Cours d'informatique

PCSI 2

Lycée Bellevue

Année 2024-2025

Ce document a pour but de donner un aperçu des notions essentielles vues en cours.
Il ne se substitue pas aux notes prises en cours, et ne remplace pas une attention soutenue en classe.

Table des matières

Chapitre 1

Programmation

Ce chapitre est dédié à la présentation des principaux éléments du langage Python que nous utiliserons tout au long de l'année. C'est principalement par la pratique de la programmation pendant les TP qu'il faudra assimiler leur manipulation !

1 Vocabulaire

1.1 Programmation et langages

Définition 1.1.1. Un *algorithme* est une suite finie et non ambiguë d'instructions permettant de résoudre un problème donné.

Exemple 1.1.2.

- Recette de cuisine
- Résolution d'une équation du second degré
- Recherche du plus petit élément d'une liste d'entiers...

On reviendra plus précisément sur la notion d'algorithme dans le chapitre 3.

Définition 1.1.3. Un *programme* est la mise en œuvre d'un algorithme dans un certain *langage de programmation*, c'est-à-dire une syntaxe compréhensible et utilisable par un ordinateur.
Un programme est écrit par l'utilisateur dans un logiciel appelé *éditeur* et exécuté dans un *interpréteur*, également appelé *shell*.

Remarque 1.1.4. L'interpréteur va lire et exécuter le programme ligne par ligne. Il se distingue en cela du *compilateur*, qui commence par convertir l'ensemble du programme en langage machine, puis l'exécuter. Cette seconde approche permet de gagner en rapidité, mais rend plus difficile la localisation d'une erreur de programmation.

Il existe de nombreux langages de programmation, plus ou moins adaptés à un contexte donné. Nous utiliserons cette année le langage *Python*, créé par Guido van Rossum en 1991 au Centrum voor Wiskunde en Informatica d'Amsterdam. C'est un langage *de haut niveau d'abstraction*, c'est-à-dire qu'il permet à l'utilisateur de se préoccuper de façon minimale de la gestion matérielle de l'ordinateur (dans quelle case mémoire stocker une variable nouvellement créée, etc). Sa syntaxe est de plus particulièrement épurée. Très répandu, de nombreuses librairies (un ensemble de fonctions utiles dans un domaine spécifique, disponibles en ligne) ont été développées pour ce langage. Enfin, il est à noter qu'il est couramment utilisé en *prototypage*, c'est-à-dire pour mettre au point un algorithme, qui sera ensuite réécrit dans un langage de plus bas niveau, plus efficace (en C par exemple).

1.2 Données

Définition 1.2.1. Les *données* sont les éléments reçus par un programme, en provenance de l'utilisateur ou de la mémoire de l'ordinateur. Le programme effectue des calculs puis renvoie de nouvelles données vers l'utilisateur ou

vers la mémoire.
Une donnée a un *type*, une *adresse mémoire*, une *valeur* et est affectée à une *variable*, c'est-à-dire à une suite de caractères désignant l'endroit de la mémoire où est stockée la donnée.

Exemple 1.2.2.

- La ligne `x=3` crée une variable appelée `x`, de type entier (*integer*) et de valeur 3.
  On dit que la variable `x` *prend la valeur* ou *reçoit* 3.
- La ligne `Mot = "bonjour"` crée une variable appelée `Mot`, de type chaîne de caractères (*string*) et de valeur `"bonjour"`.

Remarque 1.2.3. Dans de nombreux langages (par exemple en Pascal), il faut *déclarer*, c'est-à-dire créer, une variable, en précisant parfois son type, avant de lui affecter une valeur. Cela donne par exemple : `int x; x:=3;`
En Python, ces deux opérations sont simultanées : `x=3`

Définition 1.2.4. Une *expression* est une suite de caractères, composée de valeurs et d'*opérateurs*, définissant une valeur.

Exemple 1.2.5. Dans l'instruction `x=(4+6)**2`, l'expression `(4+6)**2` est composée des valeurs 4, 6 et 2 et des opérateurs + et `**` (puissance). Elle définit la valeur 100.

2 Types simples de variables

Définition 2.0.1. Un *type* est une classe d'expressions, munie d'opérateurs.

Remarque 2.0.2. En Python, pour connaître le type d'une variable `x`, on utilise l'instruction `type(x)`.

Les types *simples* (c'est-à-dire non composés) à connaître en Python sont :

- les entiers (nombres entiers, *integer*), dont les opérateurs sont :

  `+, -, *, **` (puissance),

  `//` (quotient de division euclidienne), `%` (reste dans la division euclidienne).
- les flottants (nombres réels, *float*), dont les opérateurs sont :

  `+, -, *, /, **` (puissance).
- les booléens (*boolean*) `True` et `False`, dont les opérateurs sont :

  `==, !=, <, >, <=, >=, not, and, or.`
- les chaînes de caractères (*string*) délimitées par `' '` ou `" "`, dont les opérateurs sont :

  + (concaténation), * (concaténation multiple).

3 Instructions

En plus de l'affectation de variables décrite plus haut, les instructions à connaître en Python sont :

- l'entrée `input` : elle met le programme en pause en attendant une donnée de l'utilisateur, qui sera stockée par défaut en type `string`,
  la sortie `print` : elle affiche des données à l'écran, dans le shell.

- les instructions conditionnelles :

  `if` booléen :
    instructions
  `elif` booléen :
    instructions
  `else` :
    instructions

  Attention : Pour cette instruction comme pour les suivantes, l'**indentation** est primordiale ; elle fait partie intégrante du langage.
- l'instruction itérative inconditionnelle :

  `for` variable `in` itérable :
    instructions

  Attention : L'itérable `range(n)` fait parcourir l'ensemble $[\![0, n-1]\!]$ à la variable.
- l'instruction itérative conditionnelle :

  `while` booléen :
    instructions

Attention : Il faut veiller à la *terminaison* de la boucle `while`, afin de ne pas tomber dans une boucle infinie.

4 Types composés

Les types *composés* à connaître en Python sont :

- les chaînes de caractères (*string*) délimités par `' '` ou `" "` et les n-uplets (*tuple*) délimités par `( )`, dont les opérateurs sont :

  $$+, \star,$$
- les listes (*list*) délimités par `[ ]`, dont les opérateurs sont :

  $$+, \star,$$

  et les fonctions associées sont `append` (ajout d'un élément), `pop` (suppression d'un élément d'indice indiqué), `remove` (suppression d'un élément de valeur indiquée), `insert` (insertion d'un élément).

Remarque 4.0.1. Les variables de ces types peuvent être utilisés comme itérables dans les boucles `for`.
Les listes sont structurellement plus simples à modifier et à manipuler, et seront utilisées dans la plupart des cas.

5 Fonctions

Définition 5.0.1. Une *fonction* est une suite d'instructions dépendant d'un *paramètre*, ou plusieurs, ou aucun, formant un *sous-programme* à part entière.

Remarque 5.0.2. L'utilisation de fonctions permet de découper un programme, et donc la résolution d'un problème, en plusieurs sous-parties répondant chacune à une étape dans cette résolution. C'est donc une méthode essentielle assurant une simplification et une meilleure lisibilité du programme.

En Python, la syntaxe est la suivante :

```
def fonction(paramètre) :
    instructions
    return sortie
```

Remarque 5.0.3. La ligne `return` sortie donne la valeur sortie à l'expression fonction(paramètre). Elle n'est pas nécessaire si la fonction n'est pas destinée à fournir une donnée.
En particulier, la commande `return` **est à ne pas confondre** avec la commande `print`.
Les variables créées et utilisées au sein d'une fonction sont *locales*, c'est-à-dire qu'elles ne sont pas définies (et donc pas utilisables) dans le reste du programme (on dit que la *portée* de la variable est la fonction). Il est possible de rendre une variable globale en précédant sa déclaration dans une fonction de la commande `global`.

Terminons ce chapitre en mentionnant quelques règles de bonne programmation :

- Utiliser des noms de variables et de fonctions parlants, voire explicites,
- **Commenter** les instructions en utilisant la commande `#`,
- Préférer écrire plusieurs fonctions courtes à un unique programme long.

Chapitre 2

Algorithmique

On revient dans ce chapitre sur la notion d'*algorithme*, rencontrée au chapitre 1.

1 Définitions

On rappelle la définition suivante :

Définition 1.0.1. Un *algorithme* est une suite finie et non ambiguë d'instructions élémentaires, fournissant la réponse à un problème donné.

Exemple 1.0.2.

- Résolution d'une équation du second degré
- Division euclidienne
- Algorithme du pivot de Gauss-Jordan...

Remarque 1.0.3. À la différence d'un *programme*, la notion d'algorithme est indépendante du langage de programmation utilisé : on ne s'intéresse qu'au raisonnement sous-jacent.

Pour représenter un algorithme, on a recours à un organigramme ou au *pseudo-code* :

Entrée : ...
$n \leftarrow 0$
Pour ... dans ... **faire**
  instructions
fin du Pour
Tant que ... **faire**
  instructions
fin du Tant que
Si ... **alors**
  instructions
fin du Si
Résultat : ...

Exemple 1.0.4. L'algorithme de la division euclidienne de a par b a pour pseudo-code :

Entrée : a, b
$q \leftarrow 0$
$r \leftarrow a$
Tant que $r \geq b$ **faire**
  $q \leftarrow q + 1$
  $r \leftarrow r - b$
Résultat : q, r

2 Validité d'un algorithme

Définition 2.0.1. Vérifier la *validité* d'un algorithme consiste à en vérifier :

- la *terminaison* : on montre que l'algorithme s'arrête au bout d'un nombre fini d'opérations,
- la *correction* : on montre que l'algorithme renvoie le résultat voulu.

Exemple 2.0.2. L'algorithme de la division euclidienne :

- termine car r diminue strictement à chaque itération de la boucle Tant que. Comme c'est un entier naturel, il ne peut y avoir qu'un nombre fini d'itérations d'après le principe de descente infinie. (On dit que r est un *variant de boucle*.)
- est correct car, au moment de sortie de la boucle, on a bien $r < b$ et $r + b \geq b$, soit $r \geq 0$; donc $r \in [\![0, b-1]\!]$, et on a bien $a - qb = r$.

3 Complexité

La performance d'un algorithme peut être évaluée par son *coût en temps* et par son *coût en mémoire*, c'est-à-dire, respectivement, au temps de calcul et à la quantité de mémoire nécessaires à son exécution.
On va principalement s'intéresser au coût en temps, en se plaçant dans un modèle où une opération élémentaire (addition, multiplication, test, affectation de variable...) correspond à une unité de temps (ou *unité de temps de calcul*).

Définition 3.0.1. La *complexité* en temps d'un algorithme correspond au nombre d'opérations élémentaires nécessaire à son exécution.

Remarque 3.0.2.

- La complexité dépend en général des paramètres des données d'entrée. Par exemple, la complexité d'un algorithme de tri de liste dépend de la longueur de la liste en entrée.
- La complexité est calculée par rapport au *pire des cas* de l'algorithme, c'est-à-dire au cas qui demande le plus de temps de calcul (par exemple, le pire des cas d'un test de primalité est celui où le nombre est premier).

Exemple 3.0.3.

- L'algorithme de détermination du plus grand élément d'une liste de n nombres utilise, dans le pire des cas, n affectations de variable et $n - 1$ tests, donc demande $2n - 1$ unités de temps.
- L'algorithme de détermination des éléments communs à deux listes de n éléments utilise, dans le pire des cas, n^2 tests et n affectations, donc demande $n^2 + n$ unités de temps.

Remarque 3.0.4. Ces calculs sont intrinsèquement approximatifs : les opérations considérées n'ont pas toutes la même durée, et celle-ci dépend fortement de l'implémentation de l'algorithme (langage utilisé, structure de l'ordinateur)... On préfère donc se limiter à l'*ordre* de temps mis en évidence : le premier algorithme a une complexité

de l'ordre de n, et le second une complexité *de l'ordre de* n^2.
Cette considération est d'autant plus pertinente que le nombre d'opérations effectuées par un ordinateur en une seconde est très élevé ; un algorithme est donc en général limité pour des données de grande taille, et il est alors important de savoir si, par exemple, le traitement d'une liste 10 fois plus grande demandera 10 ou 100 fois plus de temps de calcul.

Définition 3.0.5. On dit qu'une fonction $c(n)$ est *de l'ordre de* $f(n)$ ou *en* $O(f(n))$ (« en grand O de $f(n)$ »), et on note $c(n) = O\left(f(n)\right)$, s'il existe des constantes réelles A et B telles que :

$$\forall n \in \mathbb{N},\ Af(n) \leq c(n) \leq Bf(n).$$

Remarque 3.0.6. Selon l'ordre trouvé, on dira que la complexité d'un algorithme est (par ordre croissant de complexité) *logarithmique, linéaire, quadratique, polynomiale, exponentielle...*
Un algorithme est considéré *raisonnable* si sa complexité est au plus polynomiale. La recherche de tels algorithmes est un enjeu crucial dans certains domaines, et la question de leur existence fait l'objet du *problème du prix du millénaire* $P = NP$ [1].

Exemple 3.0.7.

- L'algorithme de détermination de la plus grande puissance de 2 inférieure à un entier A est en $O(\ln(A))$.
- L'algorithme itératif de calcul du $n^{\text{ème}}$ terme de la suite de Fibonacci est en $O(n)$. L'algorithme *récursif* est en $O(\varphi^n)$ où $\varphi = \dfrac{1+\sqrt{5}}{2}$, c'est-à-dire qu'il est *beaucoup plus lent* que le précédent pour de grandes valeurs de n.

Enfin, mentionnons que la complexité en mémoire peut également représenter un obstacle important, malgré les développements des supports. C'est par exemple le cas pour la recherche de nombres premiers inférieurs à N avec le crible d'Ératosthène, celui-ci nécessitant le stockage d'un tableau de taille N : N ne peut alors pas dépasser le nombre de bits en mémoire.

1. présenté par exemple dans cette vidéo de David Louapre : `https://www.youtube.com/watch?v=AgtOCNCejQ8`

Chapitre 3

Représentation des entiers et des flottants

1 Entiers

1.1 Bases de numération

La représentation binaire des entiers est rendue possible par le résultat suivant :

Théorème 1.1.1 (Représentation des entiers en base b)**.** *Soit $b \geq 2$ entier. Pour tout $n \in \mathbb{N}$, il existe $k \in \mathbb{N}$ et $a_0, a_1, \ldots, a_k \in [\![0, b-1]\!]$, avec $a_k \neq 0$, uniques tels que :*

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0.$$

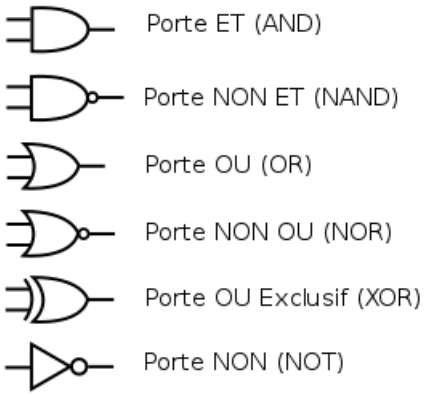
On note $n = (a_k a_{k-1} \cdots a_0)_b$, et on dit que $a_k a_{k-1} \cdots a_0$ est la représentation de n dans la base b*.*

En informatique, les bases les plus utilisées sont :

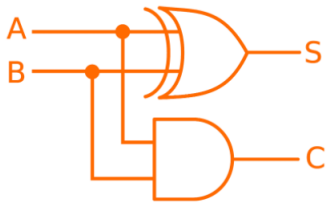
- la base *binaire* ($b = 2$), dont les chiffres, 0 et 1, sont appelés *bits (binary digits)*. Cette base permet de coder des informations sur des supports à deux états (présence/absence de courant (circuit électronique) ou de lumière (fibre optique), polarisation magnétique (disque dur)...)
- la base *hexadécimale* ($b = 16$), compatible avec la base binaire (à un chiffre hexadécimal $(0, 1, \ldots, 9, A, B, \ldots, F$ correspondent quatre bits). Elle permet de représenter un code binaire de façon plus compacte.

1.2 Portes logiques

Au niveau du microprocesseur, les opérations sont effectuées par des composants appelés *portes logiques*, qui traitent les signaux selon les opérateurs logiques usuels :



Par exemple, l'addition de deux bits x et y est effectuée par le circuit suivant :



A	B	S A + B	C Retenu
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

1.3 Entiers relatifs

Les entiers sont en général codés sur 32 (ou 64) bits. Plutôt que de faire correspondre à chaque entier sa représentation binaire (ce qui permettrait de coder les entiers de 0 à $2^{32}-1$, mais ne permettrait pas de coder les entiers négatifs), on préfère coder les entiers de -2^{31} à $2^{31}-1$ comme suit :

- les entiers $k \in [\![0, 2^{31}-1]\!]$ sont codés par leur représentation binaire,
- les entiers $k \in [\![-2^{31}, -1]\!]$ sont codés par la représentation binaire de $2^{32} - |k|$.

En pratique, cela revient, pour coder l'entier négatif k à :

- coder l'entier positif $|k|$,
- prendre le complémentaire de ce code (en changeant les 0 en 1 et inversement)
- rajouter 1.

Ce codage, appelé *complément à deux*, a l'avantage d'être compatible avec l'addition. Le premier bit donne le signe de l'entier (0 pour $+$, 1 pour $-$).

2 Flottants

La représentation binaire des réels est rendue possible par le résultat suivant :

Théorème 2.0.1 (Représentation des réels en base b)**.** *Soit $b \geq 2$ entier. Pour tout $x \in \mathbb{R}$, il existe $n \in \mathbb{Z}$ et $a_0, a_1, \ldots \in [\![0, b-1]\!]$, avec $a_0 \neq 0$, uniques tels que :*

$$x = \pm(a_0 + a_1 b^{-1} + a_2 b^{-2} + \cdots) \times b^n.$$

Le nombre $a_0, a_1 a_2 \cdots$ est appelé la mantisse *de x et n est appelé l'*exposant *de x. On dit que $a_0, a_1 a_2 \cdots \times b^n$ est l'*écriture scientifique de x dans la base b.

Les réels sont codés informatiquement par des *nombres en virgule flottante*, plus communément appelés *flottants*. Ce codage est réglementé par la norme IEEE 754 :

- 1 bit est consacré au signe : 0 pour $+$, 1 pour $-$,
- 11 bits sont consacrés à l'exposant n, codé en complément à 2,
- 52 bits sont consacrés à la mantisse $a_1 \cdots a_{52}$ (comme a_0 est nécessairement égal à 1, on ne le stocke pas).

Chapitre 4

Tris

On présente dans ce chapitre différents algorithmes de tris, avec leurs avantages et inconvénients respectifs.

Pour chaque algorithme présenté ci-dessous, la situation est la suivante : on veut trier une liste L de n nombres (ou de chaînes de caractères, ou de tout type pour lequel il existe une relation d'ordre).

1 Tris en place

Les algorithmes que l'on va écrire dans cette partie agissent directement sur la liste donnée en entrée pour la trier, sans stockage de liste auxiliaire. On parle de *tri en place*.

1.1 Tri par sélection

Le *tri par sélection* consiste à chercher le plus petit élément de la liste L et à l'échanger avec l'élément en position initiale, puis à recommencer avec les éléments restants. Voici son pseudo-code :

Entrée : L (liste de longueur n)
Pour i de 0 à $n-2$:
    $j =$ indice du plus petit des $n-i$ derniers éléments de la liste
    $L[i] \leftrightarrow L[j]$
Sortie : rien, mais la liste est ordonnée.

Sa complexité est en $O(n^2)$.

1.2 Tri par insertion

Le *tri par insertion* consiste à trier les éléments de la liste L à mesure de la lecture de celle-ci : au moment de la lecture du $k^{\text{ème}}$ élément $L[k]$ de la liste, les $k-1$ premiers éléments de la liste sont déjà triés, et on insère alors $L[k]$ à sa place parmi les précédents :

Entrée : L (liste de longueur n)
Pour k de 1 à $n-1$:
    $i \leftarrow k-1$
    Tant que $i \geq 0$ et $L[i] > L[k]$:
        $L[i] \leftrightarrow L[k]$
        $i \leftarrow i-1$
    Insérer $L[k]$ entre $L[i]$ et $L[i+1]$ Sortie : rien, mais la liste est ordonnée.

Sa complexité est en $O(n)$ dans le meilleur des cas (lorsque la liste est déjà triée), en $O(n^2)$ dans le pire des cas (lorsque la liste est triée en ordre inverse).

1.3 Tri à bulles

Le *tri à bulles*, ou *tri par propagation*, est moins efficace ; il est cependant particulièrement facile à coder. Il consiste à comparer les éléments consécutifs de la liste, et à les échanger s'ils ne sont pas dans l'ordre :
Entrée : L (liste de longueur n)
Pour i de $n-1$ à 1 :
    Pour j de 0 à $i-1$:
        Si $L[j+1] < L[j]$:
            $L[j] \leftrightarrow L[j+1]$
Sortie : rien, mais la liste est ordonnée.

2 Tris récursifs

Les algorithmes de tris suivants sont récursifs. Ils ne sont pas *en place*, c'est-à-dire qu'ils utilisent une quantité importante de variables auxiliaires, et donc davantage de mémoire. En contrepartie, ils peuvent être plus rapides que les tris précédents.

2.1 Tri fusion

Le *tri par partition-fusion*, ou *tri fusion* est un algorithme dichotomique : si la liste L contient un seul élément, elle est déjà triée ; sinon, on la sépare en deux sous-listes de même taille, que l'on trie par partition-fusion, puis on fusionne ces deux sous-listes triées afin d'obtenir une liste triée.

Cet algorithme utilise donc une fonction auxiliaire de fusion, elle-même récursive :

Fonction de fusion :
Entrée : deux listes triées A et B
Si A est vide :
    Renvoyer B
sinon, si B est vide :
    Renvoyer A
sinon, si $A[0] < B[0]$:
    Renvoyer $[A[0]]$+fusion$(A[1:],\ B)$
sinon :
    Renvoyer $[B[0]]$+fusion$(A,\ B[1:])$
Sortie : une liste triée, réunion de A et B

Tri fusion :
Entrée : L (liste de longueur n)
Si $n \leq 1$:
    Renvoyer L
sinon :
    Renvoyer fusion(tri fusion$(L[:n//2])$, tri fusion$(L[n//2:])$)
Sortie : la liste L triée

Sa complexité est en $O(n\log(n))$.

2.2 Tri rapide

Le *tri rapide* (ainsi baptisé par son inventeur, Charles Antony Richard Hoare), ou *tri pivot*, consiste à choisir un élément de la liste (le *pivot*) et à séparer les autres éléments de la liste en deux sous-listes selon qu'ils sont plus grands ou plus petits que le pivot. Ces deux sous-listes sont alors triées par tri rapide, puis réunies avec le pivot.

Entrée : L (liste de longueur n)
Si $n \leq 1$:
    Renvoyer L
sinon :
    pivot=$L[0]$
    $G \leftarrow$ éléments de $L[1:] \leq$ pivot
    $D \leftarrow$ éléments de $L[1:] >$ pivot
    Renvoyer tri rapide(G) + [pivot] + tri rapide(D)
Sortie : la liste L triée

La complexité de cet algorithme a la particularité d'être en $O(n\log(n))$ lorsque la liste est bien mélangée, mais en $O(n^2)$ lorsque la liste est déjà triée. En pratique, il est donc avantageux... de remélanger la liste avant de la trier !

3 Autres algorithmes de tri

Le *tri par comptage*, ou *tri casier* est seulement utilisable sur des listes L d'entiers, et plus particulièrement adapté lorsque $n > N$ (avec les notations du I.). Il consiste à créer une *table de comptage* T de longueur $N+1$ telle que pour tout $k \in [\![0, N]\!]$, $T[k]$ soit égal au nombre d'occurrences de k dans L. Il suffit pour cela de parcourir L, et il est ensuite immédiat de construire la liste triée de L à partir de T.

Entrée : L (liste de longueur n d'entiers de $[\![0, N]\!]$)
$T \leftarrow$ liste nulle de taille $(N+1)$
Pour i de 0 à $n-1$:
    Incrémenter $T[L[i]]$
Reconstituer la liste L triée à partir de T
Sortie : Liste ordonnée contenant $T[k]$ fois k pour k de 0 à N.

Le *tri par paquets*, quant à lui, s'applique à des listes L de nombres réels compris dans un segment $[a, b]$ donné. On va supposer qu'il s'agit de $[0, 1]$. L'idée est de créer p paquets $L_0, \ldots, L_{p-1}$, puis de répartir les éléments de L entre les paquets, le paquet L_k recevant les nombres appartenant à l'intervalle $\left[\frac{k}{p}, \frac{k+1}{p}\right[$. On trie alors chaque paquet (avec l'un des cinq algorithmes vus ci-dessus), puis on réunit les paquets.

Entrée : L (liste de longueur n de réels de $[0, 1]$)
$L_0, \ldots, L_{p-1} \leftarrow$ listes vides
Répartir les éléments de L entre les paquets
Trier les paquets
Concaténer les paquets
Sortie : la liste L triée

Chapitre 5

Graphes

On présente dans ce chapitre la notion de *graphe*, présente dans de nombreux contextes (réseaux physiques et virtuels, arbres, diagrammes), et la question du *parcours* d'un tel objet.

1 Définition

1.1 Graphes orientés

Définition 1.1.1. Un *graphe orienté* est un couple $G = (S, A)$ où S est l'ensemble des *sommets* du graphe et $A \subset S \times S$ est l'ensemble des *arcs*.

Deux sommets i et j du graphe sont dits *adjacents* ou *voisins* si (i, j) ou (j, i) appartient à A.

Un arc (i, i) est appelé une *boucle*.

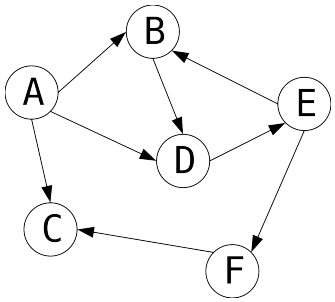
Définition 1.1.2. Un *chemin* du sommet a au sommet b est un sous-ensemble C de A de la forme $C = \{(s_1, s_2), \ldots, (s_n, s_{n+1})\}$ où $s_1 = a$ et $s_{n+1} = b$.

La *longueur* du chemin est le cardinal de C.

Remarque 1.1.3. Il est possible de *pondérer* un graphe en attribuant un *poids* à chaque arc (i, j), représentant le *coût* (distance, temps, énergie...) du passage de i à j.

Un graphe peut être représenté :

— par un *diagramme* :



— par une *matrice d'adjacence* :

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Un exemple de graphe orienté est donné par un réseau de voies automobiles, où certaines voies sont à sens unique. Pour un réseau de voies piétonnes, un graphe *non-orienté* est plus adapté.

1.2 Graphes non-orientés

Définition 1.2.1. Un *graphe non-orienté* est un couple $G = (S, A)$ où S est l'ensemble des *sommets* du graphe et A, ensemble d'éléments de la forme $\{i, j\}$ où $i, j \in S$, est l'ensemble des *arêtes*.

Les notions de voisins, de chemin, de diagramme, de matrice d'adjacence, s'étendent à ce type de graphe.

Proposition 1.2.2. *La matrice d'adjacence d'un graphe non-orienté est symétrique.*

2 Parcours de graphes

À la différence d'une liste, dont le parcours est naturellement linéaire, celui d'un graphe, objet à deux dimensions, nécessite de faire des choix. Deux types de parcours prédominent : les parcours *en profondeur* et *en largeur*.

2.1 Parcours en profondeur

Le parcours en profondeur d'un graphe consiste, à partir d'un sommet initial s, à :

— définir deux listes vides V et M,

— *visiter* le sommet s, c'est-à-dire l'ajouter à V,

— *marquer* ses voisins non-visités, c'est-à-dire les ajouter en *tête* de M,

— Réitérer l'opération sur les sommets marqués jusqu'à ce que M soit vide.

La liste M est ici une *pile*. Le dernier sommet marqué est ici le prochain sommet visité. On parle aussi de protocole *lifo* (last in, first out).

Exemple 2.1.1. Parcours de labyrinthe

2.2 Parcours en largeur

Dans le parcours en largeur, , le marquage consiste à ajouter les voisins non-visités en *queue* de M.

La liste F est ici une *file*. Le dernier sommet marqué est ici le prochain sommet visité. On parle aussi de protocole *fifo* (first in, first out).

Exemple 2.2.1. Distances (en arêtes) à un sommet donné

2.3 Algorithme de Dijkstra

Découvert par Edsger Dijkstra en 1959, l'*algorithme de Dijkstra* détermine les plus courts chemins d'un sommet s fixé à chaque autre sommet du graphe. Il s'agit d'un troisième type de parcours :

— On visite le sommet s. On crée une liste D des distances à s. Initialement, tous les sommets sont supposés à une distance infinie, sauf s qui est à distance 0.

— on visite le sommet t le plus proche des sommets déjà visités. On met alors à jour la distance de s à t, puis toutes les distances à s en rajoutant les chemins passant par t.

De nombreux autres algorithmes répondent à ce problème du plus court chemin. Par exemple, l'*algorithme de Floyd-Warshall*, découvert en 1962, a une complexité légèrement supérieure à celui de Dijkstra, mais est nettement plus simple à implémenter.