scanf` en C permet de récupérer des données depuis l'entrée standard, le plus souvent depuis une console.
Il faut lui spécifier un format attendu et des adresses pour les différentes valeurs attendus. Le format est similaire à celui de `printf`.

```
int a;
scanf("%d", &a);
```

Les fonction `printf` et `scanf` appartiennent à l'ensemble des fonctions qui permettent de gérer les entrées et les sorties d'un programme.

6 Fichiers ; Entrées et sorties d'un programme

De sorte à pouvoir nous intéresser aux différentes manière d'utiliser des entrées et des sorties dans un programme, nous devons nous intéresser à la structure

6.1 Organisation des fichiers dans le système

Dans un système d'exploitation, les fichiers sont organisés en arborescence.

On peut se déplacer dans les dossier à l'aide d'un invité de commande. On dispose de nombreuses commandes qui permettent de faire tout ce qu'on pourrait faire avec une interface graphique.

Exemple 1.37 : Quelques commandes utiles en shell UNIX

- `ls` permet d'afficher les répertoires et fichiers dans le répertoire courant ;
- `cd nom` permet de se rendre dans le répertoire `nom` ;
- `cd ..` permet de remonter au répertoire parent ;
- `mkdir nom` crée un répertoire `nom`.

Définition 1.37 : Racine

La **racine** est le répertoire dont tous les fichiers sont un descendant.

Sous unix, la racine est `/`. Sous windows, il n'existe pas de racine (chaque disque dispose de sa propre racine, mais il n'existe pas de racine commune à tous les disques).

Définition 1.38 : Chemin relatif, chemin absolu

Un **chemin absolu** est un chemin vers un fichier ou un répertoire depuis la racine du système.
Un **chemin relatif** est un chemin vers un fichier ou un répertoire depuis le répertoire courant.

Il n'est pas absolument nécessaire d'avoir une maîtrise approfondi des différentes subtilités de l'utilisation de la console, mais quelques concepts peuvent être utiles.

Ce qui nous intéresse le plus, est le fait qu'il est possible d'exécuter un programme avec la syntaxe suivante :

```
./nom_du_programme
```

.

On peut rajouter par ailleurs des arguments dans l'exécution du programme :

```
./nom_du_programme arg1 arg2 arg3
```

Nous pouvons nous intéresser à comment utiliser ces arguments.

6.2 Entrées en utilisant les arguments d'un programme

Pour l'instant, le prototype de la fonction `main` a été le suivant :

```
int main ()
```

Cependant, il est possible de modifier cette fonction main de sorte à pouvoir prendre en compte les arguments données au programme lors de son appel.

Exemple 1.38 : Arguments de la fonction main en C

On peut écrire la fonction `main` avec le prototype suivant :

```
int main(int argc, char *argv[])
```

`agrc` correspond au nombre d'arguments utilisés lors de l'appel du programme ;
`argv` est un tableau de chaînes de caractères dont dont chaque élément est un argument du programme.
Le nom du programme lui-même est le premier argument. Il y a donc toujours au moins une chaîne dans `agrv` et `agrc` est toujours au moins égal à 1.

Exemple 1.39 : Un programme somme

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    int resultat = 0;
    for (int i = 1; i<argc; i++){
        resultat = resultat + atoi(argv[i]);
    }
    printf("%d", resultat);
    return 0;
}
```

Exercice 1.18. Proposer un programme qui prend en argument des chaînes de caractères, et renvoie sur la sortie standard leurs tailles séparés par des espaces.

6.3 Fichiers vus par le processus

Un programme peut faire un appel au système d'exploitation pour lui demander l'accès à un fichier. Par l'intermédiaire de fonctions systèmes, un programme peut manipuler des fichiers qu'il a ouvert : pour chaque fichier ouvert, il dispose d'un mode d'ouverture (écriture, lecture, mode binaire, ...) et d'un curseur, c'est-à-dire l'endroit dans le fichier où il en est dans la lecture ou l'écriture.

En C, les fichiers sont manipulés comme des pointeurs vers un type FILE défini dans stdio.h. On peut ouvrir ce fichiers à l'aide de la fonction fopen, et écrire dans ce fichier à l'aide de fprintf.

Il est par la suite nécessaire de fermer ce fichier à l'aide de la fonction fclose.

Exemple 1.40 : Écriture dans un fichier en C

```c
FILE * fptr;
fptr = fopen("nom_du_fichier", "w");
fprintf(fptr, "Bonjour !\n");
fprintf(fptr, "Bonjour sur une nouvelle ligne !");
fclose(fptr);
```

Ici, l'option w indique que l'on désire écrire (Write en anglais) dans ce fichier. Si le fichier existe déjà, son contenu sera effacé pour laisser place à notre texte.
On peut utiliser l'option a (Append en anglais) pour ajouter à la fin d'un fichier au lieu de le remplacer.

Pour lire dans un fichier, on doit utiliser une mémoire tampon dans laquelle on va lire morceau par morceau le fichier.

Exemple 1.41 : Lecture dans un fichier en C

```
FILE * fptr;
char tampon[64];
fptr = fopen("nom_du_fichier", "r");
fscanf(fptr, "%s", tampon);
printf("Voici la premiere chaine que nous pouvons extraire du fichier : %s",
    tampon);
fclose(fptr);
```

L'utilisation de %d ne permet pas de récupérer l'intégralité du fichier, en effet si un blanc (espace, saut de ligne, ...) est rencontré lors de la lecture du fichier, on arrête la lecture. Il faudra utiliser d'autres méthodes pour lire l'intégralité du fichier.

Par ailleurs, l'utilisantion de fscanf (ou de scanf) avec des données de taille variable dans une mémoire tampon de taille fixe peut entraîner des soucis s'il n'y a pas suffisament de place.

On évitera donc cette méthode quand c'est possible, par exemple avec fgets.

Il existe certains fichiers spéciaux qui correspondent aux flux habituels que nous utilisons classiquement dans un programme.

Définition 1.39 : Des fichiers spéciaux

Le **flux de sortie standard** correspond aux données sortantes du programme qui sont souvent dirigées vers un terminal.
Le **flux d'entrée standard** correspond aux données entrantes du programme qui proviennent souvent des entrées fournies dans un terminal.
Le **flux d'erreur standard** correspond aux sorties liées aux erreurs qui sont souvent envoyés dans un terminal.

Ainsi, les fonctions scanf et printf font implicitement usage de ces fichiers spéciaux pour interragir avec la console.

Il est cependant possible de *rediriger* une sortie vers ou depuis un fichier.

Exemple 1.42 : Redirection

On peut rediriger l'entrée standard depuis un fichier à l'aide du symbole <.
On peut rediriger la sortie standard vers un fichier à l'aide des symboles > et >>.
On peut rediriger la sortie d'erreur standard vers un fichier à l'aides des symboles 2> et 2>>.

La distinction entre > et >> est que > efface l'ancien contenu du fichier cible avant de le remplacer avec la nouvelle sortie, tandis que >> ajoute en fin de fichier la nouvelle sortie.

Exemple 1.43 : Exemple de redirection

La commande suivante exécute le programme prog et envoie sa sortie dans un fichier sortie.out :

```
./prog > sortie.out
```

Si le fichier existait déjà, son contenu est remplacé par la sortie de prog, sinon, il est crée et son contenu devient la sortie de prog.

On peut enchaîner les commandes à l'aide de *tubes*.

Définition 1.40 : Tube

Un **tube** est un mécanisme de communication inter-processus dont les données sont organisés sous la forme d'une file.
Un tube redirige la sortie standard d'un premier processus vers le l'entrée standard d'un deuxième processus.

Exemple 1.44 : Tubes en shell UNIX

On peut utiliser le symbole | pour créer un tube entre deux programmes invoqués par des commandes.

```
history | grep cd
```

6.4 Implémentation des fichiers

Il est une dernière notion au programme sur les fichiers : la notion de leur implémentation en pratique. L'idée est que les fichiers de notre arborescence sont représentés sur le disque par un nœud d'index.

Définition 1.41 : Nœud d'index

Un **Nœud d'index** (ou inode) est une structure de données contenant des informations à propos d'un fichier.
L'inode contient des informations sur le contenu du fichier, ainsi que sur des métadonnées sur le fichier (droits d'accès, propriétaire, ...)

La structure exacte d'un nœud d'index dépend du système de fichier, mais on peut s'intéresser à un exemple simple : un vieux système de fichiers de Linux.

Exemple 1.45 : Norme EXT2

EXT2 est un système de fichier historique de Linux. Il a cédé sa place à d'autres variantes (EXT3, EXT4).
L'idée derrière l'organisation des fichiers en EXT2 est d'utiliser l'inode pour sauvegarder des informations spécifiques aux caractéristiques du fichier, ainsi que des pointeurs vers des blocs de données directement, ou indirectement.

Chapitre II

Structures hiérarchiques : Les Arbres

1 Arbres et vocabulaires des arbres

On commence par une construction intuitive des arbres et des différentes manière de les décrire. Nous préciserons ces notions dans un deuxième temps après avoir réalisé une définition plus formelle des arbres.

1.1 Arbres

Définition 2.1 : Arbre

Un **arbre** est une structure de donnée hiérarchique finie qui se ramifie depuis une origine.

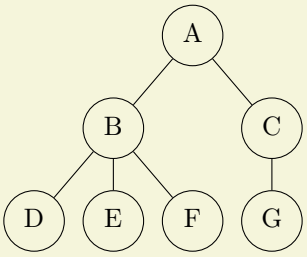
Nous avons fait plusieurs choix dans cette définition d'arbre.

Le premier est que la structure est finie, ce qui est cohérent avec l'objectif de traiter cet arbre d'un point de vue informatique.

Le second est d'organiser de manière hiérarchique un arbre en donnant un « sens » à la généalogie de l'arbre.

Exemple 2.1 : Exemple d'arbre

En informatique, on représente les arbres généralement dirigés vers le bas.



Exemple 2.2 : Utilisation des arbres

Les arbres ont de nombreuses applications directes :

- Appels récursifs d'une fonction avec plusieurs appels récursifs ;
- Représentation d'arbres de décision, de possibilités ;

Définition 2.2 : Nœud

Un **nœud** est un élément d'un arbre.

Définition 2.3 : Étiquette

Une **étiquette** est une information contenue dans un nœud, ou un identifiant pour ce nœud.
On limite parfois les étiquettes à un sous-ensemble des nœuds d'un arbre.

Quand nous nous intéresserons à la forme des arbres, nous donnerons une étiquette unique à chaque nœud de sorte à pouvoir les nommer, mais la plupart du temps, les étiquettes ne seront pas nécessairement uniques.
Quand les étiquettes sont uniques, on parle parfois de *clef*.

Ce qui est important dans notre définition d'un arbre est la notion de hiérarchie. Un nœud peut disposer d'enfants ou d'un parent.

Définition 2.4 : Hiérarchie dans un arbre

Le **parent** ou **père** d'un nœud est le nœud qui est juste avant dans l'arbre.
Un **enfant** ou **fils** d'un nœud est un nœud qui est juste après dans l'arbre.

Définition 2.5 : Racine

La **racine** d'un arbre est un nœud qui n'a pas de parent.

Définition 2.6 : Feuille

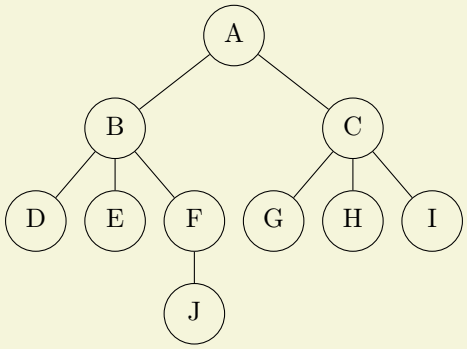
Une **feuille** (ou parfois **nœud externe**) est un nœud qui n'a pas d'enfants.

Définition 2.7 : Nœud interne

Un **nœud interne** est un nœud qui a au moins un enfant.

Exemple 2.3

La racine de cet arbre est le nœud A. B a trois enfants qui sont D, E, et F. Le parent de G est C. Les feuilles sont D, E, G, H, I et J. Les nœuds internes sont A, B, F et C.



Définition 2.8 : Profondeur d'un nœud

La **profondeur** d'un nœud est la distance à la racine du nœud, c'est-à-dire le nombre de saut qu'il faut faire depuis la racine pour atteindre le nœud.

Ainsi, la racine est à profondeur 0, et ses enfants sont à profondeur 1.

Définition 2.9 : Hauteur d'un arbre

La **hauteur** d'un arbre (ou parfois **profondeur**) est le maximum de profondeur de ses nœud.

Exercice 2.1. Quelles sont les profondeurs des nœuds de l'arbre de l'exemple précédent ? Quelle est la hauteur de l'arbre ?

Définition 2.10 : Niveau d'un arbre

Un **niveau** d'un arbre est l'ensemble des nœuds à une profondeur donnée.

1.2 Arbres binaires

Il s'avère que bien souvent, on s'intéresse à des arbres avec un nombre limité d'enfant.

Définition 2.11 : Arité dans un noeud

L' **arité** d'un nœud est le nombre de ses enfants.

Définition 2.12 : Arité d'un arbre

Un **arbre d'arité** n est un arbre dont tous les nœuds sont d'arité au plus n.

Définition 2.13 : Arbre binaire

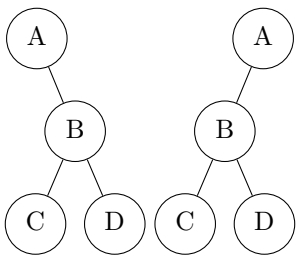
Un arbre binaire est un arbre d'arité 2.

On peut donc distinguer les enfants d'un nœuds en deux catégorie : le fils droit et le fils gauche.

Définition 2.14 : Fils gauche, fils droit

Chaque enfant d'un nœud d'un arbre binaire est un **fils gauche** ou un **fils droit**. Il y a au plus un de chaque pour chaque nœud.

Cela sous-entends que les deux arbres suivants sont différents :



En effet, dans le premier cas, B est le fils droit de A, tandis que dans le second cas, B en est le fils gauche.

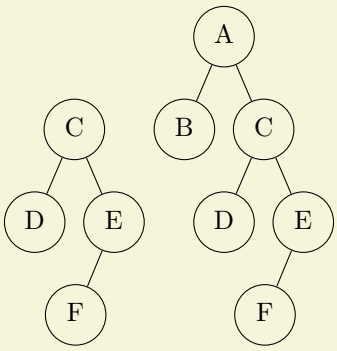
1.3 Sous-arbre

Définition 2.15 : Sous-arbre

Un **sous-arbre** a' d'un arbre a est un arbre dont la racine est un nœud n de a, et qui contient tous les descendant de n dans a.

Exemple 2.4 : Exemple de sous-arbre

L'arbre de gauche est un sous-arbre de l'arbre de droite :



Exercice 2.2. Quels sont les sous arbres de l'arbre de racine A ?

Parfois, dans le cas des arbres binaires, on parle de *sous-arbre droit* ou de *sous-arbre gauche* pour parler de l'arbre dont la racine est l'enfant droit (ou gauche) et qui contient tous les descendants de ce nœud.

Par ailleus, on a ici imposé que le sous-arbre devait contenir tous les descendants de sa racine. Il s'avère qu'en théorie des graphes, la définition est un peu différente et il faudra faire attention quand nous parlerons d'arbre au sens de la théorie des graphes lorsque nous perdons l'aspect hiérarchique des arbres.

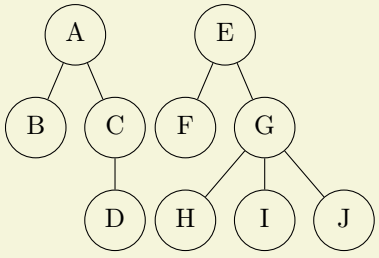
1.4 Forêt

Définition 2.16 : Forêt

Une **forêt** est une juxtaposition finie d'arbres.

Exemple 2.5 : Exemple de forêts

— Ceci est une forêt :



— La forêt vide est une forêt ;
— Un arbre est une forêt ;
— La juxtaposition de deux forêts est une forêt.

Il est possible de définir sur les forêts les différentes propriétés dont nous disposons sur les arbre.

Définition 2.17 : Taille d'une forêt

La **taille d'une forêt** est la somme des tailles des arbres qui la compose.

Définition 2.18 : Profondeur d'une forêt

La **profondeur d'une forêt** est le maximum des profondeurs des arbres qui la compose.

2 Principe d'induction structurelle

De sorte à pouvoir prouver des propriétés sur les arbres, nous généralisons le principe de récurrence avec le principe d'induction qui nous permet à la fois de créer des structures construites par induction, et de raisonner sur ces structures.

Le cadre donné est plus général que les arbres, mais c'est dans un premier temps la principal application que l'on s'en donne.

2.1 Ensemble défini de manière inductive

En informatique, on est souvent intéressé par la construction d'éléments à partir d'éléments de bases. L'idée est qu'on peut passer d'un ou plusieurs éléments à un autre élément à l'aide d'une règle de construction qui ajoute à notre ensemble.

Définition 2.19 : Définition inductive

On se donne des éléments de bases un ensemble d'éléments, et un ensemble de règles de construction qui partir d'un certain nombre d'éléments construisent un autre élément. Un ensemble **défini inductivement** (ou parfois **défini récursivement**) est un ensemble E construit comme le plus petit ensemble (au sens de l'inclusion) qui vérifie :

- E contient les éléments de bases ;
- E est stable pour toutes les règles de construction.

Il s'avère qu'il existe bel et bien un tel ensemble qui soit minimum. La démonstration repose sur le fait que quand on a plus d'éléments dans notre ensemble, alors on peut construire des élèments en plus, mais jamais des éléments en moins : en s'intéressant à l'intersection de tous les ensembles qui vérifie cette propriété, on a construit un ensemble minimal qui vérifie les propriétés.

Le souci, c'est que l'on a pas précisé dans quel plus grand ensemble on se plaçait, et on sait que l'on ne peut pas raisonner sur l'ensemble de tous les ensembles.

Volontairement, nous esquiverons cette question dans la suite. Nous donnerons un cadre plus formel.

Exemple 2.6 : Construction des entiers

On peut construire les entiers de la manière suivante :

- 0 est un entier ;
- Si n est un entier, alors $n+1$ est un entier.

En pratique, il est un peu plus compliqué de construire les entiers. Les axiomes de Peano qui permettent de construire classiquement les entiers sont un peu plus complets.

Exemple 2.7 : Construction des listes

On peut construire récursivement l'ensemble des listes d'entiers de la manière suivante :

- La liste vide `[]` est dans l'ensemble ;

— Si q est dans l'ensemble, alors pour tout $n \in \mathbb{Z}$, alors `n::q` est dans l'ensemble.

2.2 Principe d'induction structurelle

Proposition 2.1 : Principe d'induction structurelle

Soit E un ensemble défini inductivement. Soit P une propriété.
Si on a les hypothèse suivantes :

1. P est vraie pour tous les éléments de base de E ;
2. Pour chaque régle de construction, si P est vraie pour des éléments, alors, pour l'élément construit avec ces éléments et cette règle, P est vraie.

Alors, pour tout $x \in E$, $P(x)$ est vrai.

2.3 Applications aux arbres

On peut construire les arbres de la manière suivante :

Définition 2.20 : Définition inductive des arbres quelconques

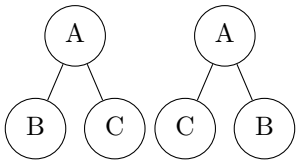
On définit les arbres étiquettés par l'ensemble E inductivement de la manière suivante :
— Les nœuds dont l'étiquette est dans E sont des arbres ;
— Pour chaque élément $e \in E$, et pour chaque ensemble fini et non vide d'arbres, on peut construire l'arbre dont la racine est un nœud étiquetté e et dont les enfants sont ces arbres.

On remarque dans un premier temps que dans notre définition précédente, on aurrait pu considérer ne pas avoir d'éléments de base, et s'autoriser un ensemble d'enfant vide.

L'intérêt ici de ne pas autoriser les ensembles vides d'enfants est de distinguer entre les éléments de bases qui sont nécessairement des feuilles, et les éléments qui ont été construits par au moins une application d'une règle de construction.

Ainsi, si on veut pouvoir n'étiquetter que les feuilles de l'arbre, on peut décider de changer les règles de construction pour qu'elles ne dépendent pas d'un élément

Dans cette définition, les enfants sont considérés comme un ensemble d'arbres, et les deux arbres suivants sont donc les mêmes :



La plupart du temps, on veut tout de même avoir un ordre sur les nœuds qui sont dans l'arbre, et on peut donc vouloir utiliser la définition suivante :

Définition 2.21 : Définition inductive des arbres quelconques avec des enfants ordonnés

On définit les arbres étiquettés par l'ensemble E inductivement de la manière suivante :
— Les nœuds dont l'étiquette est un élément de E ;
— Pour chaque élément $e \in E$, et pour chaque suite finie d'arbres a_0, a_1, ..., a_k (avec $k \geqslant 0$), on peut construire l'arbre dont la racine est un nœud étiquetté e et dont les enfants sont ces arbres dans l'ordre.

On peut définir les arbres binaires de la manière suivante :

Définition 2.22 : Définition inductive des arbres binaires

On définit les arbres binaires étiquettés par l'ensemble E inductivement de la manière suivante :
- Les nœuds dont l'étiquette est un élément de E ;
- Pour chaque élément $e \in E$, pour chaque arbre a, on peut construire l'arbre dont la racine est étiquettée par e et dont l'enfant droit est a.
- Pour chaque élément $e \in E$, pour chaque arbre a, on peut construire l'arbre dont la racine est étiquettée par e et dont l'enfant gauche est a.
- Pour chaque élément $e \in E$, pour chaque pair d'arbres a et a', on peut construire l'arbre dont la racine est étiquettée par e et dont les enfants gauche et droit sont respectivement a et a'.

Cette définition nous permet bien de construire tous les arbres binaires possibles. Par ailleurs, l'absence d'ambiguité sur la manière de construire un arbre nous permet de définir des grandeurs sur ces arbres par induction : on peut définir la grandeur sur les feuilles et de définir comment calculer la grandeur à partir de sa racine et de ses ou son enfant.

Le principe d'induction est plus général que les arbres, et nous n'aurons pas besoin de tous les enjeux du principe d'induction pour pouvoir l'utiliser pour les arbres.

3 Propriétés sur les arbres binaires, implémentation d'arbres

Maintenant que nous avons une définition plus formelle des arbres et de comment raisonner sur les arbres, nous pouvons nous permettre de définir plus formellement les différentes propriétés sur les arbres.

Les arbres qui vous nous intéresser sont principalement les arbres binaires.

Dans un premier temps, nous nous intéresserons à une implémentation en OCaml des arbres.

3.1 Implémentation des arbres binaires

On a plusieurs méthode. La première correspond à la manière dont on a définit les arbres binaires :

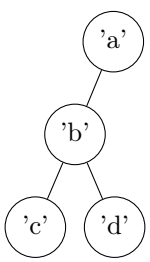
```
type 'a arbre1 = Feuille of 'a | Noeudg of 'a arbre1 | Noeudd of 'a arbre1 | Noeud2
    of 'a arbre1 * 'a arbre1
```

On a plusieurs règles de construction selon l'arbre. En réalité, il est plus simple de construire les arbres en partant d'un nœud vide et de s'en servir pour construire des feuilles :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Cela fait qu'on dispose d'un arbre en plus de la définition attendu : l'arbre vide. Nos fonctions devront donner une valeur pour l'arbre vide, mais il n'est pas nécessaire de le considérer d'un point de vue théorique.

Exercice 2.3. Comment représenter avec ce type l'arbre suivant :



Toutes les constructions que nous avons proposées partagent une propriété avec les listes OCaml : elles sont immuables. Par conséquent, il n'est pas possible de les modifier sans construire un nouvel arbre, mais l'avantage, c'est qu'il n'y aura pas de problèmes d'effets de bord.

Par exemple, si l'on veut multiplier par deux les étiquettes de tous les nœuds dans l'arbre, il faut construire un nouvel arbre.

Exemple 2.8 : Construire un arbre dont les étiquettes ont été mutlipliées par deux

```
let rec multiplier_etiquette a = match a with
| Vide -> Vide
| Noeud(k, fg, fd) ->
    let nouveau_fg = multiplier_etiquette fg in
    let nouveau_fd = multiplier_etiquette fd in
    Noeud(k*2, nouveau_fg, nouveau_fd)
```

Exercice 2.4. On considère le type arbre suivant :

```
type arbre = Vide | Noeud of bool * arbre * arbre
```

Proposer une fonction de signature `arbre -> arbre` qui retire tous les nœuds (ainsi que les descendants) dont l'étiquette est `false`.

3.2 Définitions inductives et calculs sur les arbres

Nous avons construit inductivement les arbres, et nous pouvons donc définir la plupart des descripteurs classiques à l'aide de définitions inductives.

Ces définitions se traduisent particulièrement pour des codes.

Proposition 2.2 : Hauteur d'un arbre

On peut définir la hauteur d'un arbre à l'aide de la quantité suivante :
- La hauteur d'une feuille est 0 ;
- La hauteur d'un élément construit à partir d'enfants est égale au maximum des hauteurs de ses enfants incrémentés de 1.

Cela nous donne naturellement le code suivant :

Exemple 2.9 : Calcul de la hauteur d'un arbre

```
let rec hauteur a = match a with
| Vide -> -1
| Noeud (_, gauche, droit) -> 1 + max (hauteur gauche) (hauteur droit)
```

Définition 2.23 : Taille d'un arbre

La **taille d'un arbre** est le nombre de nœud de cet arbre.

Proposition 2.3 : Taille d'un arbre

Une définition équivalente de la taille d'un arbre est la définition inductive suivante :
- La taille d'une feuille est 1 ;
- La taille d'un élément construit à partir d'enfants est égale à la somme des tailles de ses enfants incrémentées de 1.

Exemple 2.10 : Calcul de la taille d'un arbre en OCaml

```
let rec taille a = match a with
| Vide -> 0
| Noeud (_, gauche, droit) -> 1 + (taille gauche) + (taille droit)
```

Il ne faut pas confondre la *hauteur* et la *taille* d'un arbre.

Exercice 2.5. Proposer une fonction qui calcule la somme des étiquettes d'un arbre dont les étiquettes sont des entiers.

3.3 Propriétés sur les arbres

Définition 2.24 : Arbre complet

Un arbre binaire est dit **complet** si tous ses nœuds internes ont deux enfants, et que toutes ses feuilles ont la même profondeur.

Proposition 2.4 : Nombre maximum de nœuds d'une profondeur donnée

Un arbre binaire a au plus 2^l nœuds de profondeur l.
Ce maximum est atteint si tous les nœuds de niveau $k < l$ ont deux enfants.

Proposition 2.5 : Taille maximum d'un arbre binaire de profondeur donnée

Un arbre binaire de profondeur h a au plus $2^{h+1} - 1$ nœuds.
Ce maximum est atteint pour l'arbre complet de hauteur h.

Proposition 2.6 : Hauteur minimale d'un arbre binaire de taille donnée

Un arbre binaire de taille n a au moins une profondeur de $\lceil \log_2(n+1) \rceil - 1$.

Définition 2.25 : Arbre localement complet

Un arbre binaire est dit **localement complet** (ou parfois **binaire stricte**) si tous ses nœuds internes ont deux enfants.

Proposition 2.7 : Relation entre le nombre de nœuds internes et de feuilles dans un arbre localement complet

Pour tout arbre binaire localement complet avec n_i nœuds internes et n_f feuille, on a :

$$n_i + 1 = n_f$$

3.4 Implémentation en C

Le programme spécifie qu'il n'est pas nécessaire de connaître parfaitement les spécificités d'une structure hiérarchiques mutables en C. Cependant, elles peuvent être introduites pour être manipulées par la suite.

Nous allons nous servir du fait qu'un pointeur peut être nul ou non pour pouvoir implémenter notre structure d'arbre.

Exemple 2.11 : Implémentation des arbres binaire en C

```c
typedef struct _arbre {
    int valeur;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

Où les pointeurs pour les fils gauches et droits `fg` et `fd` sont égaux à `NULL` quand le nœud n'a pas le fils correspondant.

De sorte à pouvoir parcourir cet arbre, nous allons décider d'utiliser des fonctions récursives. Ce n'est pas idéale en C, mais cela est possible.

On peut décider de manipuler les arbres par une copie sur la pile de la manière suivante :

```c
int taille(arbre_t t){
    int total = 1;
    if (t.fg!=NULL){
        total += taille(*t.fg);
    }
    if (t.fd!=NULL){
        total += taille(*t.fd);
    }
    return total;
}
```

Au lieu de cela, on peut décider de manipuler un poiteur vers les arbres de la manière suivante :

```c
int taille(arbre_t * t){
    if (t==NULL){
        return 0;
    }
    return 1 + taille(t->fg) + taille(t->fd);
}
```

Une autre subtilité de cette manipulation est que la structure est mutable. Par exemple, si l'on se donne un arbre dont les deux enfants sont un même arbre, modifier un côté de l'arbre revient à modifier l'autre côté de l'arbre.

Par ailleurs, on pourra décider d'allouer les différents nœuds sur le tas de pouvoir à leur faire survivre à leurs blocs d'activations.

Maintenant que nous avons quelques calculs sur les arbres, nous pouvons nous permettre de rajouter des algorithmes et structures un peu plus élaborés sur les structures hiérarchiques.

4 Parcours d'arbre

Contrairement à une liste, il est nécessaire de faire des choix lors de l'exploration d'arbre. L'enjeu est de savoir, selon le problème, dans quel ordre on veut visiter les nœuds.

4.1 Définition générale

Définition 2.26 : Parcours d'arbre

Un **parcours d'arbre** est un algorithme de visite et de traitement des nœuds d'un arbre.

Ici, par visite, on entend que chaque nœud sera traité exactement une fois.

L' **ordre** des nœuds obtenu à partir d'un parcours est l'ordre des nœud
La plupart du temps, dans un arbre, pour un parcours, on traverse un arbre en partant de la racine puis en explorant les enfants successifs.

4.2 Parcours d'arbres binaires : préfixe, infixe, postfixe

Pour les parcours d'arbre binaire, en fonction de l'ordre dans le traitement du nœud avec les différents appels récursifs sur les enfants, on peut obtenir différent parcours d'arbre.

Définition 2.27 : Parcours préfixe

Un **parcours d'arbre préfixe** est un parcours que l'on obtient en traitant le nœud lors de l'arrivée dans le nœud, et avant la visite des enfants.

Exemple 2.12 : Parcours préfixe en OCaml

```
let rec parcours_prefixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        print_string e;
        parcours_prefixe fg;
        parcours_prefixe fd
```

Définition 2.28 : Parcours postfixe

Un **parcours d'arbre postfixe** est un parcours que l'on obtient en traitant le nœud après la visite des enfants.

Exemple 2.13 : Parcours postfixe en OCaml

```
let rec parcours_postfixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        parcours_postfixe fg;
        parcours_postfixe fd
        print_string e;
```

Définition 2.29 : Parcours infixe

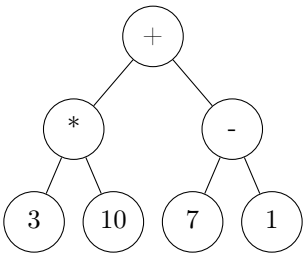
Un **parcours d'arbre infixe** est un parcours que l'on obtient en traitant le nœud entre la visite du fils gauche et le fils droit.

Exemple 2.14 : Parcours infixe en Ocaml

```
let rec parcours_infixe a = match a with
| Vide -> ()
| Noeud(e, fg, fd) ->
        parcours_infixe fg;
        print_string e;
        parcours_infixe fd
```

Naturellement, on peut définir l'ordre préfixe, postfixe et infixe à l'aide de ces parcours.

Exercice 2.6. Quels sont les ordres infixes, préfixes et suffixes pour l'arbre suivant :



4.3 Parcours d'arbre par niveau

Implicitement, dans le parcours d'arbre jusque là, on a utilisé la pile des appels. Cependant, ce n'est pas la seule structure que nous pouvons utiliser pour un parcours.

Définition 2.30 : Parcours d'arbre par niveau

Un **parcours par niveau** est un parcours d'arbre qui parcourt les arbres par profondeur croissante.

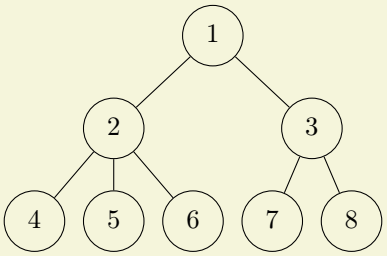
En principe, si les nœuds sont ordonnés parmi les enfants, on respectera cet ordre dans le parcours d'arbre.

Définition 2.31 : Ordre par niveau

L' **ordre par niveau** est l'ordre des nœuds obtenu par un parcours par niveau.

Exemple 2.15 : Ordre des nœuds par niveau

Dans l'arbre suivant, les nœuds sont ordonnés par ordre croissant dans l'ordre par niveau.



De sorte à pouvoir implémenter ce parcours d'arbre, nous allons utiliser une file d'attente.

Algorithme 2.1 : Parcours par niveau

$file \leftarrow$ une file qui contient la racine
Tant Que La file n'est pas vide **Faire**
    $e \leftarrow$ le premier élément que l'on prend de la file.
    Afficher e
    Ajouter tous les enfants de e à $file$.

Exercice 2.7. Implémenter un parcours d'arbre binaire par niveau en utilisant une file qui affiche les étiquettes des nœuds dans leur ordre par niveau.

On pourra utiliser le module `Queue`.

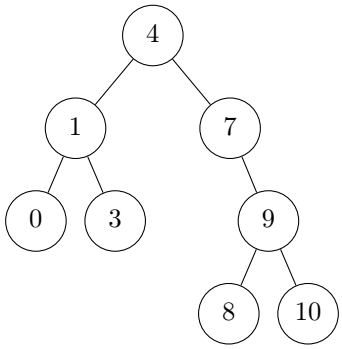
5 Arbres binaires de recherches

Une application des arbres binaires sont les arbres de recherche. L'objectif est de pouvoir chercher et ajouter des éléments efficacement dans une structure de donnée.

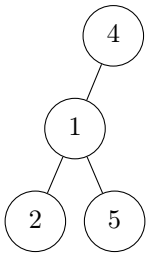
5.1 Utilisation d'un ordre total pour construire un arbre de recherche.

L'idée est la suivante : tous les nœuds dans le sous-arbre gauche auront une étiquette inférieur à celle du parent, et tous les nœuds dans le sous-arbre droit auront une étiquette supérieure à celle du parent.

Par exemple, l'arbre suivant est un arbre binaire de recherche :



Cependant, l'arbre suivant n'est pas un arbre binaire de recherche :



En effet, il y a la présence de 5 dans le sous-arbre gauche d'un nœud dont l'étiquette est 4.

Ici, nous avons utilisé des entiers pour étiquetter notre arbre, mais cela n'est pas nécessaire : nous pouvons utiliser tout ensemble totalement ordonné.

Définition 2.32 : Relation d'ordre

Une relation d'ordre sur un ensemble E est une relation binaire $\leqslant$ qui vérifie les propriétés suivantes :

$$\forall x \in E, x \leqslant x \qquad \text{(Reflexivité)}$$
$$\forall x, y \in E^2, (x \leqslant y \wedge y \leqslant x) \Rightarrow x = y \qquad \text{(Antisymmétrie)}$$
$$\forall x, y, z \in E^3, (x \leqslant y \wedge y \leqslant z) \Rightarrow x \leqslant z \qquad \text{(Transitivité)}$$

Définition 2.33 : Ensemble totalement ordonné

Un ensemble muni d'une relation d'ordre $(E, \leqslant)$ est dit totalement ordonné si pour tout $x, y \in E$, on a $x \leqslant y$ ou $y \leqslant x$ (ou les deux).

De cette manière, on peut donc construire la définition d'un arbre binaire de recherche suivante.

Définition 2.34 : Arbre binaire de recherche

Soit $(E, \leqslant)$ un ensemble totalement ordonné. Un arbre binaire est un **binaire de recherche** lorsque, pour tous nœud n d'étiquette e :
- Soit n n'as pas de fils gauche, soit pour tout nœud du sous-arbre gauche d'étiquette e', $e' \leqslant e$.
- Soit n n'as pas de fils droit, soit pour tout nœud du sous-arbre droit d'étiquette e', $e \leqslant e'$.

Exercice 2.8. Construire un arbre binaire de recherche avec les nombres de 1 à 12. On cherchera à minimiser la hauteur de l'arbre.

5.2 Opérations sur un arbre binaire de recherche

On peut ensuite utiliser cette structure pour chercher, ajouter ou retirer des éléments dans cette structure.

La première opération est la recherche d'une clef dans un arbre binaire de recherche.
L'idée principale derrière la recherche est de, de manière similaire à la recherche dichotomique, remarque que lorsqu'on compare la clef avec l'étiquette du nœud courant, on sait directement à quel sous-arbre s'intéresser.

Algorithme 2.2 : Recherche dans un arbre binaire de recherche

Entrée : A un arbre, x une étiquette
$r \leftarrow$ la racine de a.
Tant Que vrai **Faire**
  Si l'étiquette de r est égale à x **Alors**
    Renvoyer Vrai
  Si l'étiquette de r est supérieure strictement à x **Alors**
    Si r a un fils droit **Alors**
      $r \leftarrow$ le fils droit de r.
    Sinon
      Renvoyer Faux
  Sinon
    Si r a un fils gauche **Alors**
      $r \leftarrow$ le fils gauche de r.
    Sinon
      Renvoyer Faux

On peut implémenter cette recherche de la manière suivante en OCaml :

Exemple 2.16 : Recherche dans un arbre binaire de recherche en OCaml

```
let rec chercher a x = match a with
| Vide -> False
| Noeud(e, fg, fd) ->
        if e = x then true
        else if e>x then chercher fg x
        else chercher fd x
```

Il est par ailleurs très facile de trouver le minimum et le maximum dans un arbre binaire de recherche.

Exercice 2.9. Proposer une fonction pour trouver le minimum dans un arbre binaire de recherche.

Étant donné que nous n'avons à parcourir qu'un chemin depuis la racine, la complexité dépendra du chemin le plus long depuis la racine, c'est-à-dire de la hauteur de l'arbre.

Proposition 2.8 : Complexité temporelle de la recherche

La recherche dans un arbre binaire de recherche a une complexité temporelle en $O(h)$ où h est la hauteur de l'arbre dans le pire des cas.
La recherche dans un arbre binaire de recherche a une complexité temporelle en $O(n)$ où n est la taille de l'arbre dans le pire des cas.

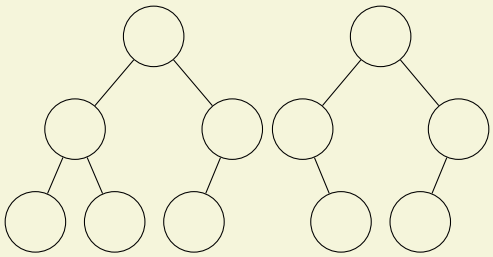
Cependant, si l'arbre est assez équilibré, on peut avoir une meilleur borne.

Définition 2.35 : Arbre presque complet côté gauche

On dit qu'un arbre binaire est **presque complet côté gauche** (ou parfois juste **presque complet**) lorsque tout ses niveaux sont complets, sauf potentiellement le dernier dont tous les nœuds sont alignés à gauche.

Exemple 2.17 : Arbre presque complet

Le première arbre est presque complet, mais pas le deuxième.



Proposition 2.9 : Hauteur d'un arbre presque complet

Un arbre presque complet de taille n est de profondeur $\lceil \log_2(n+1) \rceil - 1$.

On peut donc en déduire rapidement que, sous réserve que l'arbre est correctement équilibré :

Proposition 2.10 : Recherche dans un arbre binaire presque complet

La complexité temporelle de la recherche dans un arbre presque complet de taille n est en $O(\log(n))$.

Cela était pour la recherche. Maintenant, pour l'ajout, l'idée est la suivante : on procède comme pour la recherche, et on rajoute un nœud avec la clef en entrée

Algorithme 2.3 : Ajout dans un arbre binaire de recherche

Entrée : A un arbre, x une étiquette
$r \leftarrow$ la racine de a.
Tant Que vrai **Faire**
  Si l'étiquette de r est supérieure strictement à x **Alors**
    Si r a un fils droit **Alors**
      $r \leftarrow$ le fils droit de r.
    Sinon
      Le fils droit de r devient la feuille d'étiquette x.
      Renvoyer a
  Sinon
    Si r a un fils gauche **Alors**
      $r \leftarrow$ le fils gauche de r.
    Sinon
      Le fils gauche de r devient la feuille d'étiquette x.
      Renvoyer a

Pour l'implémentation en OCaml, nous devons reconstruire l'arbre en entier pour renvoyer l'arbre en raison de notre structure persistante pour les arbres.

Exercice 2.10. Proposer une fonction OCaml de signature `'a arbre -> 'a -> 'a arbre` qui calcule l'arbre binaire de recherche après l'ajout d'un élément en entrée.

5.3 Retrait dans un arbre binaire de recherche

Le retrait dans un arbre binaire de recherche est légèrement plus délicat. La première étape consiste à trouver le nœud que l'on cherche à retirer.

Si ce nœud est une feuille, il est facile de le retirer, on retire juste la feuille concernée sans toucher au reste de l'arbre. Si le nœud à retirer a un enfant exactement, il suffit de remplacer le nœud par son enfant.

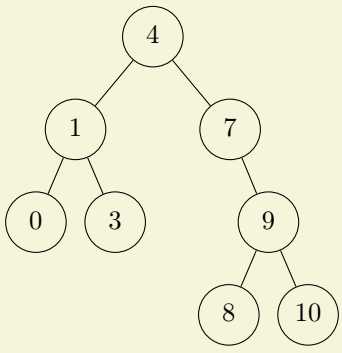
Cependant, pour les autres cas, nous allons avoir besoin d'une notion supplémentaire : le prédecesseur et le successeur. Pour les définir, on suppose que toutes les étiquettes sont distinctes (on pourrait se passer de cette définition, mais cela va nous convenir pour le moment).

Définition 2.36 : Prédécesseur, successeur dans un arbre binaire de recherche

Le *prédécesseur* d'un nœud n dans un arbre binaire de recherche est le nœud dont l'étiquette est la plus grande parmi les nœuds dont l'étiquette est plus petite que celle de n.
Le *successeur* d'un nœud n dans un arbre binaire de recherche est le nœud dont l'étiquette est la plus petite parmi les nœuds dont l'étiquette est plus grande que celle de n.

Exemple 2.18 : Prédecesseur, successeur dans un ABR

Dans l'arbre suivant, le prédecesseur du nœud d'étiquette 7 est la racine d'étiquette 4. Le sucesseur du nœud d'étiquette 1 est le nœud d'étiquette 3. Le successeur du nœud d'étiquette 3 est le nœud d'étiquette 4.

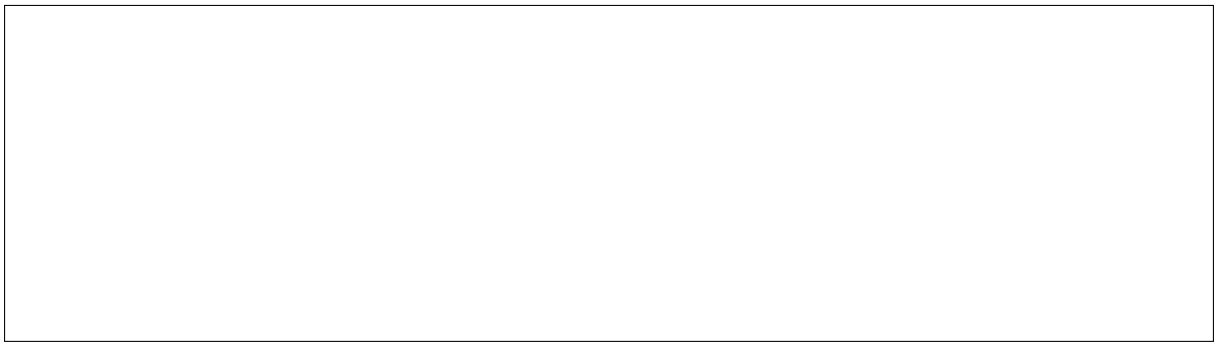


Le problème du calcul du prédecesseur et du successueur est qu'il est potentiellement nécessaire de remonter dans l'arbre pour trouver le nœud correspondant. Pour l'instant, on ne se pose pas la question de savoir comment cela peut être réalisé.

L'idée pour trouver un successeur est la suivante : si le nœud a un fils droit, il suffit de prendre le minimum du sous-arbre droit. Dans le cas contraire, il faudra chercher parmi les parents. On remonte les parents jusqu'à remonter dans un nœud par la gauche, ce qui nous assure que c'est le nœud le plus petit parmi les nœuds qui sont plus grand.

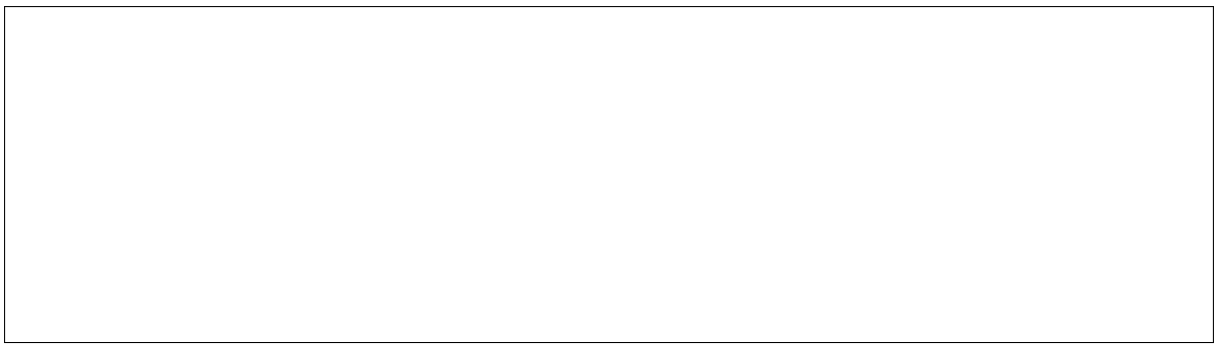
Algorithme 2.4 : Recherche d'un successeur dans un arbre binaire de recherche

Entrée : Arbre a, noeud n
  Si n a un enfant droit **Alors**
    Renvoyer Minimum du sous-arbre droit de n
  Tant Que n est un enfant droit **Faire**
    $n \leftarrow$ parent de n
  Renvoyer n



Lorsqu'on s'intéresse au cas du retrait dans le cas où le nœuds que l'on veut retirer a deux enfants, on sait que le successeur de notre nœud sera nécessairement dans le sous-arbre droit de notre arbre.

Par ailleurs, on sait que le successeur n'aura pas d'enfant gauche. Mais il subsiste toujours deux cas : le cas où le successeur est l'enfant directe du nœud à retirer, et le cas où ça n'est pas le cas.



L'avantage est qu'on peut donc que, même si on ne peut pas trouver le successeur dans le cas général avec notre type, le fait de savoir que nous sommes dans un cas spécifique fait que nous pouvons implémenter ce retrait facilement en OCaml.

Exercice 2.11. Implémenter la fonction de retrait en OCaml.

Proposition 2.11 : Complexité des opérations dans un arbre binaire de recherche

On note n la taille de notre arbre, et h sa hauteur. Les complexités temporelle des opération sur un arbre binaire de recherche sont données par le tableau suivant :

Recherche	Ajout	Retrait
$O(h)$, $O(n)$	$O(h)$, $O(n)$	$O(h)$, $O(n)$

Il s'avère que la borne $O(h)$ est effectivement mauvaise pour des arbres qui sont mal formés, mais dans la plupart des cas, les performances sont raisonnables.

5.4 Utilisation des arbres de recherche pour les ensembles et les tables d'association

Assez naturellement, on peut utiliser un arbre de recherche pour stocker un ensemble d'élément ordonnés. Par ailleurs, on peut faire un peu mieux en créant une table d'association, c'est-à-dire une structure de donnée dans laquelle il est possible de chercher, ajouter et retirer des associations clef-élément.

Définition 2.37 : Table d'association

Une **table d'association** (ou un **dictionnaire**) sur un ensemble de clefs E et d'éléments F est une structure dans laquelle il est possible de stocker des pairs d'éléments de $E \times F$ et de trouver ou retirer rapidement un élément à partir d'une clef e.
Dans une **table d'association**, pour chaque $e \in E$, il existe au plus une paire (e, f).

L'idée derrière une table d'association est d'avoir une structure dynamique qui permet de stocker des éléments par une clef, et ce avec un seul élément par clef. D'une certaine manière, c'est une généralisation non linéaire d'un tableau à un ensemble d'indices E au lieu de se restreindre à des entiers consécutifs.

Pour pouvoir réaliser un dictionnaire à l'aide d'un arbre binaire de recherche, il faut enrichir les étiquettes de l'élément à stocker.

Exemple 2.19 : Table d'association étiquettée par des entiers en OCaml

On peut utiliser le type suivant en OCaml pour stocker des éléments dans une table d'association dont les clefs sont des entiers :

```
type 'a table = Vide | Feuille of int * 'a * 'a table * 'a table
```

Ainsi, lors de l'ajout, de la recherche et du retrait, on utilisera la clef pour savoir où se trouve ou doit se trouver l'élément, puis on modifiera le nœud avec une information supplémentaire constitué de l'élément.

Exercice 2.12. Proposer une fonction OCaml qui ajoute un élément dans un tableau d'association.

Cependant, les complexités restent mauvaises dans le pire des cas, et on espère pouvoir la limiter avec des structures plus élaborées.

5.5 Arbres bicolores

Comme nous l'avons vu, la complexité de recherche et d'ajout dépendent de la structure de l'arbre, structure sur laquelle nous n'avons aucun contrôle. En particulier, certains cas pénibles surviennent

souvent dans le cas par exemple où l'on ajoute des éléments triés dans notre tableau, et on veut donc avoir des assurances pour ne jamais se retrouver dans ce genre de cas.

Certaines structures de données nous permettent d'éviter les cas difficile, et c'est le cas en particulier des *arbres bicolores*.

Définition 2.38 : Arbre bicolore

Un **arbre bicolore**
- Un nœud est soit noir soit rouge ;
- La racine est noire ;
- Tous les nœud ont 2 enfants qui peuvent être des nœud internes ou des feuilles ;
- Les étiquettes sont portées par les nœuds internes ;
- Les feuilles sont noires ;
- Les enfants d'un nœud rouge sont noirs ;
- Le chemin de la racine à toutes les feuilles contient le même nombre de nœuds noirs.

La recherche dans les arbres bicolores ne change pas par rapport à un arbre de recherche classique.

Si l'on conserve ces contraintes sur notre structures, la recherche est toujours efficace.

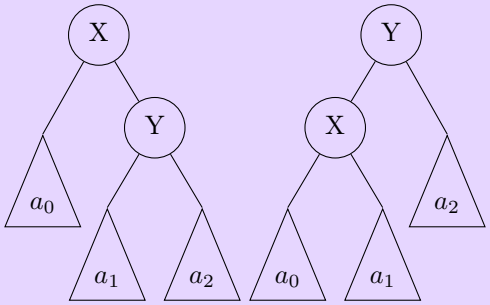
Proposition 2.12 : Recherche dans un arbre rouge-noir

La recherche dans un arbre de recherche rouge-noir de taille n s'exécute dans le pire des cas en $O(\log n)$.

Cependant, la question des opérations sur les arbres bicolore est nettement plus ardue.

Définition 2.39 : Rotations Gauche et Droite

La **rotation gauche** est la transformation de l'arbre de gauche en l'arbre de droite, c'est-à-dire la transformation d'un arbre d'étiquette X d'enfant droit d'étiquette Y en l'arbre d'étiquette Y d'enfant gauche X.



La **rotation droite** est la transformation inverse.

Proposition 2.13 : Respect des propriétés des arbres binaires de recherches

Les rotations sur un nœud d'un arbre binaire de recherche donnent des arbres qui respectent les propriétés des arbres binaires de recherche.

Algorithme 2.5 : Ajout dans un arbre rouge-noir

L'ajout dans un arbre rouge-noir procède comme pour l'ajout dans un arbre binaire de recherche avec les deux exceptions que le nœud ajouté est colorié en rouge, et qu'il faut ensuite appliquer l'une des règles suivantes pour rectifier la structure de l'arbre :

- **Le nœud ajouté est la racine :** il suffit de colorier ce nœud en noir.
- **Le parent du nœud inséré est noir :** il n'y a pas de modifications à faire, l'arbre vérifie déjà les propriétés attendues.
- **Le parent et l'oncle sont rouges :** l'oncle et le parent sont coloriés en noire et le grand-parent est colorié en rouge s'il était noir. Dans le cas où le grand parent était colorié en noir initialement, il faut appliquer récursivement la rectification de la structure à partir de cet élément.
- **Le parent est rouge, mais l'oncle est noir, et le nœud est un petit-enfant interne :** on fait une rotation pour se ramener au cas suivant.
- **Le parent est rouge, mais l'oncle est noir, et le nœud est un petit-enfant externe :** le parent est ramené à la position du grand parent par une rotation, et le parent change de couleur pour noir, tandis que le grand parent devient rouge.

Sans rentrer dans les détails du fonctionnement du retrait, on peut tout de même ennoncer la propriété suivante :

Proposition 2.14 : Complexité des opérations dans un arbre bicolore

On note n la taille de notre arbre, et h sa hauteur. Les complexités temporelle des opération sur un arbre bicolore sont données par le tableau suivant :

Recherche	Ajout	Retrait
$O(h)$, $O(\log n)$	$O(h)$, $O(\log n)$	$O(h)$, $O(\log n)$

6 Tas et files de priorité

La notion de file de priorité est une question de savoir, à tout moment et efficacement, quel est l'objet qui est le plus urgent à traiter. Une manière de représenter une file de priorité est d'utiliser une structure hiérarchique spécifique.

6.1 Structure de tas

Nous nous intéressons plus spécifiquement aux structures de tas binaires, mais il est bien sûr possible de retirer cette contrainte d'arité.

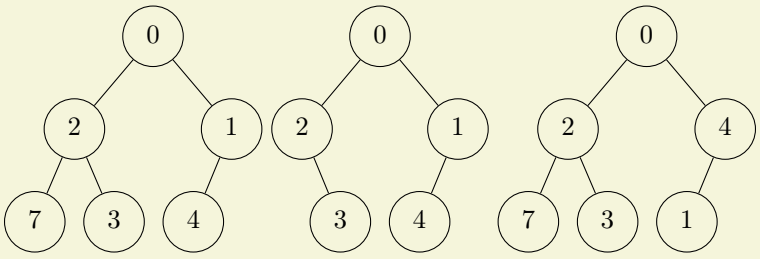
Définition 2.40 : Tas binaire

Un **tas min binaire** sur un ensemble totalement ordonné $(E, \leqslant)$ est une structure d'arbre binaire presque complet telle que pour tout nœud interne, l'étiquette du nœud est plus petite que toutes les étiquettes de ses enfants.
De même, on peut définir un **tas max binaire** comme une structure d'arbre binaire presque complet telle que pour tout nœud interne, l'étiquette du nœud est plus grande que toutes les étiquettes de ses enfants.

Il y a donc deux caractéristiques à prendre en compte dans un tas : le fait que, à l'échelle du nœud, on ait une relation d'ordre sur les nœuds ; et le fait que, à l'échelle de l'arbre dans sa globalité, celui-ci soit presque complet.

Exemple 2.20 : Tas

Parmi les arbres suivants, le premier est un tas min, le deuxième n'est pas un tas car il n'est pas presque complet, et enfin, le dernier n'est pas un tas min, car il ne respecte pas la propriété sur les étiquettes.



Il ne faut cependant pas confondre un *tas* en tant que structure de donnée, et le *tas* la zone mémoire pour l'allocation dynamique de taille variable. Cette collision dans le vocabulaire est malheureuse, car la manière dont s'organise le tas dans la mémoire n'a pas grand chose à voir avec un tas.

Cependant, dans la plupart des cas, nous n'aurons pas de soucis de confusion entre ces deux notions : elles appartiennent à des domaines éloignés de l'informatique.

6.2 Opérations sur un tas

Nous avons besoin des opérations suivantes pour les tas :

— Retirer le minimum (ou le maximum dans le cas d'un tas max) du tas et le renvoyer ;

— Ajouter un élément d'étiquette arbitraire dans le tas.

Bien sûr, nous avons besoin de conserver les deux contraintes du tas. Pour ce faire, nous allons procéder de la manière générale suivante : on essaye dans un premier temps de respecter la contrainte de presque-completude de l'arbre, puis on rétabli si nécessaire la contrainte d'ordre sur les étiquettes des nœuds.

On ne procédera donc qu'à des échanges entre un parent et un de ses enfants sans modifier la structure de l'arbre dans le deuxième temps.

Trouver l'élément minimum dans un tas min est très facile : il suffit de prendre la racine de l'arbre. Il faut maintenant que nous puissions le retirer efficacement.

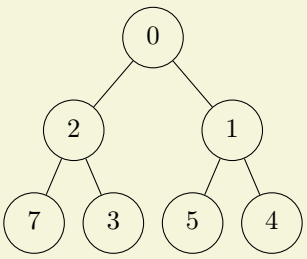
L'idée est la suivante : on remplace l'élément à la racine par l'élément le plus à droite parmi les éléments de plus grande profondeur, et on fait descendre l'élément dans la racine

Algorithme 2.6 : Retrait du minimum dans un tas min

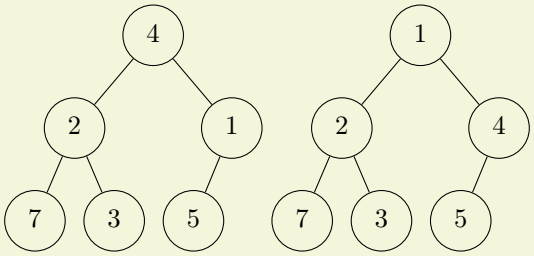
On sauvegarde la racine dans une variable ;
On remplace la racine par le dernier élément de l'arbre ;
Tant Que vrai **Faire**
  Si Si le dernier nœud modifié est plus petit que ses deux enfants **Alors**
    Renvoyer la racine que l'on avait sauvegardé.
  On inverse le dernier nœud échangé avec le plus petit de ses deux enfants.

Exemple 2.21 : Retrait dans un tas

On veut retirer le minimum du tas min suivant :



On doit réaliser les opérations suivantes :



On obtient bien un tas min à la fin privé de l'élément qui était la racine initialement.

Ainsi, pour retirer un élément, nous avions fait descendre un élément plus grand. Pour rajouter un élément, il faudra faire l'opération inverse : faire remonter l'élément à partir de la dernière position dans le tas.

Algorithme 2.7 : Ajout dans un tas

On ajoute une feuille dans le niveau incomplet, le plus à gauche possible ;
Tant Que vrai **Faire**
  Si Si le dernier nœud modifié est plus grand que son parent, ou qu'il s'agit de la racine **Alors**
    Renvoyer
  On inverse le dernier nœud échangé avec son parent.

Proposition 2.15 : Hauteur d'un tas

Un tas de taille n est de profondeur $\lceil \log_2(n+1) \rceil - 1$.

Proposition 2.16 : Complexité des opérations dans un tas min

On note n la taille de notre arbre. Les complexités temporelle dans le pire des cas des opération sur un tas sont données par le tableau suivant :

Trouver le minimum	Retrait du minimum	Ajout
$O(1)$	$O(\log n)$	$O(\log n)$

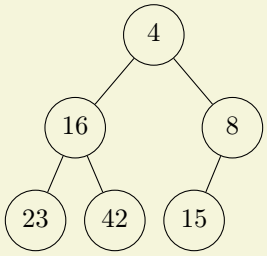
La raison pour laquelle tout cela marche si bien, c'est que la structure de l'arbre fait qu'il sera de toute façon tassé à une hauteur de l'ordre de $O(\log n)$. Nous avons pu le faire facilement grâce à l'affaiblissement de la contrainte des arbres binaires de recherche, mais le prix à payer est de ne pas pouvoir trouver facilement un élément d'une étiquette donnée.

6.3 Implémentation de tas

Il est possible d'implémenter un tas avec des structures hiérarchiques similaire à ce que nous avions plus tôt. Cependant, puisque l'arbre doit toujours être presque complet, nous pouvons supposer que tous les éléments sont contigus en mémoire, et nous pouvons donc utiliser une structure linéarisée.

Exemple 2.22 : Linéarisation d'un tas

On représente un tas par un tableau des éléments rencontrés dans leur parcours par niveau, ainsi, si on prend l'arbre suivant :



On peut en obtenir le tableau suivant :

4	16	8	23	42	15

Exercice 2.13. Si un nœud a pour indice i, quel est l'indice de sont parent ? Quel est l'indice de son fils gauche ? Quel est l'indice de son fils droit ?

On suppose que l'on dispose d'un tableau dynamique mutable de type `'a dynamique` qui possède les opérations suivantes :

- `taille` de signature `'a dynamique -> int` qui renvoie le nombre d'élément dans un tableau dynamique ;
- `retirer_fin` de signature `'a dynamique -> 'a` qui retire le dernier élément de l'arbre ;
- `ajouter_fin` de signature `'a dynamique -> 'a -> unit` qui ajoute un élément à la fin d'un tableau dynamique ;
- `modifier` de signature `'a dynamique -> int -> 'a -> unit` qui modifie la valeur dans un tableau dynamique à un indice donné ;
- `acceder` de signature `'a dynamique -> int -> 'a` qui renvoie la valeur dans le tableau dynamique à un indice donné.

Cela nous permet d'avoir la structure suivante pour un tas :

Exemple 2.23 : Structure de tas en OCaml

```
type 'a tas = 'a dynamique
```

Étant donné que nous ne voulons, dans la plupart des cas, qu'échanger deux éléments, on se donne une fonction qui permet d'échanger la valeur de deux éléments à des indices donnés :

Exemple 2.24 : Fonction d'échange de deux éléments dans un tas

```
let echanger t i j =
    let vi = acceder tas i in
    let vj = acceder tas j in
    modifier tas i vj ; modifier tas j vi
```

On peut se servir de ces primitives pour ajouter et retirer le minimum dans un tas.

Exemple 2.25 : Ajout dans un tas en OCaml

On se sert d'une fonction auxiliaire qui fait remonter un élément.

```
let ajouter t x =
    let rec remonter i =
        if i = 0 then ()
        else
            let i_parent = (i-1)/2 in
            let v_noeud = acceder t i in
            let v_parent = acceder t i_parent in
            if v_noeud < v_parent then
                begin
                    echanger i i_parent ;
                    remonter i_parent
                end
    in
        ajouter_fin t x ;
        remonter ((taille t) -1 )
```

Exemple 2.26 : Retrait du minimum dans un tas en OCaml

On se sert d'une fonction auxiliaire qui fait descendre un élément.

```
let retirer_minimum t x =
    let rec descendre i =
        let i_gauche = 2 * i + 1 in
        let i_droit = 2 * i + 2 in
        let v_noeud = acceder t i in
        if (2 * i + 1)>=(taille t)  then ()
        else if (2 * i + 1) = (taille t) - 1 then
            (if acceder t i_gauche < v_noeud then
                echanger t i i_gauche)
        else
            begin
                let v_gauche = acceder t i_gauche in
                let v_droit = acceder t i_droit in
                if v_gauche >= v_noeud && v_droit >= v_noeud then
                    ()
                else if v_gauche > v_droit then
                    (
                        echanger t i i_droit ;
                        descendre i_droit
                    )
                else
                    (
                        echanger t i i_gauche ;
                        descendre i_gauche
                    )
            end
    in
    let dernier = retirer_dernier t in
    let minimum = acceder t 0 in
    modifier t 0 dernier ;
    descendre 0;
    minimum
```

Le code du retrait est assez conséquent à cause des différents cas possible pour le nombre d'enfants,

et le choix de l'enfant dans lequel on s'engouffre.

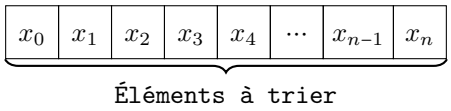
6.4 Tri par tas

Le fait de pouvoir ajouter efficacement et retirer efficacement le minimum dans notre tas nous permet de nous en servir pour réaliser un tri.

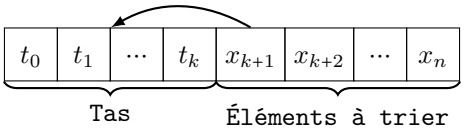
Algorithme 2.8 : Tri par tas

Entrée : Des éléments x_0, ..., x_n
$t \leftarrow$ un tas max vide.
$r \leftarrow$ un tableau de taille $n+1$.
Pour Chaque x_i **Faire**
  ajouter x_i à t
Pour Chaque i de n à 0. **Faire**
  $r[i] \leftarrow$ le minimum de t qu'on lui retire.
Renvoyer r

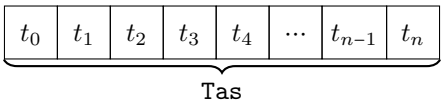
En pratique, dans l'implémentation, nous pouvons décider d'utiliser le tableau lui même que l'on veut trier en tant que tas. Initialement, aucun élément n'est dans le tas, et tous les éléments sont à trier :



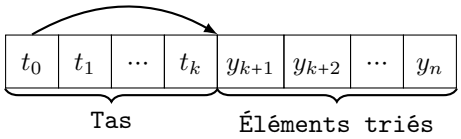
On rajoute ensuite les éléments peu à peu dans notre tas qui occupera de plus en plus de place au début du tableau.



Une fois que l'on a fini, il ne reste que le tas :



Il nous suffit de retirer les maximum successifs pour les ajouter à la fin du tableau :



Proposition 2.17 : Complexité du tri par tas

Le tri par tas sur un tableau de taille n se réalise en $O(n \log n)$ dans le pire des cas.

7 Ordres bien fondés

On cherche dans cette section à formaliser un peu mieux les question d'induction structurelle.

7.1 Ordre bien fondé

On se donne un ensemble ordonné $(E, \leqslant)$.

Définition 2.41 : Prédécesseur, sucesseur

Un *prédécesseur* de x est un élément y tel que $y \leqslant x$.
Un *successeur* de x est un élément y tel que $x \leqslant y$.

Définition 2.42 : Prédécesseur et successeur immédiat

Un *prédécesseur immédiat* de x est un prédécesseur de x distinct de x, y, tel qu'il n'existe pas de z distinct de x et y tel que $y \leqslant z \leqslant x$.
Un *successeur immédiat* de x est un successeur de x distinct de x, y, tel qu'il nexiste pas de z distinct de x et y tel que $x \leqslant z \leqslant y$.

Définition 2.43 : Élément minimal

Un *élément minimal* est un élément tel qu'il n'existe pas de y distinct de x tel que $y \leqslant x$.

Définition 2.44 : Ordre bien fondé

Un *ordre bien fondé* sur un ensemble E est une relation d'ordre telle qu'il n'existe pas de suite décroissante strictement infinie dans E au sens de cet relation.

Les ordres bien fondé possèdent une propriété similaires aux entiers. C'est sur des ensembles munis de tels ordres que l'on pourra utiliser le principe d'induction comme nous avions utilisé le principe de récurrence.

Définition 2.45 : Ensemble inductif

Un ensemble est dit *inductif* quand il dispose d'une relation d'ordre bien fondé.

7.2 Ordre induit

On se donne E un ensemble.

Proposition 2.18 : Ensemble construit par induction

Soit un sous-ensemble $B \subset E$ qu'on nomme les éléments de base, et un ensemble de fonctions $R = (r_i)_i$ tels que r_i soit une fonction d'arité a_i qui prend ses éléments arguments dans E et qui est à valeur dans E.
Il existe un petit (au sens de l'inclusion) ensemble F qui vérifie les propriétés suivantes :
- $B \subset F$;
- F est stable par chacun des $r_i \in R$.

On dit que F est **construit par induction avec les éléments de B et des règles R.**

Étant donné que nous nous sommes placés dans le cadre de E, nous pouvons faire la démonstration de l'existence de F.

Définition 2.46 : Ensemble construit par induction sans ambiguïté

On dit qu'un ensemble F construit par induction est **construit sans ambiguïté** si, pour chaque élément $e \in F$:
- Si e est un élément de base, alors il ne peut pas être construit avec une règle de construction ;
- Si e n'est pas un élément de base, alors il peut être construit avec exactement une règle r_i et un unique a_i-uplets d'éléments de F.

Proposition 2.19 : Ordre induit

Soit F ensemble **construit sans ambiguïté**. On définit **l'ordre induit** comme étant la relation suivante $\leqslant$ sur F par, pour tout f et f', $f \leqslant f'$ si et seulement si f' s'écrit avec f dans son unique construction.
Cet ordre induit est une relation d'ordre bien fondée.

Exemple 2.27 : Non-ambiguïté de la construction des arbres

Avec notre construction, les arbres dont les nœuds sont ordonnés sont construits sans ambiguïté.

Exercice 2.14. Montrer qu'un élément de base est un élément minimal pour l'ordre induit.

Proposition 2.20 : Fonction définie sur un ensemble construit par induction sans ambiguïté

Soit E' un ensemble. Soit F un ensemble construit sans ambiguïté.
On se donne une fonction f_i de E^{a_i} dans E' pour chaque règle r_i, et une valeur $e'_j \in E'$ pour tout élément de base e_j. Il existe exactement une fonction f telle que :

- $f(e_j) = e'_j$ pour chaque $e_j \in E$;
- Pour tout r_i, et tout a_i-uplets $b_1, ..., b_{a_i}$ de E, on a :

$$f(r_i(b_1, ..., b_{a_i})) = f_i(f(b_1), ..., f(b_{a_i}))$$

Cette proposition nous confirme que nous pouvons, dans le cas des ensembles construit sans ambiguïté, définir une fonction sur ces ensembles à partir de sa valeur sur les éléments de base et de sa valeur à partir d'une règle et de sa valeur sur des éléments qui permettent de construire un autre élément.

Dans cette section, nous avons considéré un ensemble plus grand E dans lequel on supposait que tous nos éléments possibles se trouvaient. Pour définir formellement les arbres, il faudrait donc avoir un ensemble E qui contienne au moins les arbres et que nous puissions ensuite construire sans ambiguïté.

Chapitre III

Structures de données relationnelles : Les Graphes

1 Théorie des graphes

Avant de pouvoir considérer les graphes d'un point de vue de la programmation ou de l'algorithmique, nous devons déjà les considérer d'un point de vue mathématique.

L'objectif est donc ici de faire un inventaire de différents types graphes et définitions, ainsi que de quelques propriétés élémentaires sur les graphes.

1.1 Graphes orientés

Définition 3.1 : Graphe orienté

Un graphe orienté G est un couple (S, A) tel que $A \subset \{(u, v) \in S^2 | x \neq y\}$. On suppose par ailleurs S fini.
S est l'ensemble des *sommets*.
A est l'ensemble des *arêtes*, que l'on nomme aussi dans le cas orienté *arcs*.

Exemple 3.1 : Quelques utilisations des graphes orientés

- Graphes de relations binaires ;
- Automates à état finis ;
- Graphes de flux de contrôle.

Souvent, on impose que les arêtes ne peuvent pas être des boucles, c'est-à-dire des flèches qui vont d'un sommet à lui-même.

Cependant, on veut parfois rajouter les boucles dans certaines définitions des graphes. On peut par ailleurs rajouter les arêtes multiples (c'est-à-dire qu'il peut exister plusieurs arêtes entre deux mêmes sommets).

Définition 3.2 : Degrés entrant et sortant

Soit (S, A) un graphe orienté. Soit $s \in S$ un sommet.
On note $d_-(s)$ le **degré entrant** de s le nombre d'arêtes dont la première composante est s. On note $d_+(s)$ le **degré sortant** de s le nombre d'arêtes dont la deuxième composante est s.
Ainsi :

$$d_+(s) = |\{(x, y) \in A | x = s\}|$$
$$d_-(s) = |\{(x, y) \in A | y = s\}|$$

Proposition 3.1 : Relation entre le degré entrant et le degré sortant

La somme des degrés entrant et celle des degrés sortants des nœuds dans un graphe sont égales au nombre d'arêtes.
Ainsi, pour tout graphe $G = (S, A)$:

$$|A| = \sum_{s \in S} d_+(s) = \sum_{s \in S} d_-(s)$$

Définition 3.3 : Chemin (à partir des arêtes)

Soit $G = (S, A)$ un graphe orienté. Un **chemin** $a_0 a_1 \cdots a_n$ est une suite finie d'arêtes de A telle que pour tout $1 \leqslant k \leqslant n$, en notant $a_k = (s_k, s'_k)$ et $a_{k-1} = (s_{k-1}, s'_{k-1})$, on ait $s'_{k-1} = s_k$, c'est-à-dire que le nœud d'arrivée d'une arête est le nœud de départ de la suivante.
La **longueur** de ce chemin est le nombre d'arêtes traversées, c'est-à-dire $n + 1$.

Le plus souvent, on note un chemin par la suite des nœuds traversés.

Définition 3.4 : Chemin (à partir des nœuds)

Soit $G = (S, A)$ un graphe orienté. Un **chemin** $s_0 s_1 \cdots s_n$ dans G est une suite finie de sommets de S telle que pour tout $1 \leqslant k \leqslant n$, (s_{k-1}, s_k) soit dans A, c'est-à-dire dont tous les couples consécutifs sont des arêtes du graphe.
La **longueur** de ce chemin est le nombre d'arêtes traversées, c'est-à-dire n. On dit qu'il s'agit d'un chemin de s_0 à s_n.

Bien sûr, ces deux définitions sont cohérentes, et on obtient bien la même longueur quelque soit la manière de considérer le chemin.

On peut aussi remarquer qu'avec la définition avec les nœuds, on peut considérer les chemins de longueur 0 qui consiste en un chemin composé d'un seul nœud.

En outre, on remarque que les arêtes traversées n'ont pas besoin d'être distinctes : on peut donc passer plusieurs fois par les mêmes nœuds et arêtes et la longueur est donc le nombre d'arêtes traversées *avec multiplicité*.

On peut se servir de cette remarque pour définir un cycle.

Définition 3.5 : Cycle dans un graphe orienté

Un **cycle** est un chemin de longueur non nulle tel que le premier nœud est égal au dernier nœud, c'est-à-dire que si il s'écrit $s_0 \cdots s_n$, alors $s_0 = s_n$, et qui passe par des arêtes toutes distinctes.

Parfois, on s'autorise certains cycles qui passent plusieurs fois par la même arête, et il s'avère que dans le cas orienté, cette condition n'est pas aussi importante que dans le cas orienté.

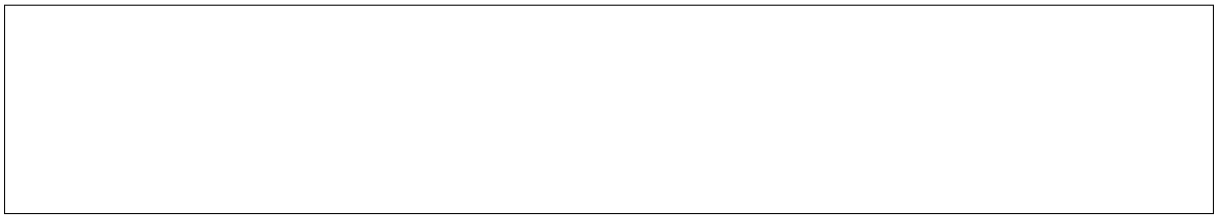
Dans tous les cas, puisque pour un cycle, il existe autant de manière de l'exprimer comme un chemin que de nœuds dans ce cycle, on veut considérer que des cycles sont égaux quand ils couvrent les mêmes arêtes : peu importe où ils commencent ou finissent.

Par exemple un cycle $u_0 u_1 \cdots u_k u_0$ est le même cycle que le cycle $u_1 \cdots u_{k-1} u_k u_0 u_1$.

Le sens est cependant important dans le cas orienté : on ne passe pas par les mêmes arêtes si l'on fait le cycle $u_0 u_1 u_2 u_0$ ou le cycle $u_0 u_2 u_1 u_0$.

Définition 3.6 : Chemin sans cycle

Un chemin est dit **sans cycle** si et seulement si il ne passe pas deux fois par le même nœud.



En appliquant le principe des tirroirs, on peut rapidement trouver une propriété pour les longueurs des chemins sans cycle.

Proposition 3.2 : Taille d'un chemin sans cycle

Soit $G = (S, A)$ un graphe. La longeur de tous chemin sans cycle est majoré par le nombre de sommets.

La question de savoir si cette borne est atteinte, c'est-à-dire s'il existe un chemin qui passe exactement une fois par chaque sommet est un peu plus compliqué, et nous allons la laisser pour plus tard.

Définition 3.7 : Graphe acyclique

Un graphe est dit **acyclique** s'il ne contient pas de cycle.

Grâce à la notion de chemin, on peut de plus définir la notion de connexité.

Définition 3.8 : Graphe fortement connexe

Un graphe orienté $G = (S, A)$ est dit **fortement connexe** si pour toute paire de sommets $s \neq s' \in S$, il existe un chemin de s à s'.

Parfois, on veut quelque chose de plus faible que la faible connexité : juste le fait qu'en ignorant le sens des flèches, on ait la connexité.

Définition 3.9 : Graphe faiblement connexe

Soit un graphe orienté $G = (S, A)$. Soit $G' = (S, A')$ avec A' défini de la manière suivante :

$$\forall s, s' \in S,\ (s, s') \in A' \Leftrightarrow (s, s') \in A \vee (s', s) \in A$$

G est dit **faiblement connexe** si et seulement si G' est fortement connexe.

Comme le nom le suggère, il y a une relation d'implication entre la forte et la faible connexité.

Proposition 3.3 : Forte connexité implique faible connexité

Tout graphe fortement connexe est faiblement connexe.

Exercice 3.1. Donne un contre-exemple pour la réciproque.

Définition 3.10 : Sous-graphe

Soit $G = (S, A)$ un graphe orienté. $G' = (S', A')$ est un **sous-graphe** de G si G' est un graphe et que $S' \subset S$ et $A' \subset A$.

La contrainte sur le fait que G' est un graphe impose que les arêtes de G' doivent avoir leurs extrémités dans S', cependant, on n'est pas obligé de prendre l'intégralité des arêtes.

On peut rajouter cette contrainte grâce à la notion de sous-graphe induit par un sous-ensemble de sommets.

Définition 3.11 : Sous-graphe induit

Soit $G = (S, A)$. Soit $S' \subset S$, et soit $A' = (S' \times S') \cap A$. $G' = (S', A')$ est le **sous-graphe de G induit par S'**.

Exercice 3.2. Montrer qu'un sous-graphe induit est bien un sous-graphe.

Les notions de sous-graphe et de connexité nous permettent de définir les composantes connexes, c'est-à-dire les morceaux connexes du graphe.

Définition 3.12 : Composante fortement connexe

Une **composante fortement connexe** est un sous-graphe fortement connexe maximal pour l'inclusion, c'est-à-dire qu'on ne peut pas agrandir en conservant la propriété de forte connexité.

Définition 3.13 : Composante faiblement connexe

Une **composante faiblement connexe** est un sous-graphe faiblement connexe maximal pour l'inclusion.

Proposition 3.4 : Décomposition d'un graphe en ses composantes connexes

L'ensemble des composantes fortement connexes d'un graphe forment une partition des nœuds de ce graphe.
L'ensemble des composantes faiblement connexes d'un graphe forment une partition des nœuds de ce graphe.

Exercice 3.3. Montrer que l'ensemble des composantes faiblement connexes d'un graphe orienté forment une partition des arêtes de ce graphe.

1.2 Graphes non-orientés

Il est parfois intéressant de considérer des graphes qui n'ont pas de directions pour les arêtes : il s'agit juste de prendre en compte la relation entre deux nœuds et pas le sens de cette relation.

Définition 3.14 : Paire

Une paire est un ensemble à deux éléments.

L'important dans une paire est qu'on peut retirer la question de l'ordre.

Définition 3.15 : Graphe non-orienté

Un graphe non-orienté G est un couple (S, A) tel que A est un ensemble de paires de S.

Exemple 3.2 : Quelques utilisations de graphes non-orientés

— Graphes pour des relations binaires symmétriques ;
— Graphes d'intervalles.

On remarque qu'on a exclue les boucles de notre définition des graphes orientés.

Définition 3.16 : Degré dans un graphe non-orienté

Le degré d'un nœud est le nombre d'arêtes à laquel il appartient. Ainsi, pour un graphe non-orienté (S, A) et un sommet $s \in S$, le degré de s noté $d(s)$ (ou parfois $\deg(s)$) est égal à :

$$d(s) = |\{a \in A | s \in a\}|$$

Proposition 3.5 : Les poignées de main

La somme des degré des sommets d'un graphe est égal à deux fois le nombre d'arêtes.
Ainsi, pour (S, A) un graphe :

$$\sum_{s \in S} d(s) = 2\,|A|$$

Définition 3.17 : Graphe non-orienté complet

Un graphe non-orienté complet $G = (S, A)$ est un graphe dont les arêtes sont toutes les paires possibles de S :

$$A = \{\{s, s'\} \,|\, s, s' \in S, s \neq s'\}$$

Quitte à renommer les nœuds, il existe exactement un graphe complet à n sommets.

Le graphe complet nous donne une borne supérieure sur le nombre d'arêtes.

Proposition 3.6 : Borne sur le nombre d'arêtes dans un graphe non-orienté

Dans un graphe (S, A), on a la relation suivante :

$$\frac{|S](|S| - 1)}{2} \geqslant |A|$$

Le cas d'égalité étant exactement le cas du graphe complet.

Pareillement que pour les graphes orientés, on peut définir des chemins sur les graphes non-orientés.

Définition 3.18 : Chemin

Soit $G = (S, A)$ un graphe non-orienté. Un **chemin** de G, $s_0 s_1 \cdots s_n$ pour $n > 0$, est une suite finie de sommet telle que pour tout $0 \leqslant k \leqslant n - 1$, on ait $\{s_k, s_{k+1}\} \in A$.

On note $n-1$ la longueur de ce chemin, et on dit que ce chemin est un chemin est entre s_0 et s_n.

Pour des raisons de simplicité, on ne fait pas la distinction entre les chemins $s_0 s_1 \cdots s_n$ et $s_n s_{n-1} \cdots s_0$ et on peut donc considérer que ceux-ci sont égaux.

Bien sûr, on peut définir les chemins par la suite des arêtes traversées.

Pour ce qui est du cas non-orienté pour les cycles, il est important de noter qu'un cycle ne passe pas plusieurs fois par la même arête, en effet dans le cas contraire, on pourrait s'intéresser

Définition 3.19 : Cycle dans un graphe non-orienté

Un cycle dans un graphe non-orienté est un chemin entre un sommet u et ce même sommet u et qui passe par des arêtes distinctes.

Lorsqu'on s'intéresse aux cycles, généralement, on s'intéresse à rotation circulaire des nœuds près : le cycle $u_0 u_1 \cdots u_n u_0$ est le même cycle que le cycle $u_1 u_2 \cdots u_n u_0 u_1$. De plus, comme nous sommes dans le cas non-orienté, il s'agit du même cycle que le cycle $u_0 u_n u_{n-1} \cdots u_1 u_0$.

Proposition 3.7 : Caractérisation des graphes non-connexes acycliques

Un graphe non-orienté est acyclique si et seulement si, pour toute paire de sommets u et v, il existe au plus un chemin entre u et v.

Il n'y a cependant pas besoin de faire la distinction entre faible et forte connexité : on peut se restreindre à la notion de connexité. Il ne s'agit plus de savoir si on peut aller de tout sommet d'un sommet à un autre, mais s'il existe un chemin entre les pairs de

Définition 3.20 : Graphe connexe

Un graphe $G = (S, A)$ non-orienté est dit **connexe** si pour tout $(s, s') \in S$, il existe un chemin entre s et s'.

On peut conserver les notions de sous-arbres et sous-arbres induits.

Définition 3.21 : Composante connexe dans un graphe non-orienté

Une composante connexe est un sous-arbre connexe maximum pour l'inclusion.

Proposition 3.8 : Composantes connexes et partition dans un graphe non-orienté

Les composantes connexes d'un graphe forment une partition de ce graphe pour les sommets et pour les arêtes.

1.3 Graphes pondérés

Pour l'instant, nous n'avons considéré les graphes que sous la forme de nœuds reliés ou non dans un sens ou dans l'autre. Dans la plupart des applications des graphes, il est intéressant de pouvoir étiquetter les arêtes par des valeurs qui donnent une information.

Définition 3.22 : Graphe pondéré

Un **graphe pondéré** (pour ses arêtes) est un graphe muni d'une fonction $p : A \to \mathbb{R}$.
p est la fonction de poids, et ses valeurs sont les poids.

Parfois, on parle aussi de graphes pondéré quand ce sont les sommets qui ont une valeur et la fonction p est définie sur S.

Par ailleurs, on peut pondérer par des valeurs qui ne sont pas nécessairement réels, soit en rajoutant la contrainte que les poids doivent dans un ensemble plus petit, ou en changeant totalement l'ensemble possible pour les poids.

Définition 3.23 : Poids d'un graphe

Le poids d'un graphe est égal à la somme des poids de ses éléments.
Ainsi, pour un graphe pondéré par ses arêtes :

$$P(G) = \sum_{a \epsilon A} p(A)$$

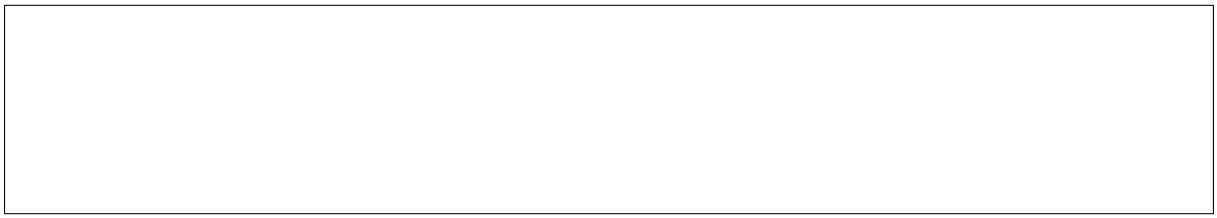
Exercice 3.4. Montrer que si tous les poids sont strictement positifs, le sous-graphe de poid maximal est exactement le graphe lui-même.
Que se passe-t-il s'il y a des poids égaux à 0 ou strictement négatifs ?

1.4 Les Arbres comme des graphes

Les graphes ressemblent à une généralisation des arbres, et on peut donc considérer les arbres comme un sous-ensemble des graphes.
Les deux principales propriétés des arbres par rapport aux graphes sont d'une part la nécessité de connexité, et d'autre part le fait qu'il n'existe toujours exactement qu'un chemin entre deux nœuds.

Définition 3.24 : Arbre au sens de la théorie des graphes

Un **arbre** au sens de la théorie des graphes est un graphe non-orienté connexe acyclique.

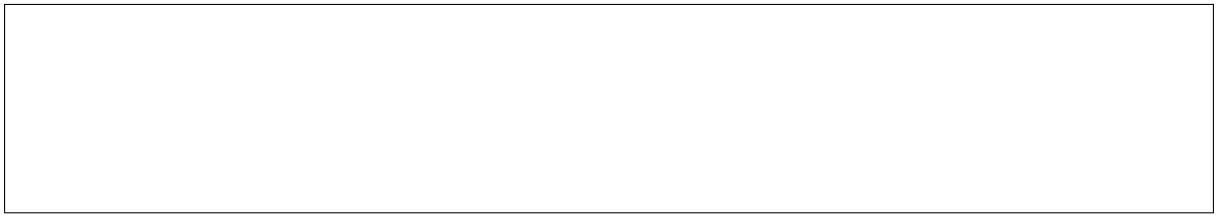


La différence fondamentale qui subiste entre les arbres au sens de la théorie des graphes, et les arbres que nous avons manipulé dans le passé est l'absence de questions de hiérarchie : il n'y a pas de racine ni de sens généalogique.

Cette apauvrissement de la structure donne quelques différences notables, en particulier sur la notion de sous-arbre (la notion de sous-graphe ne peut pas imposer d'inclure tous les descendants d'un nœud)

Définition 3.25 : Sous-arbre au sens de la théorie des graphes

Un **sous-arbre** d'un arbre au sens de la théorie des graphes est un sous-graphe connexe et acyclique de cet arbre.



Cette définition conserve bien la propriété selon laquelle il existe exactement un seul chemin entre deux nœuds :

Proposition 3.9 : Nombre de chemins dans un arbre

Soit G un graphe. G est un arbre si et seulement si pour chaque paire de sommet, il existe un unique chemin entre ces sommets.

Bien sûr, nous pouvons retomber sur la définition usuelle des arbres en rajoutant une racine.

Définition 3.26 : Arbres enracinés au sens de la théorie des graphes

Un **arbre enraciné** $T = (S, A, r)$ est un triplet tel que (S, A) est un arbre au sens de la théorie des graphes, et $r \in S$.
On dit que r est la **racine** de l'arbre enraciné.

Proposition 3.10 : Équivalence des définitions

Il existe une bijection naturelle entre les arbres enracinés au sens de la théorie des graphes et les arbres d'arité arbitraires définis par induction.

Proposition 3.11 : Nombre d'arêtes dans un arbre

Soit $T = (S, A)$ un arbre non vide. Alors $|S| - 1 = |A|$.

Proposition 3.12 : Nombre d'arêtes dans un graphe connexe

Soit $G = (S, A)$ un graphe connexe non vide. Alors on a la relation suivante :

$$|S| - 1 \leqslant |A|$$

Le cas d'égalité correspond exactement aux arbres.

1.5 Isomorphisme de graphes

Une notion importante dans les graphes est le fait qu'on s'intéresse parfois à la *forme* des graphes et non aux éléments exactes qui représente les sommets.

Définition 3.27 : Isomorphisme de graphes orientés

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes orientés. Un **isomorphisme de graphes orientés** entre G et G' est une bijection φ de S dans S' telle que (u, v) soit une arête de A si et seulement si $(\varphi(u), \varphi(v))$ est une arête de A'.

De la même manière, dans le cas non-orienté, on peut définir la notion d'isomorphisme.

Définition 3.28 : Isomorphisme de graphes non-orientés

Soit $G = (S, A)$ et $G' = (S', A')$ deux graphes non-orientés. Un **isomorphisme de graphes non-orientés** entre G et G' est une bijection φ de S dans S' telle que $\{u, v\}$ soit une arête de A si et seulement si $\{\varphi(u), \varphi(v)\}$ est une arête de A'.

Enfin, on peut définir la notion de graphes isomorphes à l'aide de la notion d'isomorphisme.

Définition 3.29 : Graphes isomorphes

Deux graphes G et G' sont isomorphes s'il existe un isomorphisme entre les deux.

Les isomorphismes nous permettent de définir des classes d'équivalences sur les graphe.

Cela nous permet de raisonner à isomorphisme près, c'est-à-dire en ne comptant qu'à un renommage des nœuds près.

Exercice 3.5. Combien de graphes non-orientés à 3 sommets existent-ils à isomorphisme près ?

Proposition 3.13 : Unicité du graphe complet à n sommets

Tous les graphes complets à n sommets sont isomorphes deux à deux.

2 Représentation des graphes

Après avoir vu les graphes d'un point de vue purement théorique, il nous faut une manière de les représenter en mémoire pour pouvoir les manipuler.

La manière dont on les représente va déterminer le coût mémoire de la représentation et le coût des opérations sur la représentation.

2.1 Matrice d'adjacence

Une première représentation qui peut nous intéresser est d'utiliser une matrice qui contienne les informations liées au graphes.

L'idée est la suivante : on choisit une énumération des nœuds de sorte à pouvoir nommer un sommet par son numéro dans l'énumération. Ainsi, une arête sera associée à un couple (ou à une paire) d'entiers entre 1 et $|S|$, et on peut stocker ces informations à l'aide d'une matrice de taille $|S| \times |S|$.

Dans la suite, on identifie un nœud avec son énumération.

Les coefficients d'une matrice M de taille $n \times n$ sont notés $m_{i,j}$ pour $1 \leqslant i \leqslant n$ où i est la ligne du coefficient et j est la colonne.

De même, les coefficients d'une matrice M' sont notés $m'_{i,j}$.

Définition 3.30 : Représentation d'un graphe orienté par une matrice

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, ..., n\}$.
La matrice d'adjacence M de G est une matrice de taille $n \times n$ dont les coefficients sont dans $\{0, 1\}$ et sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

Définition 3.31 : Représentation d'un graphe non-orienté par une matrice

Soit $G = (S, A)$ un graphe non-orienté avec $S = \{1, \cdots, n\}$.
La matrice d'adjacence M de G est une matrice de taille $n \times n$ dont les coefficients sont dans $\{0, 1\}$

et sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } \{i,j\} \in A \\ 0 & \text{sinon} \end{cases}$$

Proposition 3.14 : Propriété des matrices d'adjacence des graphes non-orientés

Une matrice d'adjacence d'un graphe non-orienté est symmétrique.

Une autre propriété dans notre cas non-orienté où on a supposé l'absence de boucles est le fait que la diagonalle est nulle.

On remarque que la matrice d'adjacence dépend de la manière dont on a énuméré les nœuds.

On remarque que la matrice dépend de l'ordre qu'on a donné sur les nœuds.
Il y a plusieurs manière de s'en sortir : soit on contraint l'ordre des sommets au moment de la construction du graphe, soit on travaille à isomorphisme de graphe près.

Proposition 3.15 : Caractérisation d'un graphe à isomorphise près par sa matrice d'adjacence

Deux graphes G et G' de matrices d'adjacence respectives M et M' sont isomorphes si et seulement si les deux conditions suivantes sont vérifiées :

1. G et G' ont le même nombre de sommets n.
2. Il existe une permutation σ de $\{1,...,n\}$ telle que, pour tout $1 \leqslant i,j \leqslant n$, on a $m_{i,j} = m'_{\sigma(i),\sigma(j)}$.

Il serait tentant d'utiliser une matrice de booléens, mais il s'avère qu'utiliser des entiers nous donne des propriétés bien bratique pour l'exponentiation de matrices d'adjacence.

Proposition 3.16 : Exponentiation de la matrice d'adjacence

Soit G un graphe dont la matrice d'adjacence est M.
Pour tout $k > 0$, M^k représente la matrice dont l'élément d'indice (i,j) correspond au nombre de chemins de longueur exactement k qui vont de i à j.

En considérant qu'il existe un chemin exactement de longueur 0 à lui-même, on peut même rajouter le cas $k = 0$, ce qui est cohérent avec l'exponenitation

Bien sûr, dans le cas pondéré, on peut stocker, au lieu de noter seulement qu'il y a une arête, on peut noter les poids des arêtes.

Définition 3.32 : Représentation d'un graphe orienté complet pondéré par une matrice d'adjacence

Soit $G = (S, A, p)$ un graphe pondéré orienté complet avec $S = \{1, \cdots, n\}$. La matrice d'adjacence de G est une matrice M de taille $n \times n$ telle que :

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ p((i,j)) & \text{sinon.} \end{cases}$$

Si le graphe n'est pas complet, on peut avoir utiliser une valeur par défaut qui corresponde au fait de ne pas avoir de lien.

Bien sûr, on peut étendre la définition au cas complet non-orienté.

Définition 3.33 : Représentation d'un graphe orienté complet pondéré par une matrice d'adjacence

Soit $G = (S, A, p)$ un graphe pondéré non-orienté complet avec $S = \{1, \cdots, n\}$. La matrice d'adjacence de G est une matrice M de taille $n \times n$ telle que :

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ p(\{i,j\}) & \text{sinon.} \end{cases}$$

Par ailleurs, dans le cas où la valeur par défaut n'est pas 0, on peut vouloir modifier les éléments de la diagonales selon les cas.

Exercice 3.6. Proposer une matrice d'adjacence pour le graphe des trajets SNCF. Quelle valeur par défaut peut-on utiliser ?

2.2 Représentation par matrice d'adjacence en OCaml

Assez naturellement, en OCaml, on peut utiliser un tableau de tableau pour représenter une matrice d'adjacence.

Exemple 3.3 : Matrice d'adjacence pour des graphes en OCaml

On peut utiliser le type suivant en OCaml pour représenter une matrice d'adjacence en OCaml :

```
type graphe = int array array
```

Dans ce cas, on cherche à ce que `m.(i).(j)` soit égal à `1` si et seulement si $m_{i,j}$ vaut 1

On peut ensuite utiliser cette structure pour des programmes sur les graphes.

Exemple 3.4 : Nombre d'arêtes dans un graphe orienté

```
let nb_aretes mat_adj =
    let n = Array.length mat_adj in
    if n = 0 then 0 else
    begin
        let res = ref 0 in
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                res:= !res + mat_adj.(i).(j)
            done
        done ;
        !res
    end
```

Exercice 3.7. Proposer une fonction qui renvoie le nombre d'arêtes dans un graphe non-orienté.

Proposition 3.17 : Taille mémoire d'un graphe par matrice d'adjacence

La taille mémoire d'un graphe représenté par sa matrice d'adjacence est en $O(|S|^2)$.

Proposition 3.18 : Coût de parcours des arêtes par matrice d'adjacence

Le parcours de toutes les arêtes d'un graphe représenté par sa matrice d'adjacence est en $O(|S|^2)$.

Proposition 3.19 : Coût d'accès à une arête par matrice d'adjacence

L'accès à une arrête par matrice d'adjacence à l'aide des indices se fait en $O(1)$.

2.3 Représentation par matrice d'adjacence en C

De la même manière, en C, on peut utiliser une matrice pour représenter les matrices d'adjacence.

Il y a plusieurs manière de représenter des matrices en C.

Une première manière de procéder est d'utiliser des tableaux de taille déterminée statiquement. C'est la méthode recommandée dans le programme.

Nous avons utilisé peu de tableau de taille statique pour l'instant, et la plupart du temps, nous avons réduit le problème à une question de pointeurs. Cependant, le fait de contraindre la taille permet en C de savoir où se trouvent les élèments en C.

Exemple 3.5 : Tableau de taille statique en C passé sur la pile

```c
const int taille = 10;

void afficher(int t[taille][taille]){
    for (int i = 0; i<taille; i++){
        for (int j = 0; j<taille;j++){
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }
}
```

Exemple 3.6 : Tableau de taille statique en C passé par pointeur

```c
const int taille = 10;

void afficher(int (*t)[taille][taille]){
    for (int i = 0; i<taille; i++){
        for (int j = 0; j<taille;j++){
            printf("%d ", (*t)[i][j]);
        }
        printf("\n");
    }
}
```

Exercice 3.8. Proposer une fonction de prototype `int nombre_aretes(int (*mat)[taille][ taille ])` qui calcule le nombre d'arêtes dans un graphe non-orienté donné par sa matrice d'adjacence.

Il s'avère qu'il est possible de manipuler des tableaux alloués dynamiquement en C.

Exemple 3.7 : Tableau de tableaux dynamiques en C

```c
int ** creer_matrice(int n, int m){
    int ** mat = (int **) malloc(n * sizeof(int *));
    for (int i = 0; i< n;i++){
        int * ligne = (int *) malloc(m * sizeof(int));
        mat[i] = ligne;
    }
    return mat;
}
int acceder(int * mat, int i, int j){
    return mat[i][j];
}
```

Une autre manière de faire est d'utiliser un tableau de taille $n \times m$ pour avoir un tableau linéarisé.

Exemple 3.8 : Tableau linéarisé en C

```c
int * creer_matrice(int n, int m){
    int * mat = (int*) malloc(n * m * sizeof(int));
    return mat;
}
int acceder(int * mat, int i, int j, int m){
    return mat[i * m + j];
}
```

Exercice 3.9. Proposer une fonction de prototype `int * lineariser (int ** mat, int n, int m)` qui crée un tableau linéarisé à partir de la matrice `mat`.

La fonction libèrera l'espace alloué à la matrice `mat`.

Proposition 3.20 : Complexité avec l'implémentation en C

On obtient les mêmes complexités avec l'implémentation en C par matrice d'adjacence.

Il s'avère que ces complexités sont les complexités que l'on obtient quand on utilise la représentation classique d'une matrice sous la forme d'un tableau $n \times n$.

Dans le cas de certaines matrices creuses (avec peu d'arêtes non nulles), on peut se permettre d'utiliser d'autres représentation qui limitent le coût en espace.

En pratique, si l'on s'attend à avoir peu d'arêtes, on peut utiliser une autre manière de représenter les graphes.

2.4 Listes d'adjacence

L'idée de la liste d'adjacence est la suivante : à chaque nœud on associe la liste des nœuds qui lui sont adjacents. Ces listes sont stockées dans un tableau de listes.

Définition 3.34 : Listes d'adjacence d'un graphe orienté

Soit $G = (S, A)$ un graphe orienté avec $S = \{1, \cdots, n\}$.
Les listes d'adjacence de G sont des listes $(l_i)_{1 \leqslant i \leqslant n}$ tels que, pour tout $1 \leqslant i, j \leqslant n$, $j \in l_i$ si et seulement si $(i, j) \in A$.

Exercice 3.10. Quelles sont les listes d'adjacence du graphe cyclique à n sommets ?

La définition s'étend sans problème aux graphes non-orientés.

Définition 3.35 : Listes d'adjacence d'un graphe non-orienté

Soit $G = (S, A)$ un graphe non-orienté avec $S = \{1, \cdots, n\}$.
Les listes d'adjacence de G sont des listes $(l_i)_{1 \leqslant i \leqslant n}$ tels que, pour tout $1 \leqslant i, j \leqslant n$, $j \in l_i$ si et seulement si $\{i, j\} \in A$.

2.5 Représentation par listes d'adjacence en OCaml

Exemple 3.9 : Listes d'adjacence en OCaml

On peut représenter les listes d'adjacence d'un graphe à l'aide du type suivant :

```
type graphe = int list array
```

On peut se servir de ce type pour réaliser des programmes sur les graphes.

Exemple 3.10 : Calcul du nœud d'arité maximale grâce aux listes d'adjacence

```
let arite_max g =
    let max_vu = ref 0 in
    let n = Array.length g in
    for i = 0 to n do
        max_vu = max (!max_vu) (List.length g.(i))
    done ;
    !max_vu
```

Pour ce qui est des graphes pondérés, on utilise parfois des couples indice-poids pour permettre de stocker l'information du poids.

Exemple 3.11 : Listes d'adjacence en OCaml pour les graphes pondérés

On peut représenter les listes d'adjacence d'un graphe pondéré par des flottants à l'aide du type suivant :

```
type graphe = (int * float) list array
```

Proposition 3.21 : Taille mémoire d'un graphe par liste d'adjacence

La taille mémoire d'un graphe représenté par sa liste d'adjacence est en $O(|S| + |A|)$.

Proposition 3.22 : Coût de parcours des arêtes par matrice d'adjacence

Le parcours de toutes les arêtes d'un graphe représenté par sa liste d'adjacence est en $O(|S|+|A|)$.

Proposition 3.23 : Coût d'accès à une arête par matrice d'adjacence

L'accès à une arrête par liste d'adjacence à l'aide des indices se fait en $O(|S|)$.

2.6 Représentation par listes d'adjacence en C

Une solution pour représenter les listes d'adjacence est d'utiliser un tableau de tableaux dont la première case est toujours le nombre d'éléments dans la liste d'adjacence.

Exemple 3.12 : Listes d'adjacence en C

On peut utiliser un tableau de tableaux à l'aide du type `int **` pour représenter les listes d'adjacence en C.

À proprement parler, il ne s'agit pas de listes au sens habituel, mais nous les manipulerons de manière similaire.

Exemple 3.13 : Recherche d'arête par liste d'adjacence en C

```
bool est_voisin(int ** g, int i, int j){
    for(int k = 1; k<g[i][0];k++) {
        if (g[i][k] == j){
            return true;
        }
    }
    return false;
}
```

Exercice 3.11. Proposer une fonction de prototype `int ** cyclique(int n)` qui construit le graphe représenté par listes d'adjacence qui correspond au graphe cyclique de taille n.

Proposition 3.24 : Coûts de la représentation par listes d'adjacence en C

En C, on obtient les mêmes complexités qu'en OCaml avec les représentations par listes d'adjacence.

2.7 Récapitulatif

Voici un récapitulatif des performances attendues avec l'implémentation classique des différentes représenation des graphes.

	Matrices d'adjacence	Listes d'adjacence
Complexité mémoire	$O(\|S\|^2)$	$O(\|S\|+\|A\|)$
Complétixé de l'accès à une arête donnée par ses indices	$O(1)$	$O(\|S\|)$
Complexité du parcours de toutes les arêtes	$O(\|S\|^2)$	$O(\|S\|+\|A\|)$

Une représentation ou une autre peut être plus ou moins intéressante selon le comportement attendue.

3 Parcours de graphes

3.1 Notions de parcours

Tout comme pour les arbres, il est possible de parcourir les éléments d'un graphe. Souvent, les éléments qui nous intéressent sont les sommets.

Définition 3.36 : Parcours des sommets d'un graphe

Un **parcours de graphe** est un algorithme qui traite tous les sommets d'un graphe une fois exactement.

Bien sûr, on peut s'intéresser à un parcours des arêtes.

Le plus souvent, on pourrait parcourir directement les nœuds d'un graphe dans l'ordre, mais de sorte à respecter la structure de graphe, on fonctionne par exploration de proche en proche des nœuds à l'aide de leur relation de voisinage.

Pour réaliser ce type de parcours, dans la plupart des cas, on va utiliser un principe où tous les nœuds seront dans trois états possibles : soit ils ont été déjà vus, soit ils sont en attente car on a vu un de leur voisin, soit ils n'ont pas du tout été vus.

Algorithme 3.1 : Structure générale d'un parcours

Dans la plupart des cas, les parcours prennent la forme suivante.

Pour Chaque Sommet s dans le graphe **Faire**
- **Si** Si s n'est pas vu ou en attente **Alors**
  - On ajoute s aux sommets en attente.
  - **Tant Que** Il y a un sommet en attente **Faire**
    - Soit u un sommet en attente.
    - u est ajouté aux sommets traités.
    - On ajoute les voisins de u qui ne sont pas traités ou en attente aux sommets en attente.

3.2 Parcours en profondeur

Étant donné que nous ne sommes pas dans un arbre binaire et que nous n'avons pas de certitudes sur l'arité des nœuds, nous ne pouvons pas définir d'ordre infixe, mais nous pouvons cependant définir une notion de parcours préfixe et postfixe.

Définition 3.37 : Parcours en profondeur d'un graphe

Un **parcours en profondeur** d'un graphe est un parcours de nœud qui, pour chaque voisin, traite entièrement les nœuds accessibles depuis ce voisin parmi les nœuds qui sont encore à traiter avant de passer au voisin suivant.

Définition 3.38 : Parcours préfixe d'un graphe

Un **parcours préfixe** d'un graphe est un parcours en profondeur qui traite le nœud après avant d'avoir traité tous les nœuds accessible depuis lui.

Définition 3.39 : Parcours postfixe d'un graphe

Un **parcours postfixe** d'un graphe est un parcours en profondeur qui traite le nœud après avoir traité tous les nœuds accessible depuis lui.

De même, on peut en déduire un ordre à partir d'un parcours en considérant l'ordre dans lequel les éléments sont traités.

Cependant, contrairement à ce qui a été observé dans les arbres, comme il n'y a pas de racines qui permettent d'avoir un point de départ canonique, ni un ordre sur les nœuds qui permettent de déterminer dans quel ordre visiter les voisins, l'ordre obtenu n'est pas unique.

Exemple 3.14 : Affichage des nœuds dans l'ordre préfixe en OCaml

On considère un graphe représenté par sa matrice d'adjacence. On utilise un tableau qui se souvient des nœuds qui ont déjà été mis en attente, et on utilise pour le traitement.

```
let afficher_prefixe_graphe g =
    let n = Array.length g in
    let deja_vu = Array.make n false in
    let rec explorer k =
        if not deja_vu.(k) then
            begin
                deja_vu.(k)<- true;
                Printf.printf "%d\n" k ;
                for i = 0 to n-1 do
                    if g.(k).(i) ==1 then
                        explorer i
                done
            end
    in
    for i = 0 to n-1 do
        explorer i
    done
```

Exercice 3.12. Proposer une implémentation de la fonction précédente quand le graphe est représenté par ses listes d'adjacence.

3.3 Parcours en largeur

Définition 3.40 : Parcours en largeur

Un **parcours en largeur** est un parcours qui traite les voisins avant de traiter les nœuds qui sont encore à traiter parmi les nœuds accessibles depuis les voisins.

Exemple 3.15 : Utilisation de parcours en largeur

— Recherche de plus cours chemin dans un graphe non pondéré ;
— Calcul de diamètre d'un graphe.

Exercice 3.13. En utilisant une file, proposer une implémentation d'une fonction qui affiche les nœuds dans l'ordre obtenu par le parcours en largeur.

3.4 Arbre obtenu par un parcours

Même si l'exploration d'un graphe ne correspond pas exactement à une exploration d'arbre, on peut obtenir un arbre à partir d'un parcours de proche en proche.

Proposition 3.25 : Arbre obtenu à partir d'un parcours de proche en proche

Lors d'un parcours de proche en proche d'un graphe connexe, le sous-graphe constitué des nœuds du graphe et dont les arêtes sont les arêtes qui ont permis d'ajouter un nœud à la liste d'attente est un arbre.
On parle de l' **arbre de parcours d'un graphe**.

Exercice 3.14. Proposer une fonction qui réalise un parcours en profondeur et qui créer l'arbre de ce parcours.

4 Recherche de chemin dans un graphe

La question de recherche de chemin dans un graphe est une question centrale dans la plupart des problèmes qui sont modélisés par des graphes.

4.1 Accessibilité dans un graphe

Définition 3.41 : Accesibilité dans un graphe orienté

Un nœud v est accessible depuis un autre nœud u dans un graphe orienté s'il existe un chemin de u à v.

Définition 3.42 : Accessibilité dans un graphe non-orienté

Un nœud v est accesible depuis un autre nœud u dans un graphe non-orienté s'il existe un chemin entre u et v.

Proposition 3.26 : Connexité et accessibilité dans un graphe orienté

Dans un graphe orienté, tous les éléments d'une composante fortement connexe sont accessibles depuis chaque élément de la composante fortement connexe.
De plus, pour chaque composante fortement connexe et un sommet de u de cette composante connexe, la composante fortement connexe est exactement les nœuds v accessibles depuis u tels que u soit accessible depuis u, ainsi que u.

Proposition 3.27 : Connexité et accessibilité dans un graphe non-orienté

Dans un graphe non-orienté, tous les éléments d'une composante connexe sont accessibles depuis chaque élément de la composante connexe.
De plus, pour chaque composante connexe et un u de cette composante connexe, la composante connexe est exactement l'ensemble des nœuds accessibles depuis u.

En particulier, dans un graphe orienté (*resp* non-orienté) fortement connexe (*resp.* connexe), tous les éléments sont accesibles depuis chaque élément du graphe.

Pour trouver tous les sommets accessibles, il suffit de réaliser un parcours de proche en proche à partir d'un nœud.

On peut donc en déduire un algorithme pour trouver les composantes connexes d'un graphe non-orienté.

Algorithme 3.2 : Trouver les composantes connexes dans un graphe non-orienté

$\mathcal{P} \leftarrow \varnothing$
Tant Que il existe un sommet qui ne soit pas dans un ensemble de la partition $\mathcal{P}$ **Faire**
    $s \leftarrow$ un sommet qui n'est pas dans un ensemble de la partition $\mathcal{P}$.
    $C \leftarrow$ la composante connexe de s obtenue par parcours de proche en proche.
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{C\}$
Renvoyer P la partition obtenue

Exercice 3.15. Proposer une implémentation en C de cet algorithme en utilisant une matrice de taille statique.

On utiliseras un tableau de taille $|S|$ pour stocker les numéros des composantes connexes des éléments.
Quelle est sa complexité temporelle ?

4.2 Plus court chemin dans un graphe non pondéré

S'il existe un chemin d'un nœud à un autre, on peut s'intéresser au minimum des longueurs parmi les chemins

Définition 3.43 : Plus court chemin entre deux nœuds dans un graphe orienté

Soit $G = (S, A)$ un graphe orienté. Soit $u, v \in G$. On note $d(u, v)$ la **distance de u à v** la grandeur définie de la manière suivante :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemin de } u \text{ à } v, \\ l & \text{sinon.} \end{cases}$$

Où l est le minimum des distances parmi les chemins de u à v s'il existe.
Un chemin de u à v qui atteint ce minimum est un **plus court chemin de u à v**.

Il s'agit bien d'un minimum car les longueurs sont entières.

Définition 3.44 : Plus court chemin entre deux nœuds dans un graphe non-orienté

Soit $G = (S, A)$ un graphe non orienté. Soit $u, v \in G$. On note $d(u, v)$ la **distance de u à v** la grandeur définie de la manière suivante :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemin entre } u \text{ et } v, \\ l & \text{sinon.} \end{cases}$$

Où l est le minimum des distances parmi les chemins entre u et v s'il existe.
Un chemin de u à v qui atteint ce minimum est un **plus court chemin de u à v**.

La notion de distance nous permet de définir des propriétés géométrique sur les graphes.

Étant donné que l'on a défini la distance même sur des nœuds qui ne sont pas accessibles, on peut définir ces propriétés sur des graphes qui n'ont pas nécessairement de contraintes de connexité.

Définition 3.45 : Rayon d'un graphe

Le rayon d'un graphe est le minimum sur les nœuds des maximum des distances aux autres nœuds. Ainsi, pour $G = (S, A)$ un graphe, on a le rayon de G noté $r(G)$ égal à :

$$r(G) = \min_{u \in S} \max_{v \in S} d(u, v)$$

Définition 3.46 : Diamètre d'un graphe

Soit $G = (S, A)$. Le **diamètre** de ce graphe est le maximum des distances entre deux nœuds. Ainsi, pour $G = (S, A)$ un graphe, on a le rayon de G noté $\text{diam}(G)$ égal à :

$$\text{diam}(G) = \max_{u,v \in S} d(u, v)$$

Proposition 3.28 : Diamètre et rayon d'un graphe non connexe

Le diamètre et le rayon d'un graphe non-orienté non connexe sont infinis.

Algorithme 3.3 : Plus court chemin dans un graphe

Pour obtenir le plus court chemin dans un graphe non-orienté (*resp.* orienté) entre u et v (*resp.* de u à v), on peut réaliser un parcours en largeur du graphe depuis u et s'arêter dès que l'on trouve v.

4.3 Plus court chemin dans un graphe pondéré

Dans un graphe pondéré, la notion de chemin n'est pas exactement la même, au lieu de s'intéresser à la distance en nombre d'arêtes, on veut s'intéresser au *poids d'un chemin*.

Définition 3.47 : Poids d'un chemin dans un graphe pondéré

Le **poids d'un chemin** $c = u_0 \cdots u_n$ est donné par la somme des poids des arêtes qui le compose. Ainsi, dans un graphe orienté :

$$p(u_0 \cdots u_n) = \sum_{k=0}^{n-1} p((u_k, u_{k+1}))$$

Et dans un graphe non-orienté :

$$p(u_0 \cdots u_n) = \sum_{k=0}^{n-1} p(\{u_k, u_{k+1}\})$$

La notion de poids d'un chemin n'est pas exactement le poids du sous-graphe car il faut prendre en compte la multiplicité des arêtes.

Cela change la notion de distance dans un graphe pondéré : d'une part on doit considérer les plats, et d'autres part on doit s'intéresser un infimum plutôt qu'un minimum.

Définition 3.48 : Distance entre deux sommets dans un graphe pondéré orienté

La distance de u à v est déterminer par l'infimum sur les poids des chemins de u à v, en prenant 0 si $u = v$, et $+\infty$ s'il n'y a pas de chemins de u à v.
Ainsi :

$$d(u,v) = \begin{cases} 0 & \text{si } u = v, \\ +\infty & \text{s'il n'existe pas de chemins de } u \text{ à } v, \\ \inf_{c \in \mathcal{C}(u,v)} p(c) & \text{sinon.} \end{cases}$$

Où $\mathcal{C}(u,v)$ est l'ensemble des chemins de u à v.

Définition 3.49 : Distance entre deux sommets dans un graphe pondéré non-orienté

La distance de u à v est déterminer par l'infimum sur les poids des chemins entre u et v, et $+\infty$ s'il n'y a pas de chemins entre u et v.
Ainsi :

$$d(u,v) = \begin{cases} +\infty & \text{s'il n'existe pas de chemins entre } u \text{ et } v, \\ \inf_{c \in \mathcal{C}(u,v)} p(c) & \text{sinon.} \end{cases}$$

Où $\mathcal{C}(u,v)$ est l'ensemble des chemins entre u et v, en condisérant le chemin vide dans le cas $u = v$ (qui nous donne donc une distance maximale de 0).

Naturellement, on peut définir la notion de diamètre et de rayon à l'aide de ces notions de distance.

La question est désormais d'obtenir ces plus courts chemins dans un graphe pondéré.

4.4 Algorithme de Dijkstra

Le premier algorithme de recherche de chemin dans un graphe est l'algorithme de Dijkstra. L'algorithme part d'un sommet source, et qui rajoute peu à peu tous les sommets à un sous-graphe en prenant à chaque fois le plus proche du sommet source parmi les sommets qui ne sont pas dans le sous-graphe.

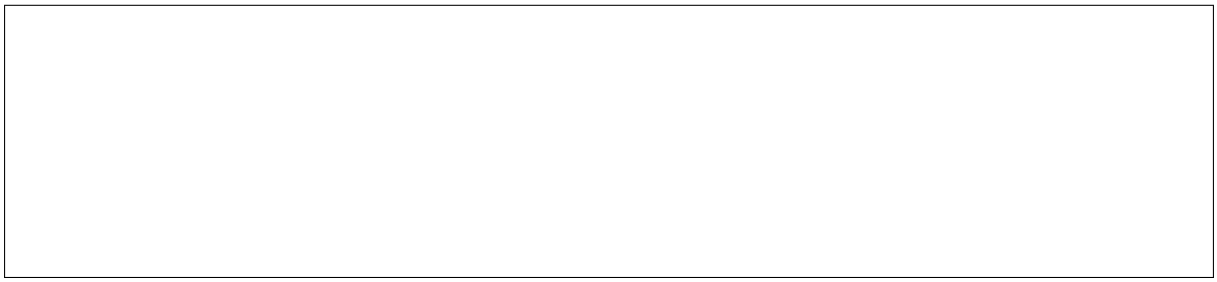
L'idée est donc de prendre à chaque fois un sommet qui est le plus proche.

Algorithme 3.4 : Algorithme de Dijkstra

On se donne $G = (S, A, p)$ un graphe orienté pondéré avec des poids positifs. Soit s un sommet source.

$P \leftarrow \varnothing$
$d[u] \leftarrow +\infty$ pour chaque sommet $u \in S$.
$d[s] \leftarrow 0$
Tant Que $S \setminus P$ contient au moins un sommet tel que $d[a] < +\infty$ **Faire**
  $a \leftarrow$ le sommet de $S \setminus P$ de plus petite distance $d[a]$.
  $P \leftarrow \{a\} \cup P$
  Pour Chaque sommet $b \in S \setminus P$ voisin de a **Faire**
    Si $d[b] > d[a] + p(a, b)$ **Alors**
      $d[b] \leftarrow d[a] + p(a, b)$

À la fin de l'algorithme, d contient le tableau des distances de s à chacun des sommets $u \in S$.



Proposition 3.29 : Invariant sur l'algorithme de Dijkstra

À chaque début de boucle dans l'algorithme de Dijkstra, on a les propriétés suivantes :

1. Pour tout sommet u de P, $d[u] = d(s, u)$;
2. Pour tout sommet u de S, $d[u] \geqslant d(s, u)$;
3. Pour tout sommet u, de $S \setminus P$ qui minimise $d[u]$ sur cet ensemblle, $d[u] = d(s, u)$.

Cette invariant nous donne une justification de l'algorithme de Dijkstra.

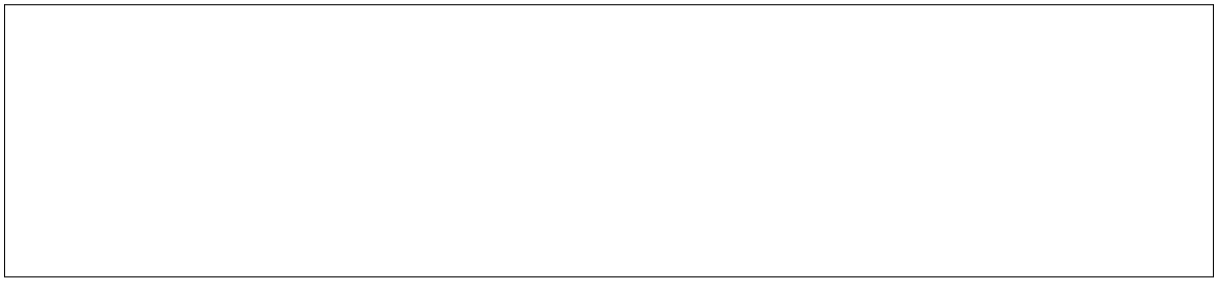
L'algorithme nous donne certes les distances des plus court chemins, mais il nous permet en plus de construire un arbre qui nous donne les plus court chemins depuis le sommet utilisé comme point de départ.

Cet algorithme est un algorithme *glouton*, à chaque étape, il cherche la meilleur action à faire, et cette optimalité locale lui permet d'avoir l'optimalité globable en remarquant que, à chaque fois qu'on ajoute un sommet à p, on a pas de manière strictement meilleure que le chemin proposé jusque là pour se rendre à ce sommet.

Pour formaliser tout cela, on s'intéresse à l'arbre qui résulte des modifications de d.

Définition 3.50 : Arbre construit par l'algorithme de Dijkstra

Soit $G = (S, A)$ un graphe orienté et soit $s \in S$ un sommet.
L'arbre construit par l'algorithme de Dijkstra est un sous-graphe $G' = (S', A')$ de G dont les sommets sont dans l'ensemble P construit, et les arêtes sont, pour chaque sommet b sauf pour le sommet s, l'arête (a, b) qui ait en dernier modifié la valeur de $d[b]$.



Proposition 3.30 : Distance dans l'arbre construit par l'algorithme de Dijkstra

Soit $G = (S, A)$ un graphe orienté.
Soit u un sommet. Soit v un sommet accessible depuis u et soit $G' = (S', A')$ l'arbre construit par l'algorithme de Dijkstra.
L'unique chemin de u à v dans G' est un plus court chemin de u à v dans G.

Pour pouvoir retrouver cet arbre, on construit un tableau des prédecesseur, c'est-à-dire un tableau, pour chaque sommet accessible, de l'autre extrémité de l'arête par laquelle on doit passer juste avant pour aller à ce sommet.

Algorithme 3.5 : Algorithme de Dijkstra avec les prédecesseurs

On se donne $G = (S, A, p)$ un graphe orienté pondéré avec des poids positifs. Soit s un sommet source.

$P \leftarrow \varnothing$
$d[u] \leftarrow +\infty$ pour chaque sommet $u \in S$.
$d[s] \leftarrow 0$
Tant Que $S \setminus P$ contient au moins un sommet tel que $d[a] < +\infty$ **Faire**
    $a \leftarrow$ le sommet de $S \setminus P$ de plus petite distance $d[a]$.
    $P \leftarrow \{a\} \cup P$
    Pour Chaque sommet $b \in S \setminus P$ voisin de a **Faire**
        Si $d[b] > d[a] + p(a, b)$ **Alors**
            $d[b] \leftarrow d[a] + p(a, b)$
            predecesseur$[b] \leftarrow a$

À la fin de l'algorithme, pour chaque sommet u accessible à part s, predecesseur contient le sommet qui le précède dans le chemin de s à u.

Pour réaliser l'implémentation de Dijkstra, on suppose que le graphe est représenté par ses listes d'adjacences. Cette représentation est particulièrement intéressante dans notre cas car on veut pouvoir parcourir les arêtes qui partent d'un sommet.

Un des autres enjeux est la manière dont on stocke les distances qu'il nous reste à voir.

Si on le fait de manière naïve, c'est-à-dire en utilisant un tableau des sommets qu'il nous reste à voir, nous allons obtenir une mauvaise complexité.

Proposition 3.31 : Complexité naïve de l'implémentation de l'algorithme de Dijkstra

L'algorithme de Dijkstra naïf a une complexité temporelle en $O(n^2)$ où n est le nombre de sommets du graphe.

On peut faire mieux en utilisant une file de priorité, un tas min. Cette structure est particulièrement intéressante car on peut chercher rapidement le sommet dont la distance à P est la plus proche.

Exemple 3.16 : Implémentation de l'algorithme de Dijkstra

On suppose qu'on dispose d'un tas de type `'a tas` dans lequel on peut ajouter des éléments à l'aide de la fonction `ajouter_tas: 'a tas -> 'a -> unit` et dans lequel on peut retirer l'élément le plus petit avec `retirer_tas: 'a tas -> 'a`.
On stockera les éléments dans le tas au format `(distance, indice)`.
Le graphe `g` est supposé du type `int list array`, et on a en entrée les deux sommets qui nous intéressent.
La fonction renvoie la distance entre u et v.

```
let distance_dijkstra g u v =
let n = Array.length g in
let deja_vu = Array.make  n false in
let t = creer_tas () in
let rec aux_liste d l = match l with
| [] -> ()
| (distance, sommet) ::q ->
    ajouter_tas t (distance + d, sommet);
    aux_liste d q
and aux_boucle () =
    let (d, sommet) = retirer_tas t in
    if sommet = v then d
    else
        begin
            if not deja_vu.(sommet) then
                begin
                    deja_vu.(sommet)<-true ;
                    aux_liste d g.(sommet)
                end ;
            aux_boucle ()
        end
in ajouter_tas t (0, u) ; aux_boucle ()
```

On a orienté le code pour qu'il s'arrête dès que v est ajouté au sous-graphe.

Proposition 3.32 : Complexité en utilisant un tas binaire de l'algorithme de Dijkstra

L'algorithme de Dijkstra avec un tas binaire a une complexité temporelle en $O((n+m)\log n)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes du graphe.

Il s'avère que l'on pourrait faire mieux avec d'autres structures plus élaborées qu'un tas-min, mais il s'agit pour l'instant d'un calcul qui a la complexité que l'on attend.

Aussi, l'algorithme de Dijkstra est exactement le même dans le cas non-orienté.

4.5 Algorithme de Floyd-Warshall

Un deuxième algorithme au programme est l'algorithme de Floyd-Warshall. Il consiste à calculer de proche en proche une matrice des distances entre les sommets en ne prenant en compte que les sommets qui passe par un sous-ensemble de sommets que l'on va faire évaluer.

Algorithme 3.6 : Algorithme de Floyd-Warshall

Soit $G = (S, A, p)$ un graphe orienté pondéré. On note $n = |S|$.

Pour Chaque $i, j \in S$ **Faire**

$$m[i][j] \leftarrow \begin{cases} 0 & \text{si } i = j, \\ p((i,j)) & \text{si } (i,j) \in A, \\ +\infty & \text{sinon.} \end{cases}$$

Pour Chaque k de 1 à n **Faire**

Pour Chaque i de 1 à n **Faire**

Pour Chaque j de 1 à n **Faire**

Si $m[i][j] > m[i][k] + m[k][j]$ **Alors**

$m[i][j] \leftarrow m[i][k] + m[k][j]$

À la fin de l'algorithme, la matrice M contient les distances de i à j de sorte à ce que $m[i][j] = d(i,j)$.

L'idée derrière cet algorithme est, qu'à l'étape k, dans la matrice $M^{(k)}$, $m_{i,j}^{(k)}$ contient la distance de i à j en ne passant que par les sommets intermédiaires de $\{1, \cdots, k\}$.

Ainsi, on utilise la relation de récurrence suivante :

$$m_{i,j}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ p((i,j)) & \text{si } (i,j) \in A, \\ +\infty & \text{sinon.} \end{cases}$$

$$m_{i,j}^{(k)} = \min\left(m_{i,j}^{(k-1)}, m_{i,k}^{(k-1)} + m_{k,j}^{(k-1)}\right)$$

Cet algorithme utilise la *programmation dynamique* qui est une manière de programmer qui consiste à utiliser des sous-structures optimales interdépendantes pour construite la solution d'un problème d'optimisation.

Nous verrons plus de choses sur la programmation dynamique dans le prochain chapitre.

Pour réaliser l'implémentation de l'algorithme de Floyd-Warshall, on s'intéresse à la représentation par matrice d'adjacence. En effet, c'est probablement le plus simple pour construire les valeurs initiales de $M^{(0)}$.

Exemple 3.17 : Implémentation de l'algorithme de Floyd-Warshall en C

Dans cette fonction, on suppose que l'entrée est la matrice qui vaut 0 sur la diagonale, $p(i,j)$ si (i,j) est une arrête à l'indice (i,j), et INT_MAX s'il n'y a pas d'arêtes.
La matrice d'entrée est modifiée directement, et elle est supposée de taille connue à la compilation.

```c
void floyd_warshall(int m[taille][taille]){
    for (int k = 0; k< taille; k++){
        for (int i= 0; i< taille; i++){
            for (int j= 0; j< taille; j++){
                if (m[i][k]!=INT_MAX && m[k][j]!=INT_MAX){
                    if (m[i][j]>m[i][k]+m[k][j]){
                        m[i][j]= m[i][k]+m[k][j]
                    }
                }
            }
        }
    }
}
```

On n'a pas besoin de faire l'initialisation ni de construire une nouvelle matrice pour la sortie car l'entrée est supposée au bon format.
Par ailleurs, on a n'a pas non plus utilisé une version temporaire de la matrice m pour construire l'état suivant : il s'avère que l'on aura toujours au moins quelque chose d'aussi bien que le résultat attendu.
La seule subtilité est la manipulation des INT_MAX de sorte à ne pas entraîner de dépassement de mémoire que l'on aurait pas rencontré autrement.

La complexité, elle, est relativement facile à calculer : il s'agit de trois boucle for imbriquée, toutes en n étapes.

Proposition 3.33 : Complexité de l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall a une complexité temporelle en $O(n^3)$ où n est le nombre de sommets du graphe.

Idéalement, pour retrouver les différents chemins, il faudrait utiliser une variante des prédecesseur, mais cette fois-ci sous la forme d'une matrice telle que predecesseur$[i][j]$ contienne le sommet qui précède j dans le chemin de i à j et de mettre à jour cette matrice au fur et à mesure du calcul.

	Dijkstra	Floyd-Warshall
Résultat	Tous les plus courts chemins depuis un sommet d'origine s	Tous les plus courts chemins entre deux sommets
Représentation	Listes d'adjacence	Matrice d'adjacence
Complexité temporelle	$O((\|S\|+\|A\|)\log\|S\|)$ avec un tas	$O(\|S\|^3)$

5 Quelques types de graphes particuliers

5.1 Graphes orientés acycliques

Dans cette section, nous nous intéressons aux graphes orientés acycliques.

Définition 3.51 : Puit, source

Soit $G = (S, A)$ un graphe orienté.
Un **puit** est un sommet de degré sortant nul.
Une **source** est un sommet de degré entrant nul.

Proposition 3.34 : Présence d'une source et d'un puit dans un graphe orienté acyclique

Un graphe orienté acyclique $G = (S, A)$ avec S non vide admet au moins un puit.
Un graphe orienté acyclique $G = (S, A)$ avec S non vide admet au moins une source.

Proposition 3.35 : Nombre d'arêtes dans un graphe orienté acyclique

Dans un graphe acyclique $G = (S, A)$, il y a au plus $\frac{|S|(|S|-1)}{2}$ arêtes.

Une notion qui vas nous intéresser est la notion de tri topologique : il s'agit d'un ordre sur les nœuds de sorte à ce qu'aucune flèche ne permette de remonter dans le tri.

Définition 3.52 : Tri topologique

Soit $G = (S, A)$ un graphe orienté avec n sommets. Un tri topologique est une énumération des sommets $u_1, ...u_n$. telle que pour pour tout $1 \leqslant i \leqslant j \leqslant n$, on ait $(j, i) \notin A$.
Un tri topologique nous dit que pour tout arrête $(u_j, u_i) \in A$, on a $j < i$.

Intuitivement, cela correspond à résoudre, à partir d'un graphe des dépendances, un ordre dans lequel réaliser les opérations de sorte à ce que l'on traite les dépendances dans le bonne ordre.

Il s'avère que dans le cas d'un graphe orienté acyclique, on peut trouver un tri topologique, et que réciproquement, le fait d'être acyclique est suffisant pour l'existence d'un tri topologique.

Proposition 3.36 : Tri topologique et graphe orienté acyclique

Soit $G = (S, A)$ un graphe orienté. G est acyclique si et seulement si G admet un tri topologique.

Proposition 3.37 : Algorithme

L'ordre postfixe d'un parcours en profondeur à partir des sources d'un graphe orienté acyclique nous donne un tri topologique de ses nœuds.

Mieux encore, on peut utiliser cette propriété pour vérifier si un arbre est un graphe orienté acyclique. Si on parcours les éléments et qu'on retrouve

Exemple 3.18 : Tri topologique et recherche de cycle

On se donne un graphe orienté représenté par ses listes d'adjacence de type int list array. Le type de sortie est le tri topologique sous la forme d'une suite des nœuds de type int list On suppose qu'on dispose d'une fonction pour trouver les sources en temps linéaire en le nombre de sources qui renvoie un tableau de taille $|S|$ dans lequel l'élément d'indice i vaut vrai si et seulement si le sommet i est une source.

```
let tri_topologique g =
    let n = Array.length g in
    let est_source = trouver_sources g in
    let deja_vu = Array.make n false in
    let rec aux_liste l acc = match l with
    | [] -> acc
    | p::q -> aux_liste q (aux_sommet p acc)
    and aux_sommet i acc =
        if deja_vu.(i) then acc
        else
            begin
                deja_vu.(i)<- true ;
                i::(aux_liste g.(i) acc)
            end
    and aux_source i acc =
        if i = n then acc
        else
            if est_source.(i) then
                aux_source (i+1) (aux_sommet i acc)
            else
                aux_source (i+1) acc
    in (aux_source 0 [])
```

5.2 Graphe de flux de contrôle

Un autre types de graphes qui va nous intéresser sont les graphes de flux de contrôle qui vous nous donner un peu de formalisme pour les

Définition 3.53 : Graphe de flux de contrôle d'un programme

On se donne P un programme. Le **graphe de flux de contrôle d'un programme** est un graphe orienté dont les sommets sont les instructions de base du programme et dont les arêtes sont les sauts possibles qu'il peut y avoir entre chaque instruction de base.
Souvent, on étiquette les arêtes par la condition nécessaire pour que cette transition soit la transition sélectionnée, et on rajoute les arêtes pour les points d'entrée et de sortie du programme.

Le graphe de flux de contrôle est donc une représentation statique du programme.

On remarque qu'on a légèrement modifié notre définition des graphes : le graphe est certes orienté, mais il admet possiblement des boucles, et il est possible d'annoter les arêtes avec des conditions.

De plus, un nœud spécifique est mis en avant : un sommet d'entrée qui est le point de départ du programme. D'autres nœuds de sorties peuvent être rajoutés.

À partir de ce graphe, on peut s'intéresser à des exécution du programme sous la forme du chemin sur en entrée spécifique.

Définition 3.54 : Chemin d'une exécution

Un **chemin d'exécution** est le chemin parcouru dans le graphe à partir d'un entrée donnée.

Le chemin d'une exécution est donc une représentation d'un comportement dynamique du programme, il s'agit de quelque chose lié à l'exécution du programme, et plus seulement à l'observation statique du programme en tant qu'élément de texte.

À partir d'un graphe de flux de contrôle d'un programme et de la notion de chemin d'exécution, on va pouvoir quantifier la qualité d'un ensemble de tests par sa capacité à couvrir un maximum de possibilités.

Définition 3.55 : Couverture de test

En se donnant un ensemble d'entrées, la **couverture de test** est la partie du graphe de flux de contrôle qui est visitée par les chemins d'exécution avec les entrées.

On peut quantifier cette couverture de tests en utilisant différentes métriques.

Définition 3.56 : Couverture de test pour les sommets

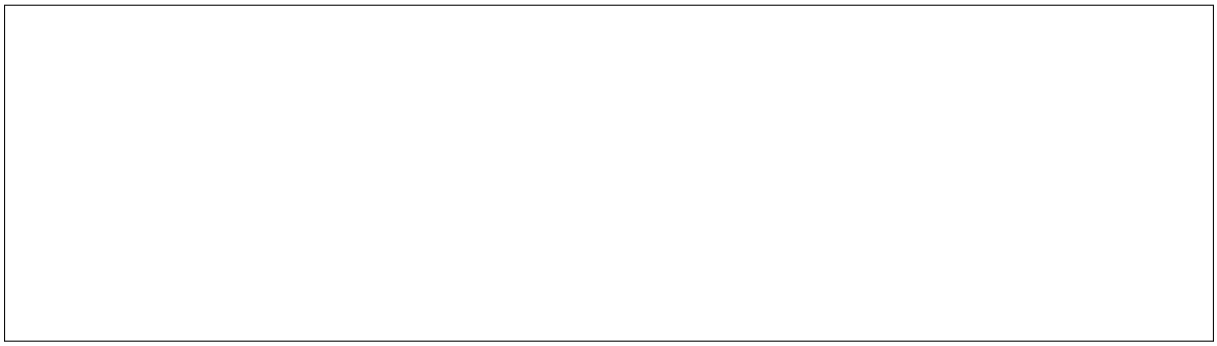
La **couverture de test pour les sommets d'un programme** est la partie des sommets qui est visitée par au moins un des tests.

Définition 3.57 : Couverture de test pour les arêtes

La **couverture de test pour les arêtes d'un programme** est la partie des arêtes qui est visitée par au moins un des tests.

Définition 3.58 : Couverture de test pour les chemins

La **couverture de test pour les chemins d'un programme** est la partie des chemins qui est visitée par au moins un des tests.



L'extension que nous avons faites des graphes pour les graphe de flux de contrôle (avoir un sommet qui sert de point de départ, et des sommets pour des points d'arrivée, autoriser les boucles), est un pas de plus vers les machines à états qui sont au programme de MPI.

La théorie des graphes est bien plus vaste que la recherche de chemin et les quelques graphes spécifiques que nous avons vus, et il reste de nombreux problèmes classiques qui seront vus en deuxième année, et encore d'autres problèmes classiques qui ne sont pas au programme de MPI.

Chapitre IV

Décomposition en sous-problèmes

Un des enjeux du programme n'est pas de maîtriser toutes les subtilités de l'algorithmique, mais d'avoir quelques cadres algorithmiques qui permettent d'analyser ou de proposer des solutions classique à des problèmes.

Dans ce chapitre, nous nous intéresserons à quatre classes d'algorithmes.

1 Algorithmes gloutons

1.1 Question d'optimisation résolue par des optimisations locales

Un problème en informatique est une manière de formaliser mathématiquement le cahier des charges pour un algorithme.

Définition 4.1 : Problème algorithmique

Un **problème algorithmique** est une question à laquelle on veut répondre par un algorithme sur un ensemble d'entrée possible.

L'idée principale derrière la question de problème est le fait qu'on veuille une méthode qui réponde dans *tous les cas*. Il ne suffit pas d'avoir un algorithme qui marche sur certains cas particulier, il faut pouvoir couvrir toutes les possibilités.

Exemple 4.1 : Quelques exemples de problèmes algorithmiques

- Étant donné un tableau en entrée, déterminer le maximum de ce tableau ;
- Étant donné un graphe et deux sommets, trouver un plus court chemin entre ces sommets ;
- Étant donné un graphe et deux sommets, déterminer s'il existe un chemin de longueur inférieur ou égale à l ;
- Proposer un algorithme qui permet de déterminer si un programme termine ou non.

Dans le cas des algorithmes gloutons, nous allons nous intéresser à un type de problèmes assez précis.

Définition 4.2 : Problème d'optimisation

Un **problème d'optimisation** est un problème dans lequel on cherche une solution qui est minimale (ou maximale selon les problèmes) pour une certaine caractéristique sur une entrée donnée.
Une solution qui atteint ce minimum (ou ce maximum) est appelé **solution optimale**.

Il y a donc deux caractéristiques : on cherche des solutions qui respectent des contraintes de structure, et en plus, on cherche parmi ces solutions, une solution qui minimise un coût (ou maximise un gain).

Par exemple, pour la recherche de chemins, il faut certes trouver des chemins, mais aussi minimiser la distance sur tous ces chemins.

Bien sûr, il existe parfois plusieurs solutions optimales, et parfois, quand l'infimum n'est pas atteint, il peut ne pas en exister du tout.

La notion de solution optimale est globale, mais on peut s'intéresser à la question des solutions localement optimales, et espérer qu'en faisant le choix qui semble le meilleur à chaque étape, obtenir la meilleure des solutions.

Définition 4.3 : Algorithme Glouton

Un **algorithme glouton** est un algorithme qui réalise à chaque étape le choix localement optimal dans la construction d'une solution pour un problème d'optimisation pour donner une solution globale.

Par exemple, l'algorithme de Dijkstra est un algorithme glouton qui fait la recherche d'un plus court chemin vers les autres sommets en, à chaque fois, prenant le sommet le plus proche vers lequel on sait qu'on a un plus court chemin.

Vous verrez plus de choses sur la notion de problème en informatique et sur leur classification en deuxième année. Dans la suite, nous nous concentrerons sur des exemples de problèmes que l'on peut résoudre par un algorithme glouton.

1.2 Problème du rendu de monnaie

On peut s'intéresser à un premier exemple classique des algorithmes gloutons, le rendu de monnaie.

Exemple 4.2 : Problème du rendu de monnaie

Le problème du rendu de monnaie est le suivant : on dispose de n de pièces de valeurs croissantes v_1, v_2, ..., v_n, et on dispose d'une valeur S à rendre.
L'objectif est de sélectionner un k-uplet i_1, ..., i_k de sorte à ce que $S = \sum_{j=1}^{k} v_{i_j}$, et de sorte à ce que k soit minimal.

Il s'avère que dans le système monétaire en euros, la méthode naïve est la meilleure : il suffit de rendre en priorité des pièces de plus grande valeur.

Algorithme 4.1 : Algorithme glouton pour le rendu de monnaie

Pour les valeurs des pièces $v_1 = 1$, $v_2 = 2$, $v_3 = 5$, $v_4 = 10$, l'algorithme suivant donne la solution optimale :
Entrée : S la somme à obtenir
Sortie : $i_1, \cdots, i_k$ les indices des pièces à utiliser

```
k ← 1
Pour Chaque j de 4 à 1 Faire
    Tant Que S ⩾ v_j Faire
        S ← S − v_j
        i_k ← j
        k ← k + 1

Renvoyer i_1, ···, i_k
```

La méthode gloutonne ne donne cependant pas toujours la bonne solution pour tous les systèmes de monnaies.

Proposition 4.1 : La méthode gloutonne n'est pas toujours optimale

Pour certaines valeurs des (v_j), la méthode gloutonne n'est pas optimale.

Exercice 4.1. Trouver un système monétaire dans lequel il y a un contre exemple pour la méthode gloutonne.

Vous verrez l'année prochaine que, parfois, même si une méthode gloutonne ne donne pas la solution optimale, on peut prouver qu'elle donne une approximation du résultat. On parle alors *d'heuristique gloutonne*.

1.3 Problème du sac à dos fractionnaire

Un des problèmes d'algorithmique les plus classiques est probablement le problème du sac à dos. Une des raisons principale est que le problème est *difficile* au sens de la complexité, ce que vous aurez le temps de voir l'année prochaine plus en détail, et que les méthodes pour résoudre ce problèmes ou pour s'y ramener sont souvent intéressantes en soi.

Définition 4.4 : Problème du sac à dos

Le **problème du sac à dos** est un problème d'optimisation qui travaille sur l'entrée suivante : on dispose d'un sac de taille T, et d'un ensemble de n objets de poids $(p_i)_{1\leqslant i\leqslant n}$ et de valeurs $(v_i)_{1\leqslant i\leqslant n}$.
L'objectif est de trouver un sous-ensemble $I \subset [\![1, n]\!]$ d'objets tel que l'on respecte la contrainte que le poids des objets choisis ne dépassent pas la taille du sac, et en maximisant la valeur totale des objets.

$$\begin{aligned} \text{Maximiser } & \sum_{i\in I} v_i \\ \text{Avec } & \sum_{i\in I} p_i \leqslant T \end{aligned}$$

Il existe de nombreuses variantes du problème du sac à dos, mais la plupart sont difficiles à résoudre.

Il n'y a donc pas d'espoir d'avoir une méthode gloutonne qui soit efficace. En effet, on serait tenté de prendre les objets par ordre de rentabilité.

Proposition 4.2 : Inefficacité de la méthode gloutonne

La méthode gloutonne de prendre les objets que l'on peut porter par ordre décroissant de $\frac{v_i}{p_i}$ ne donne pas nécessairement une solution optimale dans le problème du sac à dos.

Cependant, il existe une variante du problème du sac à dos pour laquelle il existe un algorithme glouton.

Définition 4.5 : Problème du sac à dos fractionnaire

Le **problème du sac à dos** est un problème d'optimisation qui travaille sur l'entrée suivante : on dispose d'un sac de taille T, et d'un ensemble de n objets de poids $(p_i)_{1\leqslant i\leqslant n}$ et de valeurs $(v_i)_{1\leqslant i\leqslant n}$.
L'objectif est de trouver un vecteur $(x_1,\cdots,x_n)$ de fraction des objets qui maximise la valeur, en respectant la contrainte que le poids total ne dépasse pas T.

$$\begin{aligned}\text{Maximiser } &\sum_{1\leqslant i\leqslant n} x_i v_i\\ \text{Avec } &\sum_{1\leqslant i\leqslant n} x_i p_i \leqslant T\\ &\forall 1\leqslant i\leqslant n,\ 0\leqslant x_i\leqslant 1\end{aligned}$$

Il s'agit d'une relaxation continue du problème du sac à dos : au lieu d'avoir une contrainte combinatoire de présence ou non de l'objet, on peut se permettre de prendre une fraction de l'objet.
Cela simplifie beaucoup le problème au point que l'on peut trouver une solution rapidement.

Algorithme 4.2 : Algorithme glouton pour le sac à dos fractionnaire

Quitte à prendre les objets de poids nul, on peut supposer que pour tout i, $p_i > 0$. On suppose par ailleurs que l'on dispose des objets dans l'ordre décroissant de $\frac{v_i}{p_i}$, et on a donc :

$$\frac{v_1}{p_1}\geqslant\frac{v_2}{p_2}\geqslant\cdots\geqslant\frac{v_n}{p_n}$$

.
L'algorithme suivant donne une solution optimale :
Entrée : $p_1,\cdots,p_n$ le poids des objets, $v_1,\cdots,v_n$ la valeur des objets, T la capacité du sac
Sortie : $(x_1,\cdots,x_n)$ une solution optimale
  $(x_1,x_2,\cdots,x_n)\leftarrow(0,0,\cdots,0)$
  Pour Chaque i de 1 à n **Faire**
    $x_i\leftarrow\min(1,\frac{T}{p_i})$
    $T\leftarrow T-x_i p_i$
  Renvoyer $(x_1,\cdots,x_n)$

Nous reverrons cette histoire de sac à dos plus tard dans le chapitre.

2 Diviser pour régner

Les méthodes gloutonnes incrémentent la taille de la solution itérativement, mais il est possible à la place de construire une solution à partir d'une décomposition en plusieurs sous-problèmes indépendants de taille arbitraire.

2.1 Principe du diviser pour régner

Définition 4.6 : Algorithme diviser pour régner

Un algorithme **diviser pour régner** est composé de trois étapes :
- La **séparation** (ou **division**) qui consiste à séparer un problème en sous-problèmes ;
- Le **règne** qui consiste à traiter les sous-problèmes indépendamment les uns des autres ;
- La **réunion** qui consiste à combiner les solutions des sous-problèmes pour construire la solution du problème.

Le fait que les problèmes soient indépendants est très important pour les algorithmes diviser-pour-régner.

Tout comme pour la récursivité, il est nécessaire d'avoir une codition d'arrêt pour pouvoir s'arrêter.

Même si les algorithme diviser-pour-règner sont souvent implémentés de manière récursives, le fait d'être une implémentation récursive est une caractéristique du programme et non de l'algorithme.

Il existe de nombreux algorithmes de type diviser pour régner.

D'une certaine manière, la recherche dichotomique est un algorithme de diviser-pour-régner qui décompose le problème en un seul cas plus petit. On parle parfois de *diminuer-pour-régner*.

2.2 Multiplication rapide de deux nombres : Algorithme de Karatsuba

On cherche à utiliser une méthode diviser-pour-régner pour calculer la multiplication de deux entiers positifs à n chiffres.
On propose l'algorithme suivant.

Algorithme 4.3 : Multiplication naïve en diviser-pour-régner

Entrée : $x = \sum_{0\leqslant i\leqslant n-1} a_i 2^i$ un entier, $y = \sum_{0\leqslant i\leqslant n-1} b_i 2^i$
Sortie : xy
  Si $n = 1$ **Alors**
    Renvoyer $a_0 b_0$
  Sinon
    On écrit $x = x_1 2^{\lfloor \frac{n}{2} \rfloor} + x_0$ et $y = y_1 2^{\lfloor \frac{n}{2} \rfloor} + y_0$
    $z_3 \leftarrow x_1 y_1$ calculé récursivement.
    $z_2 \leftarrow x_0 y_1$ calculé récursivement.
    $z_1 \leftarrow x_1 y_0$ calculé récursivement.
    $z_0 \leftarrow x_0 y_0$ calculé récursivement.
    Renvoyer $z_3 2^{2\lfloor \frac{n}{2} \rfloor} + (z_1 + z_2) 2^{\lfloor \frac{n}{2} \rfloor} + z_0$

Ici, l'étape de séparation est le découpage en $x = x_1 2^{\lfloor \frac{n}{2} \rfloor} + x_0$ et $y = y_1 2^{\lfloor \frac{n}{2} \rfloor} + y_0$, l'étape de reigne est le calculs récursif des z_i, et l'étape de réunion sont les additions et les décalages pour obtenir le résultat.

Souvent, dans les algorithmes de type diviser-pour-régner, la complexité n'est pas aussi immédiate que les algorithmes plus classiques, et il faut donc travailler un peu pour obtenir une formule de récurrence.

Il s'avère que la complexité obtenue n'est pas meilleure que la complexité d'une méthode classique en travaillant sur tous les chiffres.
Pire encore, la pile des appels nous donne une complexité spatiale qui est moins bonne.

Proposition 4.3 : Complexité de la méthode naïve

Dans le pire des cas, la méthode naïve a une complexité temporelle en $O(n^2)$, et une complexité spatiale en $O(n^2)$.

Mais il s'avère que l'on peut faire mieux : si l'on arrive à se raccrocher à 3 multiplications au lieu de quatre, on va pouvoir gagner en performances.

En effet, on peut remarquer que :

$$\begin{aligned} xy &= (x_1 2^{\lfloor \frac{n}{2} \rfloor} + x_0)(y_1 2^{\lfloor \frac{n}{2} \rfloor} + y_0) \\ &= x_1 y_1 2^{2\lfloor \frac{n}{2} \rfloor} + (x_1 y_1 + x_0 y_0 - (x_1 - x_0)(y_1 - y_0)) 2^{\lfloor \frac{n}{2} \rfloor} + x_0 y_0 \end{aligned}$$

Cela nous permet de n'avoir que trois multiplications à réaliser, et cela nous donne l'algorithme de Karatsuba.

Algorithme 4.4 : Algorithme de Karatsuba

Entrée : $x = \sum_{0 \leqslant i \leqslant n-1} a_i 2^i$ un entier, $y = \sum_{0 \leqslant i \leqslant n-1} b_i 2^i$
Sortie : xy
  Si $n = 1$ **Alors**
    Renvoyer $a_0 b_0$
  Sinon
    On écrit $x = x_1 2^{\lfloor \frac{n}{2} \rfloor} + x_0$ et $y = y_1 2^{\lfloor \frac{n}{2} \rfloor} + y_0$
    $z_4 \leftarrow (x_1 - x_0)(y_1 - y_0)$ calculé récursivement.
    $z_3 \leftarrow x_1 y_1$ calculé récursivement.
    $z_0 \leftarrow x_0 y_0$ calculé récursivement.
    Renvoyer $z_3 2^{2\lfloor \frac{n}{2} \rfloor} + (z_3 + z_0 - z_4) 2^{\lfloor \frac{n}{2} \rfloor} + z_0$

L'étape de séparation est toujours la même, mais les étapes de reigne et de réunion sont différentes pour n'avoir que trois multiplications, et la réutilisation de z_3 et z_0 pour calculer le coefficient devant $2^{\lfloor \frac{n}{2} \rfloor}$.

Avec l'algorithme de Karatsuba, la complexité est meilleure.

Proposition 4.4 : Complexité temporelle de l'algorithme de Karatsuba

La complexité temporelle de l'algorithme de Karatsuba est en $O(n^{\log_2 3})$.

2.3 Tri fusion

Il existe plusieurs algorithmes de tri diviser-pour-regner, mais l'un des plus simple est le tri fusion.

Algorithme 4.5 : Tri Fusion

Le **tri fusion** est un algorithme de tri de tableau de type diviser-pour-regner.

Entrée : T un tableau à trier
Sortie : Le tableau T' trié qui contient les mêmes éléments que T.
  Si le tableau T est de taille 1 ou moins **Alors**
    Renvoyer T
  $T_1, T_2 \leftarrow$ la décomposition de T en deux parties dont la taille diffère d'au plus 1.
  Trier T_1 et T_2 en T_1' et T_2'.
  Construire T' à l'aide de T_1' et T_2'.
  Renvoyer T'

Le tri-fusion repose sur le fait que faire la fusion de deux tableaux triés est relativement facile, et il est même très facile quand on travaille sur des listes.

Exemple 4.3 : Procédé de réunion en OCaml pour les listes

```
let rec reunion_liste l1 l2 = match l1, l2 with
| [], _ -> l2
| _, [] -> l1
| p::q, s::t ->
    if p< s then
        p::(reunion_liste q l2)
    else
        s::(reunion_liste l1 t)
```

La séparation est cependant un peu plus délicate, il faut dans un premier temps mesurer la taille de la liste pour faire la séparation.

Exemple 4.4 : Procédé de séparation pour les listes

On utilise une fonction auxiliaire qui découpe à un indice donné

```
let separation l =
    let rec aux l n acc = match l with
    | _ when n = 0 -> (List.rev acc), l
    | [] -> failwith "Mauvais indice dans la separation"
    | p::q -> aux q (n-1) (p::acc)
    in
    let n = List.length l in
    aux l (n/2) []
```

À partir de ces deux fonctions, on peut donc écrire une fonction pour réaliser le tri fusion en OCaml sur des listes.

Exemple 4.5 : Tri fusion en OCaml

```
let rec tri_fusion l = match l with
    | [] -> []
    | [a] -> [a]
    | _ ->
        let l1, l2 = separation l in
        let l1_triee = tri_fusion l1 in
        let l2_triee = tri_fusion l2 in
        reunion_liste l1_triee l2_triee
```

On peut ensuite analyser la complexité de cette implémentation.

Proposition 4.5 : Complexité du tri fusion

La complexité du tri fusion est en $O(n \log n)$ où n est la longueur de la liste en entrée.

Bien sûr, on aurait pu travailler sur des tableaux au lieu de travailler sur des listes. On peut gagner en performance sur l'espace occupé par les tableaux triés de manière intermédiaire.

2.4 Tri rapide

Pour certains algorithme, la séparation ne se fait pas nécessairement en deux tableaux de taille égales. C'est le cas du tri rapide.

Le tri rapide fonctionne sur le principe d'un pivot : un élément du tableau est choisi, et les autres éléments sont répartis à droite et à gauche du pivot en fonction de s'ils sont plus grands ou plus petits que le pivot.

Algorithme 4.6 : Tri rapide

Entrée : un tableau T à trier
$p \leftarrow$ un élément du tableau T
Mettre tous les éléments inférieurs à p dans la partie gauche de T, et tous les éléments supérieurs à p dans la partie droite
Trier ces deux tableaux.

L'étape importante est le fait de placer le pivot dans le tableau, on commence par implémenter une fonction qui réalise le déplacement des éléments.

Exemple 4.6 : Implémentation en C de la répartition autour du pivot

La fonction prend en argument l'indice du début du tableau, ainsi que l'indice de la fin du tableau, et renvoie l'indice du pivot à la fin de la répartition.
Le pivot sélectionné est le premier élément du tableau.

```c
void inverser(int t[], int i, int j){
    int temporaire = t[i];
    t[i]=t[j];
    t[j]=temporaire;
}

int pivoter(int t[], int debut, int fin){
    int pivot;
    int i = debut;
    int j = fin;
    pivot = t[debut];
    while (i<j){
        if (t[i+1]<=pivot){
            inverser(t, i+1, i);
            i++;
        }
        else{
            inverser(t, i+1, j);
            j--;
        }
    }
    return i;
}
```

À partir de là, on peut réaliser l'implémentation du tri rapide en C.

Exemple 4.7 : Implémentation du tri rapide en C

```c
void tri_rapide_recursif(int t[], int debut, int fin){
    int indice_pivot;
    if (fin - debut<=1){
        return;
    }
    indice_pivot = pivoter(t, debut, fin);
    tri_rapide_recursif(t, debut, indice_pivot - 1);
    tri_rapide_recursif(t, indice_pivot + 1, fin);
}
void tri_rapide(int t[], int taille){
    tri_rapide_recursif(t, 0, taille-1);
}
```

Malheureusement, dans le pire des cas, le tri rapide ne nous donne pas une bonne complexité.

Proposition 4.6 : Complexité dans le pire des cas du tri rapide

Dans le pire des cas, le tri rapide s'effectue en $O(n^2)$ où n est la taille du tableau à trier.

Il existe un procédé pour se ramener à du $O(n \log n)$ dans le pire des cas, et cela revient souvent à bien choisir le pivot pour qu'il soit proche de la médiane.

Exercice 4.2. Si on considère qu'il est possible de trouver la médiane d'un tableau en temps linéaire, quelle est la complexité dans le pire des cas du tri rapide ?

Sinon, il est possible de se contenter du cas moyen qui s'avère être intéressant.

Proposition 4.7 : Complexité dans le cas moyen du tri rapide

Sur un tableau dont les éléments sont dans un ordre aléatoire, le tri rapide s'effectue en moyenne en $O(n \log n)$.

Il est alors intéressant de prendre le pivot au hasard de sorte à éviter les pires des cas qui surviennent lorsque le tableau en entrée est presque trié.

3 Programmation dynamique

La programmation dynamique répond au souci principal des algorithmes diviser-pour-régner qui ne peuvent pas résoudre des sous-problèmes inter-dépendants en se souvenant des solutions à tous sous-problèmes.

Cette méthode est très proche du procédé de mémoïsation qui consiste à se souvenir des résultats intermédiaire au fur et à mesure du calcul.

Il s'agit d'un paradigme qui s'intéresse tout particulièrement aux problèmes d'optimisation.

Dans un premier temps, nous allons nous intéresser au procédé de mémoïsation.

3.1 De la mémoïsation à la programmation dynamique

Un des points centrales de la programmation dynamique est l'utilisation de différentes solutions, et il faut donc utiliser un principe de mémorisation pour se souvenir des différentes solutions.

Exemple 4.8 : Fibonacci avec la mauvaise complexité

La fonction suivante est de complexité temporelle exponentielle en n, il y a plusieurs appels récursifs à la fonction avec les mêmes arguments.

```
let rec fibo n = match n with
| 0 -> 0
| 1 -> 1
| _ -> fibo (n-1) + fibo (n-2)
```

Par ailleurs sa complexité spatiale est en $O(n)$.

Dans le cas des fonctions qui n'ont pas d'effet de bord, on peut utiliser le principe de mémoïsation qui permet de réutiliser les résultats précédents.

Définition 4.7 : Mémoïsation

La **mémoïsation** est le procédé de mémorisation des résultats d'une fonction sur ses entrées pour pouvoir retourner plus facilement le résultat.

La mémoïsation ne permet pas de traiter les fonctions qui ont des effets de bord, mais permet de traiter les fonctions qui renvoie une valeur.

Exemple 4.9 : Suite de Fibonacci avec mémoïsation

La fonction suivante est de complexité temporelle et spatiale linéaire en $O(n)$.

```
let fibo n =
    let memo = Array.make None (n+2) in
    memo.(0) <- Some 0 ; memo.(1) <- Some 1 ;
    let rec fibo_rec n = match memo.(n) with
    | Some x -> x
    | None ->
        let res = fibo_rec (n-1) + fibo_rec (n-2) in
        memo.(n)<- res ; res
    in fibo_rec n
```

On a ici rajouté un surcoût en mémoire (qui ne change cependant pas l'ordre de grandeur asymptotique), pour un gain temporel cette fois-ci très important.

Il s'agit d'une question de programmation dynamique,

Mais on peut faire mieux si on considère les problèmes dans le bon sens, on n'a pas besoin de se souvenir de toutes les solutions à la fois.

Exemple 4.10 : Suite de Fibonacci en complexité constante spatiale

La fonction suivante est de complexité spatiale constante.

```
int fibo(int n){
    int a = 0;
    int b = 1;
    for (int i = 0; i<n ;i++){
        int c = b;
        b = a + b;
        a = c;
    }
    return a;
}
```

On peut par ailleurs utiliser la récursivité terminale pour avoir une complexité spatiale constante, et procéder de manière récursive.

```
let fibo n =
    let rec fibo_aux a b n =
        if n = 0 then a
        else fibo_aux b (a+b) (n-1)
    in fibo_aux 0
```

La notion de mémoïsation se perd un peu, car il ne s'agit plus toujours des entrées d'une fonction, mais il s'agit toujours de construire des solutions à partir d'autres solutions : dans a et b on se souvient de la solution sur des sous-problèmes consécutifs.

Définition 4.8 : Programmation Dynamique

La méthode de **programmation dynamique** est un procédé qui consiste à calculer les solutions de problèmes plus petits pour calculer un problème plus grand.

La différence principale avec la programmation en diviser pour régner est le fait que les sous-problèmes peuvent être interdépendants : on peut avoir la nécessité de se souvenir de plusieurs problèmes.

Exemple 4.11 : L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique

L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique, en effet, on calcule les $M^{(k)}$ à partir des précédent.

$$m_{i,j}^{(k)} = \min\left(m_{i,j}^{(k-1)}, m_{i,k}^{(k-1)} + m_{k,j}^{(k-1)}\right)$$

Ici, on n'a pas besoin de se souvenir de toutes les matrices $M^{(k)}$, il suffit de se souvenir de la dernière, et même parmi les éléments qui sont dans la dernière matrice, ils ne seront pas nécessairement utilisés à la fin, mais sont nécessaire pour savoir s'ils sont importants dans l'étape suivante.

3.2 Chemin de valeur maximale dans une matrice

Nous allons nous intéresser à un problème d'optimisation.

Définition 4.9 : Chemin de valeur maximale dans une matrice

Le problème du **chemin de valeur maximale dans une matrice** est un problème pour lequel, à partir d'une matrice en entrée de taille $n \times n$ d'éléments $m_{i,j}$, on cherche un chemin de l'élément d'indice $1,1$ à l'élément d'indice n,n en ne faisant que des mouvements d'une case vers la droite, ou d'une case vers la gauche.
On cherche donc une suite $(i_1, j_1), \cdots (i_{2n-1}, j_{2n-1})$ telle que :

$$\begin{aligned} \text{Maximiser } & \sum_{k=1}^{2n-1} m_{i_k,j_k} \\ \text{Avec } & \forall k, i_k, j_k \in [\![1,n]\!] \\ & (i_1, j_1) = 1,1 \\ & (i_{2n-1}, j_{2n-1}) = n,n \\ & \forall k, (i_{k+1}, j_{k+1}) \text{ est égal à } (i_k+1, j_k) \text{ ou } (i_k, j_k+1). \end{aligned}$$

On peut explorer toutes les possibilités à l'aide d'une exploration complète de toutes les possibilités.

Exemple 4.12 : Implémentation en utilisant une exploration de toutes les possibilités

```
let longueur_chemin_maximum (m: int array array) :int =
    let n = Array.length m in
    let rec explorer i j = m.(i).(j) + match i=n-1, j = n-1 with
    | true, true -> 0
    | _, true -> explorer (i+1) j
    | true, _ -> explorer i (j+1)
    | _, _ -> max (explorer (i+1) j)
                  (explorer i (j+1))
    in explorer 0 0
```

Il s'avère que cette méthode n'est pas bonne.

Proposition 4.8 : Complexité de la méthode naïve

La complexité temporelle de cette fonction est en $O\left(\binom{2n}{n}\right)$.
La complextié spatiale de cette fonction est en $O(n^2)$.

Or, l'approximation de Stirling nous dit que $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$.

On peut se servir de la programmation dynamique pour faire mieux, au lieu de chercher tous les chemins, il nous suffit de prendre le meilleur des chemins depuis une certaine case.
En effet, le meilleur chemin suffit à calculer les autres meilleures chemins.

Proposition 4.9 : Construction d'une solution à partir des sous-solutions

On remarque que, en notant $m_{i,j}$ les coefficients de la matrice, et $l_{i,j}$ la longueur du chemin maximal depuis l'éléments d'indice (i,j), on a :

$$l_{i,j} = \begin{cases} m_{n,n} & \text{si } i = j = n, \\ m_{i,j} + l_{i,j+1} & \text{si } i = n, \\ m_{i,j} + l_{i+1,j} & \text{si } j = n, \\ m_{i,j} + \max(l_{i,j+1}, l_{i+1,j}) & \text{sinon.} \end{cases}$$

Exemple 4.13 : Implémentation en utilisant la programmation dynamique

```
let longueur_chemin_maximum (m:int array array) : int =
    let n = Array.length m in
    let memo = Array.make_matrix n n None in
    let rec aux i j = match memo.(i).(j) with
    | Some x -> x
    | None ->
        let res = m.(i).(j) + match i=n-1, j = n-1 with
        | true, true -> 0
        | _, true -> aux (i+1) j
        | true, _ -> aux i (j+1)
        | _, _ -> max (aux (i+1) j) (aux i (j+1))
        in memo.(i).(j)<-res ; res
    in aux 0 0
```

Cette implémentation est bien meilleure en terme de complexité.

Proposition 4.10 : Complexité de l'implémentation en programmation dynamique

La complexité temporelle de cette implémentation est en $O(n^2)$.
La complexité spatiale de cette implémentation est en $O(n^2)$.

Ici, on a utilisé la récursivité pour cacher l'ordre dans lequel les éléments sont calculés en pratique, cependant, on peut être intéresser pour voir de quelles valeurs on a besoin pour réaliser le calcul.

Exercice 4.3. Implémenter en C la fonction précédente de manière itérative en exploitant un ordre des éléments à traiter.

Par ailleurs, on peut construire une matrice des directions à prendre pour pouvoir retrouver le chemin à réaliser.

Exemple 4.14 : Retrouver le chemin maximum

On se donne un type pour savoir s'il faut aller à droite ou en bas.

```
type direction = Droite | Bas
```

On peut

```
let longueur_chemin_maximum (m:int array array) : (direction array array) =
    let n = Array.length m in
    let memo = Array.make_matrix n n None in
    let direction = Array.make_matrix n n Bas in
    let rec aux i j = match memo.(i).(j) with
    | Some x -> x
    | None ->
        let res =  m.(i).(j) + match i=n-1, j = n-1 with
        | true, true -> 0
        | _, true -> direction.(i).(j) <- Bas ; aux (i+1) j
        | true, _ -> direction.(i).(j) <- Droite ;aux i (j+1)
        | _, _ ->
            let res_bas = (aux (i+1) j) in
            let res_droite = (aux i (j+1)) in
            if res_bas >res_droite then
                (direction.(i).(j)<- Bas ; res_bas)
            else
                (direction.(i).(j)<- Droite ; res_droite)
        in memo.(i).(j)<-res ; res
    in aux 0 0
```

On peut ensuite reconstruire le chemin à partir de là.

```
let reconstruire (direction : direction array array) : (int * int) list =
    let n = Array.length direction in
    let rec aux i j acc =
        if (i = j) && (j = n -1) then acc
        else match direction.(i).(j) with
        |Droite -> aux i (j+1) ((i, j+1)::acc)
        |Bas -> aux (i+1) j ((i+1, j)::acc)
    in List.rev (aux 0 0 [(0,0)])
```

3.3 Distances d'édition

Nous allons nous intéresser à une notion de distance sur les mots, et voir que la programmation dynamique apporte une solution efficace au problème.

Nous allons nous intéresser à la notion de mot en informatique.

Définition 4.10 : Mot sur un alphabet

Soit $\Sigma = \{a_1, \cdots, a_n\}$ un ensemble fini de symboles que l'on appelle **alphabet**.
On considère l' **ensemble des mots** sur Σ, noté Σ^*, l'ensemble des suites finies dont les valeurs sont dans Σ.
Le mot vide est noté ϵ et sa longueur est 0, et pour un mot non vide, il est noté par la concaténation des symboles $a_{i_1}a_{i_2}\cdots a_{i_k}$, et on appelle k la longueur de ce mot.

Vous étudierez plus en détail les *langages*, c'est-à-dire les sous-ensembles de Σ^*, plus en détail l'année prochaine.

Pour l'instant, on se concentre sur les mots, et en particulier sur la notion de préfixe et de suffixe.

Définition 4.11 : Préfixe et suffixe

Soit un mot $u = a_1 \cdots a_n$ de longueur l.
Le **suffixe** de longueur $k \leqslant n$ est le mot $a_{n-k+1}a_{n-k+2}\cdots a_n$ constitué des k dernières lettres de u.
Le **préfixe** de longueur $k \leqslant n$ est le mot $a_1a_2\cdots a_k$ constitué des k premières lettres de u.

Exercice 4.4. Quel est l'ensemble des préfixe et suffixe de du mot *abaabba* ?

Nous cherchons à caractériser une distance entre deux mots. L'idée est la suivante, on veut compter le nombre d'opérations à réaliser pour passer d'un mot à un autre.

Définition 4.12 : Opération sur les mots

Une **insertion** est la transformation d'un mot $a_0\cdots a_i a_{i+1}\cdots a_k$ à un mot $a_0\cdots a_i b a_{i+1}\cdots a_k$.
Une **suppression** est la transformation d'un mot $a_0\cdots a_{i-1}a_i a_{i+1}\cdots a_k$ à un mot $a_0\cdots a_{i-1}a_{i+1}\cdots a_k$.
Une **substitution** est la transformation d'un mot $a_0\cdots a_{i-1}a_i a_{i+1}\cdots a_k$ à un mot $a_0\cdots a_{i-1}b a_{i+1}\cdots a_k$.

Définition 4.13 : Distance d'édition

La **distance d'édition** entre deux mots u et v est le nombre minimum d'opération qu'il faut pour passer d'un mot à un autre.

Exercice 4.5. Quelle est la distance d'édition entre ROUGE et BLEU ?

Il existe des variantes des distances d'édition où on peut rajouter ou retirer des opérations possibles. Ici, il s'agit de la distance de Levenshtein [1] qui est la plus classique.

Pour calculer la distance d'édition, on va utiliser la programmation dynamique en enregistrant dans une matrice.

Proposition 4.11 : Formule récursive de la distance d'édition

Pour $u = u_1u_2\cdots u_{|u|}$ et $v = v_1v_2\cdots v_{|v|}$ deux mots, on note $d(u,v)$ la distance d'édition entre ces deux mots.
Par ailleurs, on note $u[1:]$ le suffixe de u de longueur $|u|-1$ qui contient toutes les lettres de u dans l'ordre sauf la première.
On a la relation suivante :

$$d(u,v) = \begin{cases} |u| & \text{si } |v| = 0, \\ |v| & \text{si } |u| = 0, \\ d(u[1:], v[1:]) & \text{si } u_1 = v_1, \\ 1 + \min(d(u[1:], v[1:]), d(u, v[1:]), d(u[1:], v)) & \text{sinon.} \end{cases}$$

Au lieu de faire un calcul récursif à partir de la valeur à calculer, on peut utiliser une matrice de sorte à ce qu'à l'indice i, j, ait la distance entre le suffixe de u de longueur $|u| - i + 1$ et le suffixe de v de longueur $|v| - j + 1$.

1. https://fr.wikipedia.org/wiki/Distance_de_Levenshtein

Exemple 4.15 : Implémentation du calcul de la distance d'édition en OCaml

On se donne une fonction pour calculer le minimum de trois éléments.

```
let min3 x y z =
    min x (min y z)
```

On s'en sert pour calculer la distance d'édition. On commence les indices à 0, et le suffixe considéré à l'indice i est donc le suffixe dans lequel les i premières valeurs ont été retirées.
On ne mets pas dans la matrice les éléments d'indice $|u|$ et $|v|$ car le résultat peut être calculer directement.

```
let distance_edition s1 s2 =
    let n = String.length s1 in
    let m = String.length s2 in
    let memo = Array.make_matrix n m None in
    let rec aux i j = match i = n, j = m in
    | true, true -> 0
    | true, _ -> m-j
    | _, true -> n-i
    | _ -> match memo.(i).(j) with
        | Some x -> x
        | _ -> let res =
                if s1.[i] = s2.[j] then aux (i+1) (j+1)
                else min3 (aux (i+1) j) (aux i (j+1)) (aux (i+1) (j+1))
            in memo.(i).(j) <- Some res ; res
```

Proposition 4.12 : Complexité spatiale et temporelle avec la programmation dynamique

Cette implémentation a une complexité spatiale et temporelle en $O(|u||v|)$.

Mais on peut faire mieux !
En choisissant correctement l'ordre dans lequel on fait les calcul, on peut ne se souvenir que d'une partie des éléments.

Proposition 4.13 : Complexité spatiale et temoprelle avec la programmation dynamique

Il est possible d'implémenter la distance d'édition avec une complexité temporelle en $O(|u||v|)$ et une complexité spatiale en $O(\min(|u|,|v|))$.

Il existe une grande variété de problèmes qui peuvent être résolus par programmation dynamique. Nous verrons quelques exemples en exercice, ainsi que quelques variantes d'exercices que nous avons déjà vus.

4 Recherche exhaustive

Une dernière catégorie d'algorithme sont les algorithmes plus brutaux de la recherche exhaustive : on ne peut parfois pas se contenter d'une partie très réduite des possibilités, et il faut parfois explorer une grande partie de l'espace.

Nous allons tout de même essayer de diminuer le nombre de possibilités à étudier à l'aide des contraintes imposées par un problème.

4.1 Problème de satisfaction de contraintes

Un type de problème qui va nous intéresser spécifiquement dans cette partie sont les problèmes de satisfaction de contrainte.

Définition 4.14 : Problème de satisfaction de contraintes

Un **problème de satisfaction de contraintes** est un type de problème où on cherche un objet qui vérifie un certain nombre de contrainte en entrée.

L'idée est que la liste des contraintes sont spécifiées par l'entrée (dans un certains format) et qu'il faudra respecter ces contraintes si possible.

Exemple 4.16 : Problèmes de satisfaction de contraintes

- À partir d'un graphe en entrée, et d'un nombre de couleur, déterminer s'il existe un coloriage de ce graphe avec ces couleurs sans que deux sommets adjacents soient de même couleur ;
- Trouver un mot à poser au Scrable à partir d'un dictionnaire, de lettres, et d'une grille sans prendre en compte les points ;
- Le jeu de Futoshiki [a] .

a. https://en.wikipedia.org/wiki/Futoshiki

Le fait que les contraintes soient restrictives vont nous poser problème pour trouver une solution car nous avons moins d'espoir de trouver facilement parmi toutes les possiblités, mais nous allons voir une méthode qui permet justement d'utiliser les contraintes pour ne pas tout tester.

4.2 De la force brute au retour sur trace

Nous allons nous intéresser au jeu du Sudoku, et voir comment passer d'une approche par force brute qui teste sans remord toutes les possibilités à une approche un peu plus subtile qui s'acharne un peu moins dans les directions qui ne sont pas valides.

Définition 4.15 : Grille de Sudoku

Une **Grille de Sudoku** est un tableau de taille 9×9 qui contient des nombres de 1 à 9 de sorte à ce que :

- Il n'y ait pas de doublons sur une même colonne ;
- Il n'y ait pas de doublons sur une même ligne ;
- Il n'y ait pas de doublons sur l'un des 9 sous-blocs de taille 3×3.

Exemple 4.17 : Grille de Sudoku

La grille suivante est une grille de Sudoku :

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Mais les suivantes n'en sont pas :

2	5	6	7	3	8	9	4	1
3	1	8	2	9	4	5	6	7
4	9	7	1	5	6	2	3	8
1	3	5	2	4	6	7	8	9
2	6	7	5	9	8	1	3	4
8	4	9	1	7	3	2	5	6
1	4	5	3	6	8	9	7	2
2	7	6	1	4	9	5	8	3
9	8	3	2	5	7	6	1	4

2	5	4	6	3	7	9	8	1
3	1	2	5	7	4	8	6	9
4	3	7	1	5	6	2	9	8
1	6	5	2	8	9	3	4	7
7	2	6	4	9	8	1	3	5
5	4	8	9	1	3	7	2	6
8	9	1	3	6	5	4	7	2
6	7	9	8	4	2	5	1	3
9	8	3	7	2	1	6	5	4

Définition 4.16 : Problème du Sudoku

Un **Problème du Sudoku** est un problème de satisfaction de contrainte qui, à partir d'une grille incomplète, cherche à compléter la grille en une grille de Sudoku.

Exercice 4.6. Résoudre la grille suivante :

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5

Le problème est qu'on ne peut pas toujours juste rajouter des éléments dont on est sûr, parfois, il faut prendre des prises de risques et faire des supposition.

D'un point de vue informatique, il y a plusieurs manière de procéder, et l'une d'entre elle est de tout tester.

Définition 4.17 : Recherche par force brute

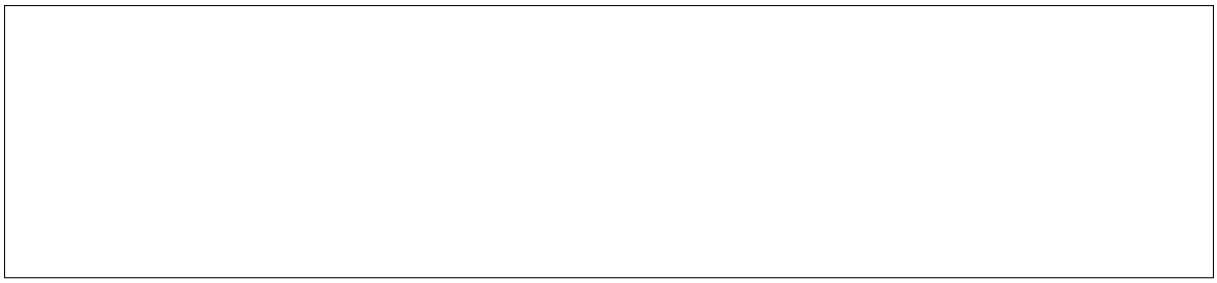
Une **recherche par force brute** est un algorithme qui teste toutes les possibilités et vérifie pour chaque possibilité si elle vérifie la propriété attendue.

Le résultat sera le bon, mais on n'aura pas de bonnes performances. On n'a pas exploité le fait que l'on pouvait, dès que l'on arrive sur une supposition qui contredit l'une des contrainte, dire que le résultat n'était pas le résultat attendu.

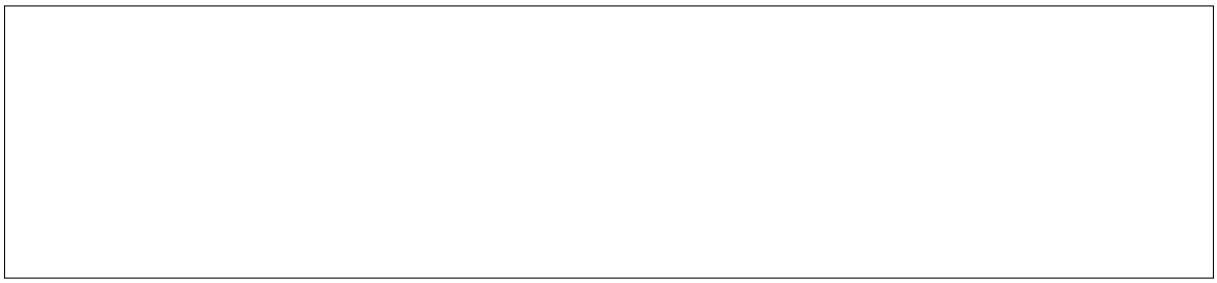
Le principe de retour-sur-trace fait exactement cela : on fait des suppositions au fur et à mesure de l'exploration des possibilités, mais on revient en arrière dès que l'on arrive sur un cas d'erreur.

Définition 4.18 : Recherche par retour sur trace

Un algorithme **retour sur trace** est un algorithme qui construit pas à pas une solution à un problème de satisfaction de contraintes en faisant des supposition à chaque fois jusqu'à arriver à une solution ou à une contradiction avec les contraintes du problèmes.
Dans le cas d'une contradiction, l'algorithme revient sur ses pas de sorte à pouvoir faire d'autres supposition.



On peut par ailleurs visualiser l'exploration des possibilités sous la forme d'un arbre, et c'est très exactement ce que nous faisons, un parcours d'arbre.



Dès lors, on peut implémenter la solution en explorant toutes les possibilités.

Exemple 4.18 : Implémentation de l'exploration par retour sur trace pour le problème du Sudoku

La fonction proposée renvoie un booléen : `true` si une solution a été trouvée dans l'exploration, `false` sinon.
Initialement, les éléments de la grille contiennent 0 si aucune contrainte n'a été imposée, et un nombre de 1 à 9 sinon.
On se donne une fonction qui permet de vérifier si la grille est valide avec la nouvelle case en i et en j.

```
bool est_valide_actuellement(int grille[9][9], int i, int j){
    for (int k = 0; k<9;k++){
        if (k!=j && grille[i][j]==grille[i][k])
            return false;
        if (k!=i && grille[i][j]==grille[k][j])
            return false;
    }

    for (int k = 0; k<3;k++){
        for (int l = 0; l<3;l++){
            int ip = (i/3)*3+k;
            int jp = (j/3)*3+l;
            if (ip!=i || jp!=j){
                if (grille[ip][jp]==grille[i][j]){
                    return false;
                }
            }
        }
    }
    return true;
}
```

On peut définir une fonction d'exploration qui s'intéresse à une case dont les indices sont données en supposant que tous les éléments avant ont déjà été traités.

```
bool explorer(int grille[9][9], int i, int j){
    if (j>=9)
        {j = 0; i = i + 1;}
    if (i>=9)
        return true;
    if (grille[i][j]==0){
        for (int val = 1; val<10; val++){
            grille[i][j] = val;
            if (est_valide_actuellement(grille, i, j)){
                if (explorer(grille, i, j+1))
                    return true;
            }
        }
        grille[i][j]=0;
        return false;
    }
    return explorer(grille, i, j+1);
}
```

Enfin, on peut présenter une fonction qui applique la précédente à partir de 0.

```
void remplir(int grille[9][9]){
    if (!explorer(grille, 0, 0)){
        printf("Grille non valide");
        abort();
    }
    return();
}
```

On a eu besoin de revenir en arrière lors des étape de retour, il faut rectifier la supposition que l'on avait faite.

4.3 Problème des huits reines

Dans le problème précédent, on a du rectifier les modifications des suppositions qui s'avéraient être eronnées, on a utilisé la pile pour pouvoir se souvenir des modifications que l'on avait réalisé.

Cependant, il est possible de faire un petit peu plus simple dans le cas où on utilise des structures non-mutables pour construire la solution.

De sorte à illustrer cette manière de faire, nous proposons de nous intéresser à un problème classique. L'objectif est de placer sur un échequier des reines de sorte à ce qu'aucune reine ne puisse en prendre une autre.

Définition 4.19 : Problèmes des n reines

Le **problème des n reines** est un problème qui, pour n un entier en entrée, cherche à construire une grille de taille $n \times n$ de sorte à ce que :
- Chaque case contient une reine ou non ;
- Il y a n reines au total ;
- Il n'y a a pas de reines sur une même ligne, une même colonne ou une même diagonale.

En implémentant les reines sous la forme d'une liste, on peut se passer de l'étape de rectification en faisant confiance aux structures non-mutables qui se trouvent sur la pile.

Exemple 4.19 : Implémentation à l'aide d'une structure non-mutable du problème des n reines

On commence par construire une fonction qui permet de vérifier si le prochain ajout dans une liste correspond bien à une liste de positions valides :

```
let est_compatible (i,j) (k, l) =
    if i=k then false
    else if j = l then false
    else if k - i = l - j then false
    else if i - k = l - j then false
    else true

let rec est_valide liste nouveau = match liste with
| [] -> true
| p::q -> est_compatible p nouveau && est_valide q nouveau
```

On utilise une fonction auxiliaire qui dispose d'un accumulateur qui se souvient des reines placées pour l'instant et du prochain indice à considérer. Par ailleurs, k contient la longueur de l'accumulateur.

```
let n_reines n =
    let rec aux i j acc k =
        if k = n then Some acc
        else if j>=n then aux (i+1) 0 acc k
        else if i >= n then None
        else
            begin
                if est_valide acc (i,j) then
                    let nouvel_acc = (i,j)::acc in
                    match aux i (j+1) nouvel_acc (k+1) with
                    | Some x -> Some x
                    | None -> aux i (j+1) acc k
                else
                    aux i (j+1) acc k
            end
    in aux 0 0 [] 0
```

Ici, on n'a pas eu besoin de modifier l'accumulateur en remontant dans la pile : le fait qu'il soit toujours local nous permet de ne pas avoir à le faire.

C'est l'un des intérêts des structures non-mutable dans le cas d'explorations récursives de plusieurs possibilités.

4.4 Rencontre au milieu

Pour certaines tailles de données, la recherche exhaustive est impossible, et on peut vouloir séparer le problème en plusieurs.

Exemple 4.20 : Temps d'exécution avec cette méthode

Il ne s'agit pas d'une méthode diviser-pour-régner, car en effet

Chapitre V

Logique

Dans un premier temps, nous nous intéressons à la *syntaxe* des formules logique, pour nous intéresser ensuite à la *sémantique* des formules logiques.

Enfin, nous nous intéressons à l'utilisation de ce formalisme pour exprimer des problèmes de décision sous la forme d'une recherche de satisfiabilité sur les formules logiques.

1 Syntaxe des formules logiques

Cette section est purement syntaxique : nous ne nous intéressons pas au sens des symboles mais sur leur organisation au sein d'une formule logique.

1.1 Formules sous la forme d'un arbre de syntaxe

Nous utilisons une définition par induction des formules de la manière suivante.

Définition 5.1 : Formule logique du calcul propositionnel

Soit $\mathcal{V}$ un ensemble. Les éléments de V sont appelés les **variables propositionnelles** ou **variables**.
Les **formules logique du calcul propositionnel** sont les formules définies inductivement de la manière suivante :

- Les formules réduite à un nom de variable v sont des formules ;
- Si φ est une formule, alors $\neg\varphi$ est une formule qu'on appelle négation de φ ;
- Si φ et ψ sont des formules, alors $\varphi \vee \psi$ est une formule qu'on appelle **disjonction** de φ et ψ ;
- Si φ et ψ sont des formules, alors $\varphi \wedge \psi$ est une formule qu'on appelle **conjonction** de φ et ψ ;
- Si φ et ψ sont des formules, alors $\varphi \to \psi$ est une formule qu'on appelle **implication** de φ vers ψ ;
- Si φ et ψ sont des formules, alors $\varphi \leftrightarrow \psi$ est une formule qu'on appelle **équivalence** de φ et ψ.

$\vee$, $\wedge$, $\neg$, $\to$ et $\leftrightarrow$ sont appelés les **connecteurs logiques**.

Exemple 5.1 : Exemples de formules logiques

On se donne a, b, c des variables. Voici des exemples de formules logiques :

$$a \vee c$$

$$\neg b$$

En revanche, les objets suivants ne sont pas des formules logiques :

$$a\neg$$

$$ab \vee c$$

Nous pouvons représenter ces formules logiques en écrivant les caractères qui lui correspondent. Cependant, il est parfois nécessaire de représenter les formules avec des parenthèses de sorte à pouvoir expliciter la structure qui se cache derrière.

Exemple 5.2 : Formules parenthésées

Soit a, b et c des variables, les formules suivantes sont différentes :

$$(a \vee b) \to c$$

$$a \vee (b \to c)$$

Notre construction est sans ambiguïté, grâce à cette distinction entre les différentes formules, il n'existe qu'une manière de construire chacune des formules.

Cependant, on peut avoir envie de se passer de certaines parenthèses en donnant un ordre de priorité et une associativité pour les formules.

Définition 5.2 : Conventions de notation

Exemple 5.3 : Conventions de notation pour l'associativité à gauche et la priorité

On peut même itérer sur certaines notation pour condenser certaines écritures.

Définition 5.3 : Convention de notation pour

$$\bigvee_{i=1}^{n} \varphi_i = \dots$$

$$\bigwedge_{i=1}^{n} \varphi_i = \dots$$

Nous voyons que nous avons deux types de connecteurs logiques que l'on peut distinguer selon le nombre de formules dont on a besoin pour construire une formule avec ce connecteur.

Définition 5.4 : Arité d'un connecteur logique

L' **arité** d'un connecteur logique est le nombre de sous-formules qu'il connecte.

Exemple 5.4 : Arité des connecteurs logique du calcul propositionnel

Les symboles $\vee$, $\wedge$, $\to$, $\leftrightarrow$ sont des connecteurs d'arité 2.
Le symbole $\neg$ est un connecteur d'arité 1.

On rajoute parfois deux connecteurs logiques spéciaux d'arité 0 à notre syntaxe.

Définition 5.5 : Symboles V et F

On peut enrichir notre syntaxe de deux connecteurs logiques V et F d'arité 0.

Dans la suite, nous pourrons les rajouter à notre syntaxe si nécessaire.

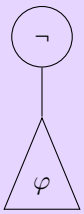
Étant donné que nous avons utilisé une définition inductive des formules, nous pouvons les apparenter à des arbres.

Définition 5.6 : Représentation sous forme d'arbre d'une formule logique

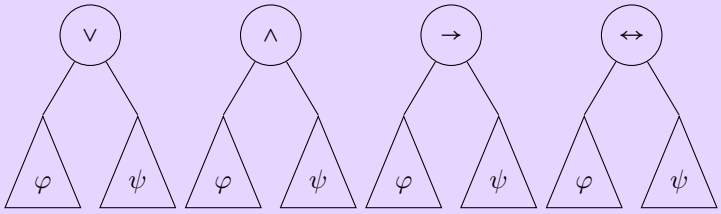
L'arbre qui représente une formule est l'arbre dont les étiquettes sont dans $\{\vee, \wedge, \neg, \rightarrow, \leftrightarrow\} \cup V$:

— Les formules réduite à une variable v sont représentés par des feuilles :

— La négation d'une formule φ est représenté par un arbre dont la racine est étiquetée par $\neg$, n'a qu'un enfant et dont l'enfant est l'arbre correspondant à l'arbre de φ :



— Les connecteurs d'arité 2 entre deux formules φ et ψ sont représentés par un arbre dont la racine est étiquettée par le connecteur, dont les deux enfants sont dans l'ordre l'arbre représentant φ, et l'arbre représentant ψ :



On remarque que cela ne correspond pas exactemenet à des arbres binaires car il n'y a pas toujours un enfant gauche ou droit dans le cas des nœuds internes : il n'importe pas de savoir si l'unique enfant d'un arbre étiquetté par la négation est un sous-arbre gauche ou droite.

Même si nous ne pouvons pas tout les arbres possibles avec cette représentation, la fonction qui nous donne la représentation sous la forme d'un arbre est injective.

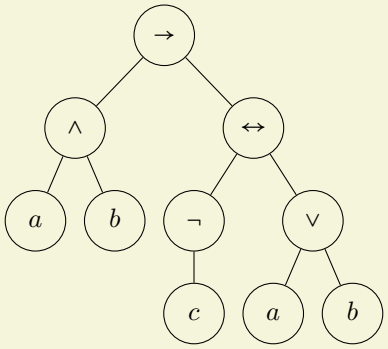
Proposition 5.1 : Unicité de la formule représenté par un arbre

Il existe au plus une formule qui soit représenté par un arbre.

En fait, nous pourrions avoir une bijection avec un sous ensemble des arbres étiquetés $\{\vee, \wedge, \neg, \rightarrow, \leftrightarrow\} \cup V$ en fixant les étiquettes possibles selon le nombre d'enfant du nœud.

Exemple 5.5 : Exemple de représentation sous forme d'arbre d'une formule logique

Par exemple, la formule $(a \wedge b) \rightarrow ((\neg c) \leftrightarrow (a \vee b))$ est représentée par l'arbre suivant :



Étant donné que nous travaillons sur des structures inductives, nous pouvons réaliser des preuves par induction de manière assez naturelle sur les formules logiques.

Proposition 5.2 : Arité d'un arbre d'une formule logique

Dans l'arbre représentant une formule logique, chaque nœud a au plus deux enfants.

De même, et puisque la construction des formules logique est sans ambiguïté, nous pouvons nous servir d'une définition inductive pour définir des grandeurs sur les formules.

Définition 5.7 : Taille d'une formule

On définit inductivement la **taille d'une formule** φ notée $|\varphi|$ de la manière suivante :

$$|\varphi| = \begin{cases} 1 & \text{si } \varphi = u \in V, \\ 1 + |\psi| & \text{si } \varphi = \neg\psi, \\ 1 + |\psi_1| + |\psi_2| & \text{si } \varphi = \psi_1 \vee \psi_2, \\ 1 + |\psi_1| + |\psi_2| & \text{si } \varphi = \psi_1 \wedge \psi_2, \\ 1 + |\psi_1| + |\psi_2| & \text{si } \varphi = \psi_1 \to \psi_2, \\ 1 + |\psi_1| + |\psi_2| & \text{si } \varphi = \psi_1 \leftrightarrow \psi_2. \end{cases}$$

Définition 5.8 : Hauteur d'une formule

On définit inductivement la **hauteur d'une formule** φ notée $h(\varphi)$ de la manière suivante :

$$h(\varphi) = \begin{cases} 1 & \text{si } \varphi = u \in V, \\ 1 + h(\psi) & \text{si } \varphi = \neg\psi, \\ 1 + \max(h(\psi_1), h(\psi_2)) & \text{si } \varphi = \psi_1 \vee \psi_2, \\ 1 + \max(h(\psi_1), h(\psi_2)) & \text{si } \varphi = \psi_1 \wedge \psi_2, \\ 1 + \max(h(\psi_1), h(\psi_2)) & \text{si } \varphi = \psi_1 \to \psi_2, \\ 1 + \max(h(\psi_1), h(\psi_2)) & \text{si } \varphi = \psi_1 \leftrightarrow \psi_2. \end{cases}$$

Définition 5.9 : Sous-formule

On définit inductivement l'ensemble des sous-formules de φ noté $F(\varphi)$ de la manière suivante :

$$F(\varphi) = \begin{cases} \{u\} & \text{si } \varphi = u \in V, \\ \{\varphi\} \cup F(\psi) & \text{si } \varphi = \neg\psi, \\ \{\varphi\} \cup F(\psi_1) \cup F(\psi_2) & \text{si } \varphi = \psi_1 \vee \psi_2, \\ \{\varphi\} \cup F(\psi_1) \cup F(\psi_2) & \text{si } \varphi = \psi_1 \wedge \psi_2, \\ \{\varphi\} \cup F(\psi_1) \cup F(\psi_2) & \text{si } \varphi = \psi_1 \to \psi_2, \\ \{\varphi\} \cup F(\psi_1) \cup F(\psi_2) & \text{si } \varphi = \psi_1 \leftrightarrow \psi_2, \end{cases}$$

Exercice 5.1. Quelle est la taille de la formule $(a \wedge b) \to ((\neg c) \leftrightarrow (a \vee b))$? Sa hauteur ? Ses sous-formules ?

Bien évidemment, ces définitions sont compatibles avec les définitions que nous avions faites sur les arbres.

Proposition 5.3 : Compatibilité des définitions

La taille d'une formule correspond à la taille de sa représentation sous forme d'arbre.
La hauteur d'une formule correspond à la hauteur de sa représentation sous forme d'arbre.
Les sous-formules d'une formule sont exactements les formules correspondant à un sous-arbre de la représentation sous forme d'arbre de la formule.

Dans la suite, nous pourrons mélanger sans ambiguïté les représentation sous forme d'arbre et les formules.

Exemple 5.6 : Implémentation d'une formule logique à l'aide d'un type récursif

On peut représenter une formule logique à l'aide du type suivant. On suppose que les variables prépositionnelles constituent l'ensemble des chaînes de caractère.

```
type formule =
    | Var of string
    | Neg of formule
    | Disjonction of formule * formule
    | Conjonction of formule * formule
    | Implication of formule * formule
    | Equivalence of formule * formule
```

Exercice 5.2. À quel objet OCaml correspond la formule $(a \wedge b) \to ((\neg c) \leftrightarrow (a \vee b))$?

On peut désormais utiliser ce type pour implémenter des algorithmes sur les formules logiques.

Exemple 5.7 : Calcul de la taille d'une formule

```
let rec taille f = match f with
| Var _ -> 1
| Neg x -> 1 + taille x
| Disjonction (x, y) -> 1 + taille x + taille y
| Conjonction (x, y) -> 1 + taille x + taille y
| Implication (x, y) -> 1 + taille x + taille y
| Equivalence (x, y) -> 1 + taille x + taille y
```

Exercice 5.3. Proposer une fonction qui calcul la hateur d'une formule.

1.2 Quantificateurs

On peut rajouter deux connecteurs logiques particulier qui nous permettent de quantifier sur les variables.

Définition 5.10 : Quantificateurs universel et existentiel

Le connecteur logique de **quantification universelle** permet à partir d'une formule φ et d'une variable prépositionnelle x, de construire la formule $\forall x, \varphi$.
Le connecteur logique de **quantification existentielle** permet à partir d'une formule φ et d'une variable prépositionnelle x, de construire la formule $\exists x, \varphi$.

Ces quantificateurs sont bien d'arité 1 : même s'ils construisent une nouvelle formule à partir d'une variable et d'une formule, c'est bien le fait d'être construit à partir d'une formule exactement qui nous importe.

Pouvoir quantifier sur les variables nous ouvre une nouvelle logique, celle du premier ordre qui permet de quantifier sur les variables (mais pas sur les propositions).

Définition 5.11 : Formule logique du calcul des prédicats du premier ordre

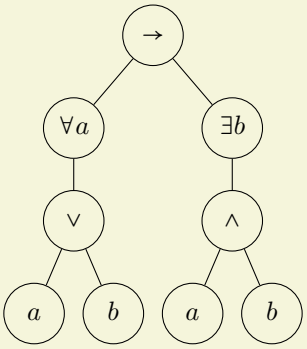
Soit $\mathcal{V}$ un ensemble. Les éléments de V sont appelés les **variables propositionnelles** ou **variables**.
Les **formules logique du calcul des prédicats du premier ordre** sont les formules définies inductivement de la manière suivante :

- Les formules réduite à un nom de variable v sont des formules ;
- Si φ est une formule, alors $\neg\varphi$ est une formule qu'on appelle négation de φ ;
- Si φ et ψ sont des formules, alors $\varphi \vee \psi$ est une formule qu'on appelle **disjonction** de φ et ψ ;
- Si φ et ψ sont des formules, alors $\varphi \wedge \psi$ est une formule qu'on appelle **conjonction** de φ et ψ ;
- Si φ et ψ sont des formules, alors $\varphi \to \psi$ est une formule qu'on appelle **implication** de φ vers ψ ;
- Si φ et ψ sont des formules, alors $\varphi \leftrightarrow \psi$ est une formule qu'on appelle **équivalence** de φ et ψ :
- Si φ est une formule, et v une variable, alors $\forall v, \varphi$ est une formule qu'on appelle **quantification universelle** ;
- Si φ est une formule, et v une variable, alors $\exists v, \varphi$ est une formule qu'on appelle **quantification existentielle**.

On peut toujours représenter les formules sous la forme d'un arbre.

Exemple 5.8 : Représentation sous forme d'arbre avec les quantificateurs

La formule $(\forall a, a \vee b) \to (\exists b, a \wedge b)$ peut se représenter par l'arbre suivant :



On peut étendre de manière assez naturelles les notions de taille, hauteur, et sous-formule à partir de là.

De plus on peut rajouter deux concepts que l'on peut définir inductivement : les concepts de variables libres et variables liées.

Une variable libre est une variable qui est présente dans la formule sans être fixée par un quantificateur, tandis qu'une variable liée est une variable qui est présente dans la formule et qui est fixée par un quantificateur.

Définition 5.12 : Variable libre

On défini récursivement l'ensemble des variables libres d'une formule φ, noté $L(\varphi)$ de la manière suivante :

$$L(\varphi) = \begin{cases} \{u\} & \text{si } \varphi = u \in V, \\ L(\psi) & \text{si } \varphi = \neg\psi, \\ L(\psi_1) \cup L(\psi_2) & \text{si } \varphi = \psi_1 \vee \psi_2 \text{ ou } \varphi = \psi_1 \wedge \psi_2 \\ & ou\varphi = \psi_1 \to \psi_2 \text{ ou } \varphi = \psi_1 \leftrightarrow \psi_2, \\ L(\psi) \setminus \{x\} & \text{si } \varphi = \forall x, \psi \text{ ou } \varphi = \exists x, \psi \end{cases}$$

Définition 5.13 : Variable liée

On défini récursivement l'ensemble des variables liées d'une formule φ, noté L de la manière suivante :

$$L(\varphi) = \begin{cases} \varnothing & \text{si } \varphi = u \in V, \\ L(\psi) & \text{si } \varphi = \neg\psi, \\ L(\psi_1) \cup L(\psi_2) & \text{si } \varphi = \psi_1 \vee \psi_2 \text{ ou } \varphi = \psi_1 \wedge \psi_2 \\ & ou\varphi = \psi_1 \to \psi_2 \text{ ou } \varphi = \psi_1 \leftrightarrow \psi_2, \\ L(\psi) \cup \{x\} & \text{si } \varphi = \forall x, \psi \text{ ou } \varphi = \exists x, \psi \end{cases}$$

Il s'avère que ces concepts ne sont pas exactement complémentaires.

Exemple 5.9

Dans la formule suivante, la variable x est liée et libre, la variable y est liée mais pas libre, la variable z est libre mais pas liée, et la variable w n'est ni l'un ni l'autre.

$$(\forall x, z) \wedge (\exists y, x)$$

Parfois, on veut justement se débrouiller pour que les variables présentes dans la formule soient découpées en deux ensembles disjoints des variables libres et liées.

L'un des soucis est qu'on peut avoir plusieurs variables du même nom, et il est important de parler de portée et de différencier les variables selon leurs occurences.

Définition 5.14 : Occurence d'une variable

Une **occurence** d'une variable est une sous-formule réduite à cette variable.
Une **occurence libre** d'une variable est une occurence d'une variable qui contribue à la liberté de la variable.
Une **occurence liée** d'une variable est une occurence d'une variable qui contribue au caractère lié de la variable.

La notion d'occurence permet de distinguer les sous-formules dans une formule : des arbres de mêmes structures, réduits à la même variable, se distinguent par leur position dans la formule.

Définition 5.15 : Portée dans une formule logique

La **portée** d'une variable liée par un quantificateur $\forall x, \varphi$ ou $\exists x, \varphi$ est l'ensemble des occurences libres de x dans φ.

Pour les occurences libres d'une variable, on peut réaliser une substitution.

Définition 5.16 : Substitution d'une variable

Une substitution d'une variable x par une formule ψ dans φ est le remplacement des occurences libres de x dans φ par la formule ψ.
On écrit parfois $\varphi[x \to \psi]$.

Exercice 5.4. Proposer une définition inductive de la substitution.

Il y a quelques subtilités sur les substitutions de variables qui ne sont pas au programme, et l'on peut se contenter d'une formalisation intuitive de la question de substitution.

2 Sémantique de vérité du calcul propositionnel

Nous pouvons désormais nous intéresser à la sémantique des formules, c'est-à-dire au sens qu'elles portent.

Dans cette partie, on esquive les parties techniques liées aux quantificateurs, et on se restreint donc au calcul propositionnel.

2.1 Valuation et évaluation

La notion de sémantique se base sur une transformation d'une formule logique vers le vrai ou le faux. Il y a plusieurs manière de les nommer, parfois 1 et 0, parfois T et F, parfois $\top$ et $\bot$, mais dans le cadre du cours, nous utiliserons le formalisme avec V et F.

Définition 5.17 : Valuation

Une **valuation** sur un ensemble $\mathcal{V}$ de variable est une fonction v de $\mathcal{V}$ vers $\{V, F\}$.

L'idée derrière la notion de valuation est de transformer les variables vers des valeurs. Naturellement, nous pouvons utiliser cette transformation des variables vers les valeurs pour évaluer la valeur de vérité d'une formule logique.

Nous devons donc définir les opérations sur les valeurs $\{V, F\}$ pour pouvoir les appliquer inductivement.

Définition 5.18 : Algèbre de Boole

On définit les fonctions suivantes sur $\{V, F\}$:

$$\neg x = \begin{cases} V & \text{si } x = F, \\ F & \text{sinon.} \end{cases}$$

$$x \vee y = \begin{cases} V & \text{si } x = V \text{ ou } x = V, \\ F & \text{sinon.} \end{cases}$$

$$x \wedge y = \begin{cases} V & \text{si } x = V \text{ et } y = V, \\ F & \text{sinon.} \end{cases}$$

Ici, nous avons noté de la même manière les fonctions et les connecteurs logiques : c'est un abus de notation qui est cohérent avec la manière dont on défini inductivement l'évaluation d'une formule logique.

Le plus souvent, pour économiser des parenthèses, on veut donner un ordre de priorité sur certaines de ces fonctions : la négation est prioritaire sur les autres fonctions en cas d'ambiguïté.

Exemple 5.10 : Priorité de la négation sur les autres fonctions

La formule $\neg a \vee b$ est interprétée $(\neg a) \vee b$.

Par ailleurs, puisque certaines fonctions sont associatives, on peut se passer de parenthèses.

Proposition 5.4 : Associatitivé des fonctions $\vee$ et $\wedge$

Pour tout $x, y, z \in \{V, F\}$, on a :

$$(x \vee y) \vee z = x \vee (y \vee z)$$
$$(x \wedge y) \wedge z = x \wedge (y \wedge z)$$

Exemple 5.11 : Utilisation de l'associativité pour retirer les parenthèses

On peut écrire $x \vee y \vee z$ pour $(x \vee y) \vee z$ comme $x \vee (y \vee z)$.
On peut écrire $x \wedge y \wedge z$ pour $(x \wedge y) \wedge z$ comme $x \wedge (y \wedge z)$.

Cela est donc différent de l'associativité à gauche au sens syntaxique des formules.

Par conséquent, on pourra parfois parler de conjonction de plusieurs valeurs sans rentrer dans les détails de l'ordre des formules.

Définition 5.19 : Conjonction et disjonction d'un ensemble de vérité

Soit I un ensemble d'indices. Soit une fonction a de $\{V, F\}^I$ (que l'on note indiciellement).

$$\bigvee_{i \in I} a_i = \begin{cases} V & \text{s'il existe } i \in I \text{ tel que } a_i = V, \\ F & \text{sinon.} \end{cases}$$
$$\bigwedge_{i \in I} a_i = \begin{cases} V & \text{s'il n'existe pas } i \in I \text{ tel que } a_i = F, \\ F & \text{sinon.} \end{cases}$$

On remarque qu'on n'a pas besoin d'avoir un ordre ici sur les valeurs dont on fait la conjonction contrairement au cas de la conjonction de formule.

Définition 5.20 : Évaluation

L' **évaluation** d'une formule φ sur une valuation v notée $[\![\varphi]\!]_v$ est définie inductivement :

$$[\![\varphi]\!]_v = \begin{cases} v(u) & \text{si } \varphi = u \in V, \\ \neg[\![\psi]\!]_v & \text{si } \varphi = \neg\psi, \\ [\![\psi_1]\!]_v \vee [\![\psi_2]\!]_v & \text{si } \varphi = \psi_1 \vee \psi_2, \\ [\![\psi_1]\!]_v \wedge [\![\psi_2]\!]_v & \text{si } \varphi = \psi_1 \wedge \psi_2, \\ \neg[\![\psi_1]\!]_v \vee [\![\psi_2]\!]_v & \text{si } \varphi = \psi_1 \to \psi_2, \\ (\neg[\![\psi_1]\!]_v \wedge \neg[\![\psi_2]\!]_v) \vee ([\![\psi_1]\!]_v \wedge [\![\psi_2]\!]_v) & \text{si } \varphi = \psi_1 \leftrightarrow \psi_2. \end{cases}$$

C'est très exactement ici que nous donnons une valeur sémantique au calcul prépositionnel.

Cela nous donne une implémentation naturelle des valuation et de l'évaluation sur une valuation.

Exemple 5.12 : Implémentation d'une valuation

Si on utilise l'ensemble des variables comme l'ensemble des chaînes de caractères, et en utilisant le type `bool` pour représenter $\{V, F\}$, une valuation est exactement une fonction de type `string -> bool`.

Exemple 5.13 : Implémentation d'une évaluation en OCaml

On peut implémenter l'évaluation d'une formule φ en OCaml sur une valuation v de la manière suivante :

```
let evaluer phi v = match phi with
| Var x -> v x
| Conjonction (psi1, psi2) -> evaluer psi1 v && evaluer psi2 v
| Disjonction (psi1, psi2) -> evaluer psi1 v || evaluer psi2 v
| Implication (psi1, psi2) -> (not evaluer psi1 v)|| evaluer psi2 v
| Equivalence (psi1, psi2) ->
    let val1 = evaluer psi1 v in
    let val2 = evaluer psi2 v in
    (val1 && val2)||(not val1 && not val2)
```

2.2 Modèle

Étant donné une formule logique, on peut s'intéresser aux valuations qui évalue cette formule au vrai.

Définition 5.21 : Modèle

Un **modèle** pour une formule logique φ est une valuation v telle que $[\![\varphi]\!]_v$ soit égal à V.

Le concept de modèle est légèrement plus lagre que le concept de valuation, mais dans notre cas, cela revient sensiblement au même.

Définition 5.22 : Satisfiabilité

Une formule est dite **satisfiable** s'il existe un modèle pour cette formule.

Exemple 5.14 : Exemple de formule satisfiable ou non satisfiable

Nous nous intéresserons un peu plus spécifiquement aux questions de satisfiabilité dans la dernière section sur les problèmes de recherche de satisfiabilité.

Il existe un autre concept lié aux formules satisfiables qui est la notion de tautologie, une formule qui est toujours vraie, quelque soit le modèle.

Définition 5.23 : Tautologie

Une formule est une **tautologie** lorsque toute valuation est un modèle pour cette formule.

Exemple 5.15 : Exemple de tautologie

Bien évidemment, le concept de tautologie semble plus contraignant que celui de satisfiabilité.

Proposition 5.5 : Tautologie et satisfiabilité

Si une formule est une tautologie alors elle est satisfiable.

Les variables qui ne sont pas présentes dans une formule ne nous intéressent pas pour l'évaluation sur une valuation de cette formule, et on peut donc s'intéresser seulement aux variables qui sont présentes dans la formule.

De plus, comme le nombre de formule est fini, on peut représenter le nombre fini de valeurs possibles pour ces variables.

On peut en effet exploiter la propriété suivante qui nous confirme qu'on peut prendre n'importe quelle

Proposition 5.6 : Égalité des évaluations pour des valuations égales sur les variables d'une formule

Soit φ une formule. Soit v et v' deux valuations telles que pour toute variable a présente dans φ, on ait $v(a) = v'(a)$.
Alors, on a :

$$[\![\varphi]\!]_v = [\![\varphi]\!]_{v'}$$

Dès lors, on peut définir la table de vérité en ne considérant qu'une seule des valuations en fixant les valeurs pour chacune des variables présente dans la formule.

Définition 5.24 : Table de vérité

La table de vérité d'une formule prépositionnelle φ de variables $a_1, a_2, \cdots a_n$ est une table qui fait figurer pour chaque 2^n possibilités pour les a_i leurs valeurs et la valeur de φ avec une valuation pour cette possibilité.

a_1	a_2	$\cdots$	a_n	φ
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
x_1	x_2	$\cdots$	x_n	$[\![\varphi]\!]_v$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Où v est une valuation qui associe x_i à a_i pour tout $1 \leqslant i \leqslant n$.

On peut rajouter quelques colonnes intermédiaires pour simplifier le calcul.

Exemple 5.16 : Table de vérité d'une formule

Exercice 5.5.

Assez naturellement, la table de vérité nous donne autant d'information que les modèles.

Proposition 5.7 : Caractérisation des formules satisfiables et des tautologie à partir de la table de vérité

Une formule est satisfiable si et seulement si il existe un V dans la dernière colonne de sa table de vérité.
Une formule est une tautologie si et seulement si il n'existe pas de F dans la dernière colonne de sa table de vérité.

2.3 Équivalence et conséquence logique de formules

Dès lors que l'on a donné un sens aux formules, on peut s'intéresser aux implications et équivalences des formules d'un point de vue des modèles.

Définition 5.25 : Équivalence de formules

Deux formules φ et ψ sont **équivalentes** lorsque pour toute valuation v, on a $[\![\varphi]\!]_v = [\![\psi]\!]_v$, et on note $\varphi \Leftrightarrow \psi$.

Ici, il n'est pas question de s'intéresser à la manière de construire des formules de même sens, on détermine l'équivalence par les modèles et donc par l'observation dans l'évaluation.

Vous verrez l'année prochaine plus de détails sur la manière de définir des axiomes pour déterminer des preuves formelles sans passer par les modèles.

De la même manière que pour la satisfiabilité et le caractère tautologique, on peut caractériser l'équivalence à l'aide de la table de vérité.

Proposition 5.8 : Caractérisation des formules équivalentes à partir de la table de vérité

Deux formules sont équivalentes si et seulement si elles ont la même table de vérité.

Ici, il y a deux choses dans la question de la table de vérité. Il faut avoir les mêmes variables, et avoir l'égalité de l'évaluation pour chacune des possibilités de ces valeurs.

En particulier, on peut utiliser

Proposition 5.9 : Caractérisation de la satisfiabilité pour des formules équivalentes

Soit φ et ψ deux formules équivalentes. Alors φ est satisfiable si et seulement si ψ est satisfiable.

Ainsi, si on cherche à déterminer le caractère satisfiable d'une formule, on peut raisonner sur des formules équivalente à cette formule.

Proposition 5.10 : Loi de De Morgan

Proposition 5.11 : Réecriture sans conjonction

De la même manière on peut écrire sans disjonctions.

De plus, dans un autre temps, on peut modifier une formule pour obtenir une formule équivalente où les négations sont placées seulement au niveau des variables.

Proposition 5.12 : Réécriture dont les négations sont toujours sur les variables

Proposition 5.13 : Tiers Exclu

Proposition 5.14 : Décomposition de l'implication

Exercice 5.6. Montrer l'équivalence des formules ... et ....

Définition 5.26 : Conséquence logique d'une formule

On dit qu'une formule φ a pour **conséquence logique** une autre formule ψ lorsque tout modèle de φ est un modèle de ψ, et on note $\varphi \Rightarrow \psi$ ou $\varphi \vDash \psi$.

Exemple 5.17

De manière plus générale, on peut se concentrer sur les conséquences logiques d'un

Définition 5.27 : Ensemble de formules ayant pour conséqunce logique une formule

Soit Γ un ensemble de formules, on dit que Γ a pour conséquence logique une formule ψ si une valuation qui est un modèle pour chacune des formules de Γ est un modèle pour φ.
On note alors $\Gamma \vDash \varphi$.

Exemple 5.18

Ici, on n'a fait aucune supposition sur Γ en tant qu'ensemble, ni finitude ni dénombrabilité. Mais il s'avère qu'on peut souvent se restreindre au cas finis.

Proposition 5.15 : Ensemble de formules fini ayant une conséquence logique une formule

Soit Γ un ensemble fini de formules. Soit φ une formule.
Γ a pour conséqunece logique φ si et seulement la conjonction des formules de Γ a pour conséquence logique φ.

Le fait d'avoir un ensemble de formules au lieu d'une conjonction dans la définition des formules nous permet de ne pas se restreindre justement au cas fini.

3 Minimisation de formule

3.1 Table de Karnaugh

Définition 5.28 : Code de Gray

Définition 5.29 : Table de Karnaugh

3.2 Algorithme de Quine

Algorithme 5.1 : Algorithme de Quine

4 Problèmes de Satisfiabilité

Une des applications du calcul propositionnel est la définition d'un certain type de problèmes qui détermine si une formule donnée en entrée sous une certaine forme est satisfiable.

4.1 Formes normales

Pour déterminer la taille d'une entrée

Définition 5.30 : Formes normales

Proposition 5.16 : Existence d'une forme normale

4.2 SAT et n-SAT

Définition 5.31 : Problème de satisiafibilité d'une formule sous forme disjonctive

Proposition 5.17 : La satisfiabilité d'une formule sous forme disjonctive

Ce n'est donc pas les formules sous forme disjonctive qui vont nous intéresser, mais sous forme conjonctive.

Définition 5.32 : SAT

La taille de l'entrée est la taille de la formule.

Exemple 5.19

Puisque nous savons comment se ramener à une forme normale disjonctive, et qu'il est simple de se traiter le cas de la forme normale disjonctive, il serait tentant de s'y ramener.

Cependant, une forme normale disjonctive construite à partir d'une forme normale conjonctive est potentiellement très grande.

Proposition 5.18 : Taille de la forme normale

Définition 5.33 : n-SAT

Vous verrez l'année prochaine qu'il est possible

4.3 Expression d'un problème sous forme d'un problème de satisfiabilité

Les différents problèmes de satisfiabilité nous permettent de décrire d'autres problèmes, et de nous ramener à un problème de satisfiabilité.

Définition 5.34 : Réduction

L'idée qui se cache derrière est bel et bien de passer d'un problème à un autre en étant cabable de transformer toute instance du problème de départ en une instance du problème auquel on veut se ramener.

Parfois, ce qui nous intéresse est de pouvoir réaliser cette transformation de manière peu couteuse dans un certain sens de complexité : il faut que la réduction soit rapide à faire.

Définition 5.35 : Réduction polynomiale

La notion de réduction n'est pas exactement au programme de MP2I, mais elle occupe une part importante du programme de seconde année, en particulier dans les questions de classe de complexité et de calculabilité.

Dans le cas du programme de première année, on se concentre principalement sur le fait de réduire un problème à une question de satisfiabilité.
Nous cherchons donc à partir d'un problème, pour tout instance de ce problème, pouvoir construire une instance de n-SAT ou de SAT, et si possible rapidement.

Dans un premier exemple classique, nous allons nous intéresser à un problème de coloration d'un graphe.

Définition 5.36 : Problème de coloration d'un graphe

Le **problème de coloration d'un graphe** La taille de l'entrée est $|A| + |S|$ pour ce qui est des questions de complexité.

Proposition 5.19 : Réduction du problème de coloration d'un graphe à SAT

Ici, on s'est réduit à 3-SAT, mais il s'avère que se ramener à n-SAT pour un n plus grand n'est pas un problème car il est relativement facile de se ramener à 3-SAT.

Proposition 5.20 : Réduction polynomiale de n-sat à 3-SAT

On ne peut malheureusement pas faire de réduction polynomiale de n-SAT à 2-SAT. Vous verrez l'année prochaine que 2-SAT est un problème pour lequel on dispose d'un algorithme polynomiale pour les résoudre, tandis que 3-SAT (et SAT ainsi que n-SAT pour $n \geqslant 3$) sont des problèmes NP-complets.

Même si 3-SAT est NP-complets, s'y ramener est pratique car on peut ensuite utiliser des algorithmes et des heuristiques spécifiques aux problèmes de satisfiabilité et utiliser des solveurs dont le travail est très exactement de résoudre les problèmes de satisfiabilité.

Chapitre VI

Bases de Données

Chapitre VII

Algorithmique des textes

1 Sérialisation

Annexe A

Solutions des exercices

1.1 : Le programme affiche 3 et renvoie le message d'erreur 4.