1 Analyse d'algorithmes

1.1 Terminaison

Pour montrer la terminaison d'une fonction, c'est la même idée sous-jacente selon qu'elle soit récursive ou écrite avec des boucles :

- pour une fonction récursive, on peut montrer que les arguments des appels récursifs diminuent strictement au sens d'un ordre bien fondé, par exemple sur un entier positif strictement plus petit, sur le fils gauche ou droit d'un arbre binaire, sur la queue d'une liste chaînée, sur un couple plus petit selon l'ordre lexicographique...
- pour montrer qu'une boucle termine, on peut exhiber un **variant de boucle**, c'est-à-dire une quantité qui diminue strictement (à nouveau au sens d'un ordre bien fondé, même si on peut généralement travailler avec des entiers) ;
- une boucle **Pour** dont la plage d'indices est définie explicitement termine toujours si le corps de la boucle termine, car on peut utiliser `valeur max - indice courant` comme variant de boucle.

1.2 Correction

Pour montrer qu'une fonction est correcte :

- pour une fonction récursive, on peut prouver la correction par induction structurelle ou récurrence ;
- pour une fonction avec des boucles, on peut exhiber un **invariant de boucle** (à ne pas confondre avec variant) : une propriété qui est vraie avant la boucle et qui se conserve à chaque passage dans la boucle sera vraie après la boucle.

1.3 Complexité

- On distingue la complexité **temporelle** (combien de temps va mettre le calcul) et la complexité **spatiale** (de quel espace mémoire a besoin le calcul). On ne compte généralement pas les arguments d'une fonction dans le calcul de la complexité spatiale.
- Notations de Landau :
  * $f(n) = \mathcal{O}(g(n))$ signifie que f est asymptotiquement majorée par une constante fois g ;
  * $f(n) = \Omega(g(n))$ signifie que f est asymptotiquement minorée par une constante fois g ;
  * $f(n) = \Theta(g(n))$ signifie que f et g sont asymptotiquement du même ordre de grandeur.
- On distingue la complexité dans le **pire cas**, le **meilleur cas**, **moyenne** ou **amortie**. Ces deux dernières ne sont pas à confondre : la complexité moyenne considère la moyenne sur toutes les entrées d'une certaine taille, alors que la complexité amortie est calculée sur un grand nombre d'opérations successives.
- Pour déterminer la complexité d'un algorithme, on compte un ordre de grandeur du nombre d'opérations élémentaires : écriture/lecture d'une variable ou opération arithmétique.
- Pour une fonction avec des boucles, on peut calculer sous forme de somme. Par exemple `for i = 0 to n - 1 do for j = i + 1 to n - 1 do` résultera en $\mathcal{O}\left(\sum_{i=0}^{n-1}(n-i-1)\right) = \mathcal{O}(n^2)$ (à multiplier par la complexité du corps de la boucle).
- Pour une fonction récursive, on peut trouver une formule de récurrence vérifiée par le nombre d'opérations élémentaires, sans oublier les cas d'arrêts de la fonction. On présente quelques formules classiques et formules closes associées. Les Θ peuvent être remplacés par des $\mathcal{O}$. La valeur dans les Θ correspond à la complexité temporelle des calculs supplémentaires à chaque appel récursif :
  * $C(n) = C(n-1) + \Theta(1)$ se résout en $C(n) = \Theta(n)$;
  * $C(n) = C(n-1) + \Theta(n^k)$ plus généralement se résout en $C(n) = \Theta(n^{k+1})$;
  * $C(n) = C(n/2) + \Theta(1)$ se résout en $C(n) = \Theta(\log n)$;
  * $C(n) = C(n/2) + \Theta(n)$ se résout en $C(n) = \Theta(n)$;
  * $C(n) = 2C(n/2) + \Theta(1)$ se résout en $C(n) = \Theta(n)$;
  * $C(n) = 2C(n/2) + \Theta(n)$ se résout en $C(n) = \Theta(n \log n)$.

2 Gestion de la mémoire

2.1 Représentation des nombres

- Un entier **non signé** est représenté en machine comme sa représentation binaire sur un certain nombre de bits. C'est-à-dire que si $n = \sum_{i=0}^{m} b_i 2^i$ avec les $b_i \in \{0, 1\}$, alors n sera représenté en $b_0 b_1 \dots b_m b_{m+1} \dots b_{k-1}$ sur k bits. Cela peut entraîner un dépassement d'entier si $m \geqslant k$ (les entiers sont représentés modulo 2^k).
- Un entier **signé** est représenté par son **complément à deux** : un entier $n \in [\![-2^{k-1}, 2^{k-1} - 1]\!]$ est représenté sur k bits par :
  - la représentation non signée sur k bits de n si $n \geqslant 0$;
  - la représentation non signée de $2^k + n$ si $n < 0$.
- Un nombre est dit **dyadique** s'il peut s'écrire comme un entier divisé par une puissance de deux. Les flottants représentables en machine avec les normes usuelles sont tous dyadiques.
- La représentation machine d'un flottant x est composée d'un bit de **signe**, de plusieurs bits d'**exposant** et de plusieurs bits de **mantisse** :

s	e	m

  La valeur associée est alors $x = (-1)^s \times (1, m)_2 \times 2^e$. Certaines normes utilisent un **exposant décalé** d'une certaine valeur.

2.2 Pointeurs et portée des variables

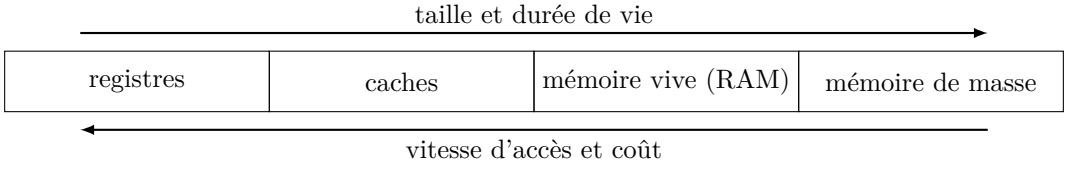
- Une **variable** lie un identifiant (son nom), un objet (sa valeur) et un type (son type).
- La **portée** d'un identifiant commence là où le nom est défini et s'arrête à la fin du programme pour une variable globale ou à la fin du bloc (structure de contrôle, fonction, ...) pour une variable locale.
- Les objets sont stockées dans la mémoire à des **adresses mémoire** qu'on manipule à l'aide de **pointeurs**.
- En C, il existe une adresse particulière `NULL`, qui ne désigne aucun emplacement mémoire.
- Une copie superficielle consiste à recopier les adresses mémoires et peut créer de la liaison de données.

2.3 Modèle mémoire lors de l'exécution d'un programme

- On distingue trois zones mémoire dans l'exécution d'un programme. La durée de vie d'un objet dépend de la manière dont il a été alloué.
  - la **zone statique**, où sont écrites les variables globales ;
  - la **pile**, où sont écrites les variables locales ;
  - le **tas** (qui n'est pas un tas au sens arborescent), où est allouée dynamiquement la mémoire.
- La zone statique existe pendant toute la durée de vie du programme.
- La pile est de taille limitée. Il peut y avoir un débordement de pile (*stack overflow*) si on dépasse l'espace limite, par exemple lors d'un trop grand nombre d'appels récursifs imbriqués.
- Les objets alloués sur le tas (avec `malloc`) existent jusqu'à être explicitement désallouées (`free`).
- Accéder à un objet en dehors de sa durée de vie (sur la pile ou le tas) entraîne un comportement indéfini.
- En OCaml, le **GC** (*Garbage Collector*) gère l'allocation mémoire automatiquement.

2.4 Modèle physique

La mémoire physique est découpée en plusieurs zones, chacune avec des avantages et des inconvénients. Les registres sont au niveau du processeur, la mémoire de masse dans les disques durs.



3 Structures de données

3.1 Type abstrait et implémentation

Dans la description d'une structure de données, on distingue le **type abstrait** et l'**implémentation** :

- le type abstrait correspond à la description théorique de ce que doit contenir la structure de données, ainsi que les opérations qu'on souhaite effectuer dessus. On peut parfois décrire la complexité attendue pour chacune des opérations, mais la complexité est généralement associée à une implémentation donnée ;
- l'implémentation correspond à la réalisation effective de la structure de données, c'est-à-dire la manière de la représenter en machine, ainsi que la description des algorithmes associés à chacune des opérations, et leurs complexités.

3.2 Quelques structures et implémentations associées

On considère qu'à chaque structure de données est associée une opération de création de structure vide et de test d'égalité à la structure vide.

- Une **pile** est une structure linéaire. Les opérations associées sont **empiler** et **dépiler**, dans l'ordre LIFO (*Last In First Out*). Dépiler supprime et renvoie le dernier élément empilé dans la pile. Les implémentations possibles sont :
  - liste chaînée ;
  - tableau (pile de taille bornée).

  Les complexités sont en $\mathcal{O}(1)$ dans les deux cas pour empiler et dépiler.
- Une **file** est une structure linéaire. Les opérations associées sont **enfiler** et **défiler**, dans l'ordre FIFO (*First In First Out*). Défiler supprime et renvoie l'élément qui est dans la file depuis le plus longtemps. Les implémentations classiques possibles sont :
  - deux piles (une pile d'enfilage et une pile de défilage : on vide la pile d'enfilage dans la pile de défilage lorsque cette dernière est vide et qu'on souhaite défiler) ;
  - tableau circulaire (file de taille bornée) ;
  - liste chaînée avec un pointeur vers le maillon de début et le maillon de fin (on enfile à la fin et on défile au début de la liste chaînée).

  L'enfilage est toujours en $\mathcal{O}(1)$, le défilage est en $\mathcal{O}(1)$, mais en amorti pour la première implémentation et en pire cas pour les deux autres implémentations.
- Un **dictionnaire** ou **table d'association** permet les opérations suivantes : créer ou supprimer une association (clé, valeur), tester si une clé possède une valeur associée, renvoyer la valeur associée à une clé. Les implémentations possibles sont :
  - table de hachage : on utilise une **fonction de hachage** pour transformer les clés en entiers, qu'on utilise comme indices dans un tableau. Comme la fonction de hachage n'est pas nécessairement injective, il peut y avoir des cas de **collisions**. Ces dernières peuvent être gérées par **adressage ouvert** (on cherche des cases disponibles si la case prévue est déjà occupée) ou par **chaînage** (une case contient la liste chaînée des associations ayant le même haché). Toutes les opérations sont en $\mathcal{O}(1)$ en moyenne, mais en $\Theta(n)$ dans le pire cas, où n est le nombre total d'associations contenues dans la table.
  - arbre binaire de recherche (voir partie suivante). Les opérations sont linéaires en la hauteur de l'arbre dans le pire cas. Avec des arbres auto-équilibrés, elles sont en $\mathcal{O}(\log n)$, où n est le nombre total d'associations dans le dictionnaire.
- Une **file de priorité** permet les opérations suivantes : ajouter un couple (priorité, valeur) ou extraire le couple de priorité extrémale. L'extremum peut être un maximum ou un minimum, mais toujours le même pour une file de priorité donnée. On peut implémenter une file de priorité de manière abstraite par un **tas binaire** (voir partie suivante). Les opérations sont logarithmiques en la taille du tas dans le pire cas.

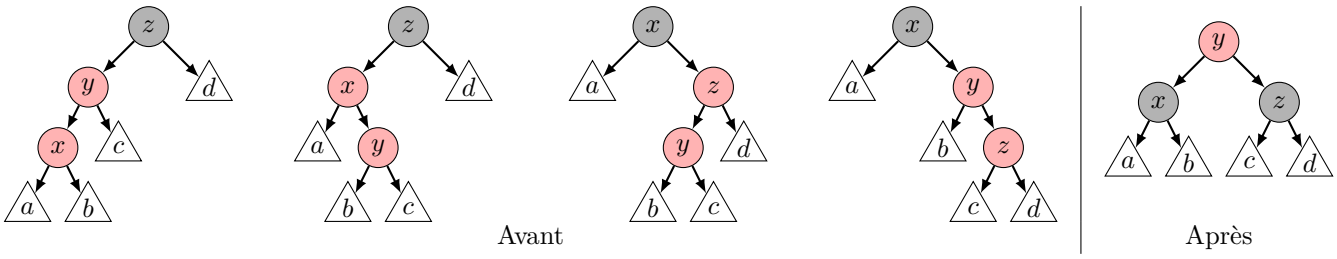
4 Arbres enracinés

4.1 Définitions, relations et implémentations

- Un **arbre enraciné** (ou simplement arbre s'il n'y a pas d'ambiguïté) peut être vu comme un ensemble muni d'une opération de **parenté** vérifiant : il existe un unique nœud sans parent, appelé **racine** ; tout nœud non racine possède un unique parent ; pour tout nœud, il existe une unique suite d'ancêtres menant à la racine, chaque nœud étant le parent du précédent. Les enfants d'un nœud peuvent être ordonnés ou non.
- On définit par induction les **arbres binaires** :
  - l'arbre **vide** est binaire ;
  - une racine ayant pour enfants gauche et droit deux arbres binaires est un arbre binaire.
- L'arbre vide est de **taille** 0 et de **hauteur** -1 (par convention). Pour un arbre binaire $a = N(g, d)$, on a les relations $|a| = |g| + |d| + 1$ et $h(a) = \max(h(g), h(d)) + 1$.
- Un arbre binaire a vérifie $h(a) + 1 \leqslant |a| \leqslant 2^{h(a)+1} - 1$.
- Le **parcours en profondeur** parcourt inductivement de gauche à droite les enfants d'un nœud :
  - parcours **préfixe** : la racine est placée avant les parcours des enfants ;
  - parcours **infixe** (cas binaire uniquement) : la racine est placée entre les parcours des deux enfants ;
  - parcours **postfixe** : la racine est placée après les parcours des enfants.
- Le parcours en **largeur** parcourt les nœuds de gauche à droite, par profondeur croissante.

4.2 Arbres binaires de recherche

- Un **arbre binaire de recherche** (ABR) est un arbre binaire étiqueté par des éléments dans un ensemble totalement ordonné tel que pour tout nœud x, tous les nœuds de l'enfant gauche de x sont $\leqslant x$ et tous les nœuds de l'enfant droit de x sont $\geqslant x$. Les inégalités peuvent être strictes selon les définitions.
- Un arbre est ABR si et seulement si son parcours infixe est croissant.
- Sans optimisation particulière, on effectue les opérations suivantes :
  - la recherche d'un élément x se fait en descendant dans l'arbre, en choisissant de continuer à gauche ou à droite selon la comparaison de la racine à x ;
  - l'insertion se fait de la même manière, jusqu'à atteindre un arbre vide, remplacé par une feuille ;
  - la suppression peut se faire en remplaçant le nœud supprimé par le plus grand élément de l'enfant gauche (ou l'autre enfant si l'un des enfants est vide) ;
  - toutes ces opérations sont linéaires en la hauteur de l'ABR, qui peut être linéaire en la taille.
- Les arbres **rouge-noir** (ou bicolores) permettent d'améliorer la complexité dans le pire cas des opérations.
  - Un arbre rouge-noir est un ABR dont les nœuds sont colorés en rouge ou noir et vérifiant les propriétés suivantes :
    - la racine est noire ;
    - un nœud rouge n'a pas d'enfant rouge ;
    - tous les chemins depuis un nœud donné jusqu'à un descendant vide contiennent le même nombre de nœuds noirs.
  - Un arbre rouge-noir a vérifie $\log_2(|a| + 1) - 1 \leqslant h(a) \leqslant 2\log_2(|a|) + 1$.
  - Insertion :
    - la feuille insérée est colorée en rouge ;
    - on effectue un rééquilibrage de tous les nœuds parcourus depuis la feuille créée jusqu'à la racine, tant qu'il existe un nœud rouge qui possède un enfant rouge ;
    - si un nœud rouge a un enfant rouge, on effectue des rotations pour rééquilibrer et un changement des couleurs, selon les cas :



— si la racine devient rouge, on la colore en noir.

* L'opération de suppression est plus complexe et peut être réalisée de différentes manières. Une possibilité est de créer un éventuel nœud **double-noir**, qui compte pour deux nœuds noirs dans le décompte du nombre de nœuds noirs d'un nœud jusqu'à un descendant vide. Le rééquilibrage consiste alors à faire remonter le nœud double-noir jusqu'à ce qu'il puisse disparaitre (si un nœud rouge y possède un fils double-noir x et un fils noir z ayant deux fils noirs, on peut transformer x et y en noir et z en rouge), ou qu'il atteigne la racine (qu'on recolore alors en noir). On ne détaille pas tous les cas de rééquilibrage.

4.3 Tas

On décrit ici les opérations pour un tas-max, permettant l'extraction de l'élément de priorité maximale.

- Un tas binaire est un arbre binaire **quasi-complet** – tous les niveaux de profondeurs sont remplis, sauf éventuellement le dernier, rempli par la gauche – et **tournoi** – un nœud est plus petit que son parent.
- On peut implémenter un tas binaire de taille bornée dans un tableau :
  * la racine du tas est à l'indice 0 ;
  * les enfants gauche et droit du nœud d'indice i sont aux indices $2i+1$ et $2i+2$ respectivement ;
  * le parent du nœud d'indice i est à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$.
- L'opération de **remontée** d'un nœud consiste à l'échanger avec son parent tant que l'ordre tournoi n'est pas respecté.
- L'opération de **descente** d'un nœud consiste à l'échanger avec son plus grand enfant tant que l'ordre tournoi n'est pas respecté.
- Pour insérer un élément, on l'ajoute à la première position de feuille disponible (celle d'indice n pour un tas de taille n), puis on le fait remonter. La complexité est $\mathcal{O}(\log n)$.
- Pour extraire le plus grand élément, qui se trouve à la racine, on l'échange avec la dernière feuille (celle d'indice $n-1$), puis on fait descendre la nouvelle racine. La complexité est $\mathcal{O}(\log n)$.
- Pour transformer un tableau de taille n en tas, on fait descendre ses éléments successivement depuis la fin. La complexité est $\mathcal{O}(n)$.
- Le **tri par tas** trie un tableau de taille n en temps $\mathcal{O}(n \log n)$: on transforme le tableau en tas, puis on effectue n extractions successives..

4.4 Union-Find

La structure *Union-Find* permet de gérer des unions et recherches de représentants dans une partition. On peut l'implémenter par une forêt (ensemble d'arbres).

- Chaque arbre correspond à une classe d'équivalence. La racine de chaque arbre est le représentant.
- On peut implémenter cette structure avec deux tableaux :
  * un tableau de parenté p, où $p[x]$ est l'indice du parent de x (ou -1 pour une racine) ;
  * un tableau de tailles t, où $t[x]$ est la taille du sous-arbre enraciné en x.
- L'opération *Find* consiste à trouver le représentant de la classe d'un élément. Tant qu'on n'a pas atteint la racine, on remonte l'arbre. Le parent de tous les ancêtres ainsi parcourus devient le représentant.
- L'opération *Union* consiste à fusionner les classes d'équivalence de deux éléments. On commence par trouver les représentants, puis celui dont la classe est la plus grosse devient le parent de l'autre.
- Les complexités sont en $\mathcal{O}(\alpha(n))$ où n est la taille de l'ensemble partitionné et α une fonction à croissance extrêmement lente ($\alpha(n)$ vaut moins de 4 en pratique).

5 Graphes

5.1 Représentations

- Un **graphe** est un couple $G = (S, A)$ où S est un ensemble fini de **sommets** et A est un ensemble d'**arêtes**.
- G est dit **orienté** si $A \subseteq S \times S$. Une arête (s, s) est appelée **boucle**.
- G est dit non **orienté** si $A \subseteq \mathcal{P}_2(S)$.
- Un **chemin** est une suite de sommets, chacun étant voisin du précédent. Le chemin est **simple** si on ne passe pas deux fois par la même arête, **élémentaire** si on ne passe pas deux fois par le même sommet. On peut voir un chemin non vide comme la suite des arêtes qui le composent.
- On peut implémenter en machine un graphe par tableau de listes d'adjacence : pour $s \in S$, $G[s]$ est la liste des **voisins** de s.
  - Avantage : gain en espace mémoire, accès en temps constant aux voisins d'un sommet.
  - Inconvénient : test d'existence d'une arête en temps linéaire.
- On peut implémenter G par matrice d'adjacence : pour $(s, t) \in S^2$, $G[s][t]$ est un booléen qui vaut « vrai » si et seulement si (s, t) (cas orienté) ou $\{s, t\}$ (cas non orienté) est dans A.
  - Avantage : test d'existence d'une arête en temps constant.
  - Inconvénient : espace mémoire toujours en $\Theta(|S|^2)$, accès aux voisins en temps linéaire.
- Un **graphe pondéré** est un triplet $G = (S, A, f)$ où (S, A) est un graphe et $f : A \to \mathbb{R}$ est une fonction de pondération.
- On peut implémenter un graphe pondéré par tableau de listes d'adjacence (les éléments d'une liste sont alors des couples (voisin, poids)) ou par matrice d'adjacence ($G[s][t]$ est égal au poids de l'arête de s à t ou 0 si $s = t$ ou $+\infty$ si l'arête n'existe pas).

5.2 Parcours de graphes

- Principe d'un parcours : traiter un sommet s'il n'est pas déjà traité puis relancer un parcours depuis ses voisins.
- Complexité : $\mathcal{O}\left(|S| + \sum_{s \in S} \deg(s)\right) = \mathcal{O}(|S| + |A|)$ avec des listes d'adjacence.
- Parcours en profondeur, DFS :
  - on fait le parcours du premier voisin avant de passer aux autres ;
  - permet de trouver un ordre topologique dans un graphe orienté sans cycle (l'ordre inverse d'un DFS postfixe) ;
  - applications : calcul des composantes connexes, détection de cycles, algorithme de Kosaraju, qui consiste à faire un parcours de graphe selon un ordre topologique du graphe transposé, pour trouver les composantes fortement connexes.
- Parcours en largeur, BFS :
  - on ajoute les voisins dans une file (FIFO) qui détermine l'ordre de traitement ;
  - applications : calcul des composantes connexes, recherche des plus courts chemin dans un graphe non pondéré.

5.3 Plus courts chemins pondérés

On peut adapter chaque algorithme pour trouver des chemins plutôt que des distances, en travaillant avec des tableaux de prédécesseurs.

- Algorithme de Dijkstra :
  - trouve les distances depuis un sommet de départ à tous les autres ;
  - applicable uniquement dans un graphe à poids positifs ;
  - à chaque itération, on extrait le sommet le plus proche du sommet de départ, et on met à jour la distance pour ses voisins ;
  - complexité : $\mathcal{O}((|S| + |A|) \log |S|)$ avec des listes d'adjacence.
- Algorithme de Floyd-Warshall :
  - trouve toutes les distances entre deux sommets quelconques ;

    - applicable uniquement dans un graphe orienté sans cycle négatif ou dans un graphe non orienté à poids positifs ;
    - algorithme de programmation dynamique : le plus court chemin de s à t ne passant que par des sommets intermédiaires inférieurs ou égaux à u est soit un plus court chemin de s à t ne passant que par des sommets intermédiaires strictement inférieurs à u, soit la concaténation d'un plus court chemin de s à u et de u à t, chacun ne passant que par des sommets intermédiaires strictement inférieurs à u ;
    - complexité : $\mathcal{O}(|S|^3)$ avec une matrice d'adjacence.
- Algorithme A^* (*a-star* ou *a-étoile*) :
    - trouve la distance entre un sommet de départ et un sommet d'arrivée ;
    - à chaque itération, on extrait le sommet qui minimise la somme entre la distance depuis le sommet de départ et l'estimation donnée par une heuristique au sommet d'arrivée ;
    - si l'heuristique est admissible (c'est-à-dire optimiste dans son estimation), l'algorithme est exact ;
    - complexité : comme Dijkstra si l'heuristique est consistante (respecte une inégalité triangulaire), peut être exponentielle sinon.

5.4 Arbres couvrants

On ne considère ici que des graphes non orientés (pondérés ou non).

- Un **arbre** est un graphe qui satisfait les conditions suivantes (deux suffisent et entraînent la troisième) : connexe, sans cycle et $|A| = |S| - 1$.
- Un arbre **couvrant** est un sous-graphe qui est un arbre et contient tous les sommets.
- L'algorithme de **Kruskal** permet de trouver un arbre couvrant de poids minimal :
    - on part d'un graphe vide puis on parcourt les arêtes par poids croissant et on garde celles qui ne créent pas de cycle ;
    - on peut utiliser une structure Union-Find pour garder en mémoire les composantes connexes et tester la création d'un cycle ;
    - complexité : $\mathcal{O}(|A| \log |S|)$.

5.5 Couplages

On ne considère ici que des graphes non orientés, non pondérés.

- Un **couplage** est un sous-ensemble d'arêtes C tel que deux arêtes de C n'ont jamais de sommet en commun.
- Un couplage est **maximal** (pour l'inclusion) si l'ajout d'une arête fait que ce n'est plus un couplage.
- Un couplage est (de cardinal) **maximum** s'il est de cardinal maximum parmi tous les couplages.
- Un chemin **augmentant** pour un couplage C est un chemin qui commence et termine par une arête hors de C et alterne entre une arête hors de C et une arête de C.
- Théorème de Berge : un couplage est maximum si et seulement s'il n'existe pas de chemin augmentant.
- Recherche d'un couplage maximum : tant qu'il existe un chemin augmentant σ, on calcule un nouveau couplage $C' = C \Delta \sigma$, où Δ représente la différence symétrique.
- Dans le cas particulier des graphes bipartis, un chemin augmentant peut être trouvé par un parcours de graphe particulier (qui part et termine d'un sommet non couvert et alterne les arêtes).

6 Paradigmes algorithmiques

6.1 Algorithmes gloutons

- Un **algorithme glouton** est généralement une manière intuitive de résoudre un problème :
  - on construit une solution petit à petit, sans jamais revenir sur des choix faits précédemment ;
  - on cherche un optimum local à chaque itération.
- Exemples : algorithme de Kruskal (optimal), construction de l'arbre de Huffman (optimal), rendu de monnaie (optimal ou non selon le système monétaire), voyageur de commerce (non optimal).

6.2 Diviser pour régner

- Un algorithme de type **diviser pour régner** consiste à découper un problème en nouvelles instances plus petites, les résoudre récursivement, puis reconstruire la solution :
  - les sous-problèmes ont une taille qui doit être une **fraction** de la taille initiale ;
  - les sous-problèmes sont **indépendants** les uns des autres.
- La complexité vérifie généralement $C(n) = aC(n/b) + f(n)$ où a est le nombre de sous-problèmes, n/b leurs tailles et $f(n)$ la complexité du calcul des sous-problèmes et de la construction de solution.
- Exemples : recherche dichotomique dans un tableau trié, tri fusion, tri rapide.

6.3 Programmation dynamique

- Un algorithme de **programmation dynamique** consiste à résoudre récursivement un problème à partir des solutions de sous-problèmes, en gardant en mémoire les solutions des sous-problèmes, qui ne sont pas nécessairement indépendants, pour éviter de les recalculer.
- Approche *Bottom-up* : calcul des sous-problèmes dans le bon ordre (à déterminer), approche *Top-down* : calcul récursif de la solution en **mémoïsant** les résultats (même ceux qui ne serviront plus), en utilisant une table de hachage, par exemple.
- Exemples : algorithme de Floyd-Warshall, rendu de monnaie, sac à dos.

6.4 Algorithmes probabilistes

- Algorithme **Monte Carlo** : temps de calcul toujours identique, mais résultat variable. Exemples : test de primalité (Fermat, Miller-Rabin), recherche d'une coupe minimale (Karger).
- Algorithme **Las Vegas** : résultat correct, mais temps de calcul variable. Exemple : tri rapide randomisé.
- On peut transformer un algorithme Monte Carlo qui résout un problème en algorithme Las Vegas si on a un moyen de vérifier la solution : on recommence tant que la réponse est incorrecte.

6.5 Approximations

- Un algorithme qui résout un problème d'optimisation est une α-**approximation** si la solution renvoyée est à un facteur α près d'une solution optimale.
- Un algorithme glouton peut parfois être un algorithme d'approximation (exemple : sac à dos où on prend toujours l'objet avec le meilleur ratio valeur/poids).

6.6 Backtracking et Branch & Bound

- Un algorithme de ***backtracking*** construit une solution satisfaisant des contraintes petit à petit en explorant en profondeur l'arbre des possibilités de construction, en revenant en arrière lorsqu'on réalise qu'une solution ne peut pas être continuée. Exemples : N-reines, résolution de sudoku.
- Un algorithme de ***branch and bound*** résout un problème d'optimisation. On utilise :
  - une **heuristique d'évaluation**, qui évalue une solution partielle. En comparant la valeur avec l'optimum courant, on peut détecter qu'une solution partielle ne peut pas améliorer l'optimum ;
  - une **heuristique de branchement**, qui permet de déterminer de quelle manière commencer à compléter la solution partielle, en espérant ne pas avoir à explorer les autres manières.
- Exemple : voyageur de commerce ; poids partiel et plus proche voisin comme évaluation et branchement.

7 Algorithmique du texte

7.1 Recherche de mot

Dans cette partie, on cherche un mot M de taille p dans un texte T de taille n. On utilise une notation similaire au *slicing* de Python : $T[i:j]$ signifie le facteur de T de l'indice i (inclus) à j (exclu).

- L'algorithme naïf consiste à tester, pour chaque indice $i \in [\![0, n-p]\!]$, si M apparaît dans T depuis l'indice i. Sa complexité dans le pire cas est $\Theta(np)$.
- L'algorithme de **Boyer-Moore(-Horspool)** effectue un pré-traitement sur le mot M, pour sauter des indices dans l'application de l'algorithme naïf :
  - on effectue la comparaison entre lettres de la droite vers la gauche (on commence par comparer $M[p-1]$ et $T[i+p-1]$ et on termine par $M[0]$ et $T[i]$) ;
  - si on constate $M[j] \neq T[i+j]$:
    - si $T[i+j]$ n'apparaît pas dans M, on peut recommencer à $i+p$;
    - si $T[i+j]$ apparaît dans M, on aligne $T[i+j]$ et sa dernière occurrence avant l'indice j dans M ;
    - on peut améliorer ce principe en considérant les occurrences de $T[i+j+1:i+p]$ dans M.
  - complexité : $\mathcal{O}(p+n)$ en moyenne, $\mathcal{O}\left(p + \frac{n}{p}\right)$ dans le meilleur cas.
- L'algorithme de **Rabin-Karp** utilise une fonction de hachage pour éviter de vérifier des alignements incorrects dans l'algorithme naïf :
  - on commence par calculer un haché h_M de M et un haché h_T de $T[0:p]$;
  - si pour un indice i, $h_M = h_T$, on compare M et $T[i:i+p]$;
  - on met ensuite à jour h_T comme haché de $T[i+1:i+p+1]$;
  - on choisit la fonction de hachage pour limiter les collisions et faire une mise à jour en temps constant ;
  - complexité : $\mathcal{O}(p+n)$ en moyenne, $\Theta(pn)$ dans le pire cas.

7.2 Compression sans perte

- L'algorithme de **Huffman** part du principe que pour compresser un texte, il vaut mieux encoder les lettres les plus fréquentes par des petits mots de code. Il est optimal (en espace) pour la compression sans perte et construit un arbre permettant la compression et la décompression :
  - la construction de l'arbre de Huffman est un algorithme glouton :
    - créer une file de priorité des couples $(\text{freq}(a), \texttt{Feuille}(a))$ pour chaque lettre a ;
    - tant qu'il reste au moins deux arbres, extraire les deux de plus petite fréquence (f_1, A_1) et (f_2, A_2) et insérer $(f_1+f_2, \texttt{Nœud}(A_1, A_2))$;
    - l'arbre final est l'arbre de Huffman ;
  - l'encodage d'une lettre se fait en suivant le chemin de la racine jusqu'à la feuille qui contient cette lettre : un 0 pour une branche gauche, un 1 pour une branche droite ;
  - la décompression se fait en lisant le texte compressé : si on lit un 0 (resp. 1), on emprunte la branche de gauche (resp. droite), si on arrive sur une feuille, on écrit la lettre et on remonte à la racine.
- L'algorithme de **Lempel-Ziv-Welch** garde en mémoire tous les motifs déjà rencontrés pour les encoder efficacement (sur une taille fixe) :
  - on se fixe une table initiale de taille (qui servira comme point de départ pour la compression et la décompression) ;
  - pour compresser un texte T, en commençant à $i=0$:
    - on calcule le plus long mot $T[i:i+j]$ encodé dans la table et on écrit le code associé ;
    - on ajoute un nouveau code pour $T[i:i+j+1]$;
    - on modifie i en $i+j$.
  - pour décompresser un texte codé, on recrée la table petit à petit en lisant les codes :
    - on lit le premier code, associé au mot u (nécessairement présent dans la table), si le deuxième code n'est pas dans la table, on ajoute un code pour $uu[0]$;
    - tant que possible, on lit le code suivant, associé à un mot xv, puis on ajoute à la table un code pour ux, puis on modifie u en xv.

8 Apprentissage automatique et théorie des jeux

On utilise k, κ et K pour désigner des notions différentes (mais toujours notées k dans la littérature).

8.1 Apprentissage supervisé

- L'algorithme des **k-plus proches voisins** permet de faire de la classification (ou de la régression) à partir d'un ensemble de données étiquetées :
  - on cherche les k plus proches voisins (selon une distance euclidienne, Manhattan, ...) ;
  - on renvoie la classe majoritaire parmi ces voisins (classification) ou la moyenne des valeurs associées à ces voisins (régression).
- Un arbre **κ-dimensionnel** peut accélérer la recherche des plus proches voisins :
  - on crée un arbre binaire où chaque nœud interne est associé à une donnée étiquetée, et dont les enfants contiennent les données de part et d'autre d'un hyperplan passant par le nœud, en changeant de vecteur normal selon la profondeur (en cyclant sur les κ dimensions) ;
  - lorsqu'on cherche les plus proches voisins de x, on fait un parcours d'arbre, et à chaque nœud, on commence par explorer le demi-espace où se trouve x, puis on explore l'autre demi-espace si le point le plus loin parmi les k plus proches actuels est à une distance supérieure à celle entre x et l'hyperplan.
- L'algorithme **ID3** (*Iterative Dichotomizer 3*) permet la construction d'un arbre de décision :
  - l'**entropie** d'une partition $\{C_1, \dots, C_m\}$ d'un ensemble E de N est $H(E) = -\sum_{j=1}^{m} \frac{|C_j|}{N} \log\left(\frac{|C_j|}{N}\right)$;
  - si tous les éléments de E sont de la même classe, on crée une feuille correspondant à cette classe ;
  - si $E = \emptyset$, on crée une feuille correspondant à la classe majoritaire du nœud parent ;
  - si tous les éléments ont les mêmes attributs, on crée une feuille correspondant à la classe majoritaire ;
  - sinon, on choisit l'attribut a qui maximise le gain et on crée un nœud d'attribut a et pour enfants les arbres de décisions créés à partir des sous-ensembles de E regroupés selon les valeurs de a, le gain étant la différence entre l'entropie de E et la moyenne pondérée des entropies des sous-ensembles.
- La **matrice de confusion** permet d'évaluer la pertinence d'un algorithme : on utilise un jeu de test (différent du jeu d'apprentissage) et un coefficient m_{ij} correspond au nombre de données classées comme i alors qu'elles auraient dues être classées comme j. Les erreurs sont les coefficients en dehors de la diagonale.
- Il y a un risque de **surrapprentissage** si on essaie de trop coller aux données d'apprentissage, car on donne alors plus d'importance au bruit lié au choix aléatoire des données qu'au signal réel (exemple : renvoyer une interpolation au lieu d'une régression linéaire dans un nuage de points).

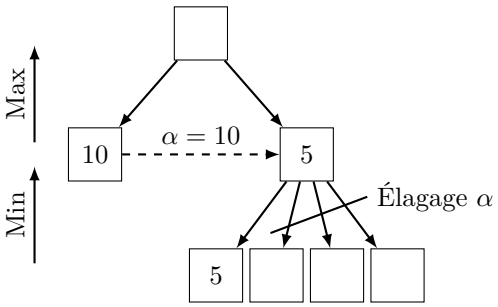
8.2 Apprentissage non supervisé

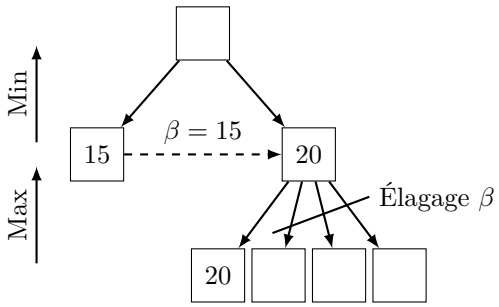
Les algorithmes au programme permettent de faire du partitionnement (ou *clustering*) d'un ensemble de données non étiquetées.

- Le **Clustering Hiérarchique Ascendant** (CHA) consiste à regrouper les données petit à petit :
  - à chaque itération, on fusionne les classes les plus proches, selon une distance de classe définie à l'avance. On peut par exemple utiliser la liaison simple (les classes dont les points sont les plus proches), la liaison complète (les classes dont les points les plus éloignés sont les plus proches) ou la liaison barycentrique (les classes dont les barycentres sont les plus proches), ...
  - on peut représenter les fusions par un **dendrogramme** : en abscisse les différentes données, et en ordonnées les distances de fusion ;
  - on arrête l'algorithme selon le critère de notre choix : nombre de classes obtenues, dépassement d'un seuil de fusion, diamètre d'une classe, ...
- L'algorithme des **K-moyennes** considère les centroïdes de chaque classe, qui sont mis à jour à chaque itération.
  - on initialise K centroïdes de manière aléatoire, soit avec des données, soit en prenant des points au hasard dans l'espace, puis on répète les deux étapes suivantes jusqu'à convergence (qui est assurée) :
    - on associe chaque donnée au centroïde le plus proche ;
    - on remplace chaque centroïde par le barycentre des données qui lui sont associées ;
  - il peut y avoir convergence vers un minimum local.

8.3 Théorie des jeux

- Un **jeu à deux joueurs** peut être représenté à l'aide d'un graphe orienté :
  - chaque sommet est une **position** du jeu, appartenant à un seul des deux joueurs ;
  - les voisins d'un sommet sont les positions atteignables en jouant les coups possibles ;
  - les puits (degré sortant nul) sont les positions **finales**, gagnante pour un joueur ou match nul ;
  - il y a une unique source (degré entrant nul) : la position **initiale** ;
  - le graphe est généralement biparti (les joueurs jouent à tour de rôle).
- Une **partie** est un chemin depuis la position initiale à une position finale.
- Une **stratégie** pour un joueur j est une fonction qui à chaque position du joueur renvoie l'un des voisins.
- Une stratégie f est **gagnante** pour j si quelle que soit la stratégie de l'adversaire, une partie en suivant f est gagnée par j.
- Une position est **gagnante** pour j s'il existe une stratégie gagnante pour j depuis cette position.
- On appelle **attracteur** de j l'ensemble des positions gagnantes pour j. Le calcul de l'attracteur de j peut se faire par récurrence :
  - on pose $A_0(j)$ l'ensemble des positions finales gagnantes pour j ;
  - pour $n \in \mathbb{N}$, $A_{n+1}(j)$ est l'union de $A_n(j)$, des sommets de j ayant un voisin dans $A_n(j)$ et des sommets de l'adversaire de j ayant tous leurs voisins dans $A_n(j)$;
  - $(A_n(j))_{n\in\mathbb{N}}$ est une suite ultimement stationnaire qui converge vers l'attracteur de j.
- On peut utiliser les attracteurs pour trouver une stratégie **optimale** (permettant de gagner lorsque c'est possible, ou de ne pas perdre lorsque c'est possible).
- Dans le cas où les attracteurs ne peuvent pas être calculés (car le graphe des positions est trop gros), on peut estimer la pertinence d'un coup avec l'algorithme **Min-Max** :
  - on choisit une profondeur d'exploration p ;
  - depuis une position s, on attribue un **score** à chaque sommet à distance $\leqslant p$;
  - si t est une position finale ou à distance $= p$, le score de t est donné par une **heuristique** ;
  - sinon, si t est une position du joueur 1, le score de t est le **maximum** des scores des voisins de t ;
  - sinon, le score de t le **minimum** des scores des voisins de t.
  - on choisit comme coup à jouer le voisin de s qui maximise le score.
- On peut accélérer le calcul de l'algorithme Min-Max par un **élagage alpha-beta** :
  - l'idée est similaire à celle d'un algorithme Branch and Bound ;
  - on garde en mémoire une borne inférieure α et une borne supérieure β du score du nœud racine ;
  - on coupe des branches si on peut conclure qu'on ne pourra pas améliorer le score :





9 Concurrence et synchronisation

9.1 Concurrence

- Un algorithme est **séquentiel** si ses instructions sont exécutées l'une après l'autre, toujours dans le même ordre. Il est **concurrent** si plusieurs **fils d'exécution** s'exécutent en parallèle, chacun étant séquentiel.
- Tout entrelacement d'instructions des fils d'exécutions est possible.
- Une opération est **atomique** si son exécution se fait toujours d'un seul bloc, sans interruption ou intercalage possible. L'incrémentation d'une variable n'est pas atomique.

9.2 Synchronisation

- La **section critique** d'un programme est une partie du programme où on lit ou écrit sur des ressources partagées. On parle de :
  - **exclusion mutuelle** si deux fils ne peuvent pas être simultanément dans la section critique ;
  - **absence d'interblocage** si lorsqu'au moins un fil tente d'accéder à la section critique, au moins un y parvient au bout d'un temps fini ;
  - **absence de famine** si chaque fil qui souhaite entrer dans la section critique y parvient au bout d'un temps fini.
- L'absence de famine implique l'absence d'interblocage.
- Un **mutex** est un objet de synchronisation permettant :
  - le verrouillage : aucun fil ne peut terminer le verrouillage d'un mutex déjà verrouillé ;
  - le déverrouillage : seul le fil qui a verrouillé le mutex peut le déverrouiller.
- On peut implémenter un mutex pour deux fils 0 et 1 avec l'algorithme de **Peterson** :
  - on utilise un tableau V (pour « veut entrer en section critique ») de 2 booléens (initialement faux) et une variable de tour t (initialement 0) ;
  - si le fil i veut verrouiller le mutex, il pose $V[i]$ à vrai, cède la priorité à l'autre (en posant $t = 1 - i$), puis attend jusqu'à ce que son tour vienne ($t = i$) ou que l'autre ait fini ($V[1 - i]$ vaut faux) ;
  - si le fil i veut déverrouiller le mutex, il pose $V[i]$ à faux.
- On peut implémenter un mutex pour n fils avec l'algorithme de **la boulangerie de Lamport** :
  - on utilise un tableau V de n booléens (initialement faux) et un tableau T (pour « ticket ») de n entiers (initialement 0) ;
  - si le fil i veut verrouiller le mutex, il pose $V[i]$ à vrai, puis $T[i]$ à $\max(T) + 1$, puis il attend tant qu'il existe un fil j tel que $V[j]$ vaut vrai et $(T[j], j) \prec (T[i], i)$ par ordre lexicographique ;
  - si le fil i veut déverrouiller le mutex, il pose $V[i]$ à faux.
- Les deux algorithmes précédents satisfont l'exclusion mutuelle, l'absence d'interblocage et de famine.
- Un **sémaphore** est un objet de synchronisation associé à un compteur permettant :
  - la décrémentation : si le compteur est strictement négatif après décrémentation, le fil qui a décrémenté est mis en attente ; on parle également d'acquérir ou d'attendre le sémaphore ;
  - l'incrémentation : après incrémentation, s'il y a au moins un fil en attente, un tel fil est réveillé ; on parle également de relâcher ou de signaler le sémaphore.
- Certains problèmes classiques peuvent être résolus avec des mutex ou des sémaphores :
  - le **dîner des philosophes** : en protégeant chaque baguette par un mutex, si, quand il veut manger, chaque philosophe s'empare de sa baguette de gauche, puis celle de droite, il y a interblocage ; une solution et de faire faire l'inverse au dernier philosophe, ou d'utiliser un sémaphore pour limiter à 4 les philosophes qui peuvent prendre une baguette ;
  - le **producteur-consommateur** : en utilisant deux sémaphores `ressources` (initialisé à 0) et `emplacements` (initialisé à la taille de la mémoire partagée), un producteur attend `emplacements` pour produire, et libère ensuite `ressources`, un consommateur attend `ressources` puis libère `emplacements` ;
  - le **rendez-vous** pour deux fils : en utilisant deux sémaphores initialisés à 0, chaque fil libère son sémaphore, puis attend celui de l'autre.

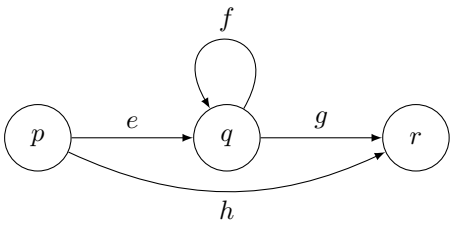
10 Théorie des langages

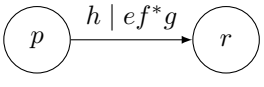
10.1 Automates finis

- Un **automate fini** est un quintuplet $A = (Q, \Sigma, \Delta, I, F)$ où Q est un ensemble d'**états**, Σ un **alphabet**, I un ensemble d'états **initiaux**, F un ensemble d'états **finaux** et $\Delta : Q \times \Sigma \to \mathcal{P}(Q)$ est une **fonction de transition**.
- On étend la définition de Δ à $\mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ par $\Delta(X, a) = \bigcup_{q \in X} \Delta(X, a)$.
- Si I est un singleton et $|\Delta(q, a)| \leqslant 1$, A est **déterministe**. Il est **non déterministe** sinon.
- La **fonction de transition étendue** est Δ^* définie par :
  - $\Delta^*(X, \varepsilon) = X$;
  - pour $u \in \Sigma^*$ et $a \in \Sigma$, $\Delta^*(X, au) = \Delta^*(\Delta(X, a), u)$.
- Un mot u est **reconnu** par A si $\Delta^*(I, u) \cap F \neq \emptyset$. On note $L(A)$ le langage reconnu par A. Deux automates sont **équivalents** s'ils reconnaissent le même langage.
- Pour tout automate déterministe, il existe :
  - un automate déterministe équivalent **complet** ($|\Delta(q, a)| = 1$) ;
  - un automate déterministe équivalent **émondé** (tout état appartient à un chemin acceptant).
- On peut étendre la définition de Δ en étiquetant les transitions par ε. Il existe toujours un automate non déterministe équivalent, avec autant d'états, sans ε-transition.
- Pour tout automate non déterministe, avec ou sans ε-transition, il existe un automate déterministe équivalent. L'algorithme des **parties** permet d'en construire un : on considère Δ comme une fonction déterministe sur un automate dont les états sont les éléments de $\mathcal{P}(Q)$.
- On peut implémenter un automate comme un graphe orienté dont les arêtes sont étiquetées.

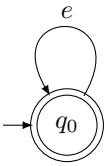
10.2 Langages rationnels

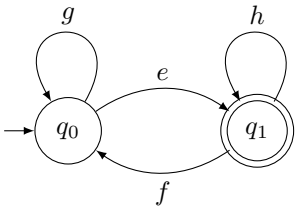
- L'ensemble des **langages rationnels** sur Σ est défini par induction :
  - $\emptyset$, $\{\varepsilon\}$ et $\{a\}$ pour tout $a \in \Sigma$ sont des langages rationnels ;
  - l'union, la concaténation et l'étoile de Kleene d'un nombre fini de langages rationnels sont des langages rationnels.
- On peut utiliser une **expression régulière** pour représenter un langage rationnel : on remplace $\{\varepsilon\}$ et $\{a\}$ par ε et a, et l'union par $|$ ou $+$.
- Le **théorème de Kleene** affirme qu'un langage est rationnel si et seulement s'il est reconnu par un automate fini.
- L'algorithme de **Berry-Sethi** permet de construire un automate à partir d'une expression régulière donnée :
  - on linéarise l'expression (on numérote les lettres pour les différencier) ;
  - on construit l'automate de **Glushkov** avec un état initial q_0, final si le mot vide est dans le langage, un état par lettre, final si c'est une dernière lettre, une transition de q_0 vers chacune des premières lettres, et une transition de a_i vers b_j, étiquetée par b, pour chaque facteur $a_i b_j$ de taille 2.
- L'algorithme de **Brzozowski-McCluskey** (ou **élimination des états**) permet de trouver une expression régulière à partir d'un automate donné :
  - quitte à rajouter deux états et des ε transitions, on se ramène à un automate avec un seul état initial et un seul état final ;
  - tant qu'il reste des états non initiaux et non finaux, on les élimine (attention, p peut être égal à r) :





  - après toutes les suppressions on obtient l'un des deux automates suivants ; l'expression régulière cherchée est e^* dans le premier cas et $(g|eh^\star f)^\star eh^\star$ dans le deuxième.





- D'autres méthodes existent pour faire les conversions, comme l'algorithme de **Thompson**, qui consiste à construire un automate avec ε-transitions à partir d'une expression régulière, ou l'algorithme de **McNaughton-Yamada** qui trouve une expression régulière à partir d'un automate, sur un principe similaire à l'algorithme de Floyd-Warshall.
- Les langages rationnels sont stables par intersection finie et complémentaire.
- On peut utiliser le **lemme de pompage** pour prouver qu'un langage n'est pas rationnel : pour tout langage rationnel L, il existe $n \in \mathbb{N}$ tel que si $u \in L$ vérifie $|u| \geqslant n$, alors il existe des mots x, y, z tels que $u = xyz$, $|xy| \leqslant n$, $|y| > 0$ et $xy^*z \subseteq L$.
- Exemples : $\{a^n b^n \mid n \in \mathbb{N}\}$, $\{uu \mid u \in \Sigma^*\}$ ou le langage des palindromes sont des langages non rationnels.

10.3 Grammaires hors-contextes

- Une **grammaire hors-contexte** est un quadruplet $G = (\Sigma, V, P, S)$ où Σ est un alphabet **terminal**, V est un alphabet de **variables** (ou **non-terminaux**), disjoint de Σ, $S \in V$ est la **variable de départ** (ou **axiome**) et P est un ensemble de **règles de productions** de la forme $X \to \alpha$ où $X \in V$ et $\alpha \in (\Sigma \cup V)^*$.
- Une **dérivation immédiate** $\Rightarrow$ applique une règle $X \to \alpha$ en remplaçant une occurrence de X par α.
- Une dérivation immédiate est **gauche** $\Rightarrow_g$ (resp. droite $\Rightarrow_d$) si la variable remplacée est la plus à gauche (resp. droite).
- Une **dérivation** $\Rightarrow^*$ est une suite de dérivations immédiates. Un mot $u \in \Sigma^*$ est **généré** par G s'il existe une dérivation $S \Rightarrow^* u$. On note $L(G)$ le langage **engendré** par G.
- Un langage est **algébrique** s'il est engendré par une grammaire hors-contexte. Tout langage rationnel est algébrique.
- Pour $u \in L(G)$, un **arbre de dérivation** de u est un arbre étiqueté vérifiant :
  - la racine est S ;
  - les nœuds internes sont étiquetés par des éléments de V ;
  - les feuilles sont étiquetées par des éléments de $\Sigma \cup \{\varepsilon\}$;
  - la lecture des feuilles de gauche à droite forme le mot u ;
  - un nœud interne X ayant pour enfants $\alpha_1, \ldots, \alpha_k$ vérifie $X \to \alpha_1 \ldots \alpha_k \in P$ (et $\alpha_i \neq \varepsilon$ si $k > 1$).
- G est dite **ambiguë** s'il existe $u \in L(G)$ possédant deux arbres de dérivation distincts, ou de manière équivalente deux dérivations gauches ou deux dérivations droites.
- Décider si une grammaire est ambiguë est un problème indécidable.
- L'analyse **syntaxique** est une étape indispensable lors de la compilation d'un programme. Elle consiste à déterminer un arbre de dérivation du programme dans la grammaire du langage :
  - on peut procéder par analyse descendante (construction de l'arbre depuis la racine) : en partant de la variable de départ, on essaie d'appliquer des règles de production par backtracking, jusqu'à obtenir le mot voulu, en revenant en arrière si on obtient les mauvais terminaux ;
  - on peut procéder par analyse ascendante (construction de l'arbre depuis les feuilles) : en partant du mot, on le réduit petit à petit en appliquant les règles inverses, jusqu'à obtenir la variable de départ ;
  - elle s'effectue après l'analyse **lexicale** qui consiste à découper le programme en une liste de **tokens**, qui forment l'alphabet terminal de la grammaire associée.

11 Logique

11.1 Logique propositionnelle

- La **syntaxe** des formules propositionnelles sur un ensemble de variables propositionnelles fini V est défini par induction par :
  - $\top$, $\bot$ et $x \in V$ sont des formules propositionnelles ;
  - la négation $\neg$, conjonction $\wedge$, disjonction $\vee$ et implication $\rightarrow$ d'une ou deux formules propositionnelles sont des formules propositionnelles.
- On peut associer un arbre binaire à une formule : les feuilles sont les variables, $\top$ et $\bot$; les nœuds internes sont les connecteurs logiques.
- Une **valuation** est une fonction $\mu : V \to \{0, 1\}$.
- La **sémantique** des formules propositionnelles est définie par induction en étendant la définition des valuations :
  - $\mu(\top) = 1$, $\mu(\bot) = 0$;
  - $\mu(\neg\varphi) = 1 - \mu(\varphi)$;
  - $\mu(\varphi \wedge \psi) = \min(\mu(\varphi), \mu(\psi))$;
  - $\mu(\varphi \vee \psi) = \max(\mu(\varphi), \mu(\psi))$;
  - $\mu(\varphi \rightarrow \psi) = \max(1 - \mu(\varphi), \mu(\psi))$.
- On peut représenter la sémantique d'une formule φ par une **table de vérité** : une colonne par variable et pour φ, une ligne par valuation.
- On appelle **modèle** d'une formule φ une valuation μ telle que $\mu(\varphi) = 1$. On note $\mathcal{M}(\varphi)$ l'ensemble des modèles de φ.
- On a les égalités : $\mathcal{M}(\neg\varphi) = \overline{\mathcal{M}(\varphi)}$; $\mathcal{M}(\varphi \wedge \psi) = \mathcal{M}(\varphi) \cap \mathcal{M}(\psi)$; $\mathcal{M}(\varphi \vee \psi) = \mathcal{M}(\varphi) \cup \mathcal{M}(\psi)$.
- φ est **sémantiquement équivalente** à ψ, noté $\varphi \equiv \psi$, si $\mathcal{M}(\varphi) = \mathcal{M}(\psi)$.
- φ **implique sémantiquement** ψ (ou ψ **conséquence logique** de φ), noté $\varphi \vDash \psi$, si $\mathcal{M}(\varphi) \subseteq \mathcal{M}(\psi)$.
- φ est une **tautologie**, noté $\vDash \varphi$, si toute valuation est un modèle de φ.
- φ est **satisfiable** si elle possède un modèle, c'est-à-dire $\mathcal{M}(\varphi) \neq \emptyset$. On appelle `SAT` le problème de décision déterminant si une formule est satisfiable ou non.
- Un **littéral** est une variable propositionnelle ou sa négation. Une **forme normale conjonctive** (FNC) est une conjonction de disjonction de littéraux. Une **forme normale disjonctive** (FND) est une disjonction de conjonction de littéraux.
- Toute formule est sémantiquement équivalente à une FNC et à une FND.
- k-`SAT` est le problème décidant si une FNC dont les clauses ont au plus k littéraux est satisfiable.
- L'algorithme de **Quine** permet de résoudre `SAT` pour une formule φ. Il consiste à construire un arbre binaire :
  - toute formule est sémantiquement équivalente à une formule **simplifiée** sans $\top$ ni $\bot$ ou à $\top$ ou à $\bot$, calculable en temps polynomial ;
  - la racine de l'arbre est la formule simplifiée de φ ;
  - pour une formule simplifiée $\neq \top$ et $\neq \bot$, on choisit une variable propositionnelle x apparaissant dans la formule, l'enfant gauche est l'arbre calculé récursivement avec la formule simplifiée où on a remplacé x par $\top$, l'enfant droit où on a remplacé x par $\bot$.
  - φ est satisfiable si l'arbre possède une feuille $\top$.

11.2 Logique du premier ordre

- Un **langage du premier ordre** est la donnée de symboles de fonctions et de relations. Une fonction d'arité 0 est une **constante**.
- Les **termes** sont définis par induction : les variables (non propositionnelles) et les constantes sont des termes, l'application d'une fonction à des termes est un terme.
- La syntaxe de la logique du premier ordre est définie par induction :
  - une formule atomique, c'est-à-dire l'application d'une relation à des termes est une formule du premier ordre ;
  - si φ et ψ sont des formules de la logique du premier ordre et x une variable (non propositionnelle), alors $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\exists x\, \varphi$ et $\forall x\, \varphi$ sont des formules du premier ordre.
- Si φ est une formule du premier ordre et x une variable :

  - une **occurrence** de x est une feuille étiquetée par x dans l'arbre syntaxique ;
  - une occurrence de x est **liée** si elle possède un ancêtre $\exists x$ ou $\forall x$; elle est **libre** sinon ;
  - la variable x (pas une occurrence) est **libre dans** φ si elle possède une occurrence libre ;
  - on note $V_F(\varphi)$ (F pour *Free*) l'ensemble des variables libres de φ.
- Pour φ une formule, x une variable et t un terme, la **substitution** de x par t dans φ, notée $\varphi[x := t]$, est obtenue en :
  - renommant toutes les occurrences liées dans φ d'une variable y libre dans t ;
  - remplaçant toutes les occurrences libres de x dans φ par t.

11.3 Déduction naturelle

- Un **séquent** $\Gamma \vdash \varphi$ est formé d'un ensemble de formules Γ (**contexte**) et d'une formule φ (**conclusion**).
- Une **règle d'inférence** $\dfrac{\Gamma_1 \vdash \varphi_1 \quad \Gamma_2 \vdash \varphi_2 \quad \dots \quad \Gamma_k \vdash \varphi_k}{\Gamma \vdash \varphi}$ est formée de **prémisses** $\Gamma_i \vdash \varphi_i$ et d'une **conclusion** $\Gamma \vdash \varphi$. Une règle sans prémisse est un **axiome**.
- Un **système déductif** est un ensemble de règles d'inférence. Une **preuve** d'un séquent $\Gamma \vdash \varphi$ dans un système déductif est un arbre vérifiant :
  - les nœuds internes sont des séquents, la racine est $\Gamma \vdash \varphi$, les feuilles sont des axiomes ;
  - si un nœud interne $\Gamma' \vdash \varphi'$ a pour enfants $\Gamma_1 \vdash \varphi_1$, …, $\Gamma_k \vdash \varphi_k$, alors il existe une règle d'inférence $\dfrac{\Gamma_1 \vdash \varphi_1 \quad \Gamma_2 \vdash \varphi_2 \quad \dots \quad \Gamma_k \vdash \varphi_k}{\Gamma' \vdash \varphi'}$ dans le système déductif.
- La logique classique est un système déductif possédant les règles suivantes :
  - axiome $\dfrac{}{\Gamma, \varphi \vdash \varphi}\ \text{ax}$, affaiblissement $\dfrac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ \text{aff}$;
  - l'une des règles parmi raisonnement par l'absurde $\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ raa, le tiers-exclu $\dfrac{}{\Gamma \vdash \varphi \vee \neg\varphi}\ \text{te}$ ou l'élimination de la double négation $\dfrac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}\ \neg\neg_e$
  - des règles d'introduction et d'élimination pour chaque connecteur logique :
    - implication : $\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ \to_i$ et $\dfrac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ \to_e$;
    - conjonction : $\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge_i$, $\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ \wedge_e^g$ et $\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ \wedge_e^d$;
    - disjonction : $\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^g$, $\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^d$ et $\dfrac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \sigma \quad \Gamma, \psi \vdash \sigma}{\Gamma \vdash \sigma}\ \vee_e$;
    - négation : $\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ \neg_i$ et $\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot}\ \neg_e$;
    - atomiques : $\dfrac{}{\Gamma \vdash \top}\ \top_i$ et $\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot_e$;
    - existentiel : $\dfrac{\Gamma \vdash \varphi[x := t]}{\Gamma \vdash \exists x \varphi}\ \exists_i$ et $\dfrac{\Gamma \vdash \exists x\, \varphi \quad \Gamma, \varphi \vdash \psi \quad x \notin V_F(\Gamma) \cup V_F(\psi)}{\Gamma \vdash \psi}\ \exists_e$;
    - universel : $\dfrac{\Gamma \vdash \varphi \quad x \notin V_F(\Gamma)}{\Gamma \vdash \forall x\, \varphi}\ \forall_i$ et $\dfrac{\Gamma \vdash \forall x\, \varphi}{\Gamma \vdash \varphi[x := t]}\ \forall_e$.
- La logique classique est :
  - **correcte** : si $\Gamma \vdash \varphi$ est prouvable, alors $\Gamma \vDash \varphi$ (la conjonction des formules de Γ implique sémantiquement φ) ;
  - **complète** : si $\Gamma \vDash \varphi$, alors $\Gamma \vdash \varphi$ est prouvable.

12 Calculabilité

12.1 Problèmes de décision

- Un **problème de décision** est la donnée d'une instance (la description d'un objet) et d'une question binaire. Une instance est **positive** si la réponse associée est vrai, **négative** sinon.
- La **taille** $|x|$ d'une instance x est l'espace mémoire nécessaire pour la représenter dans le modèle de calcul choisi. Dans le cas particulier d'un entier n, la mémoire occupée est en $\mathcal{O}(\log n)$, car cela correspond au nombre de bits utilisé pour écrire n.
- On peut assimiler un problème de décision A au sous-ensemble des instances positives de l'ensemble I_A des instances de A. Ainsi, pour une instance $x \in I_A$, $x \in A$ signifie que x est une instance positive.
- Un problème de décision A se **réduit many-one** à B, noté $A \leqslant_{\mathsf{m}} B$, s'il existe une fonction calculable $f : I_A \to I_B$ telle que $x \in A \Leftrightarrow f(x) \in B$. Elle est **polynomiale**, noté $\leqslant^{\mathsf{p}}_{\mathsf{m}}$, si de plus $f(x)$ peut se calculer en temps polynomial en $|x|$.

12.2 Décidabilité

- Un problème de décision est **décidable** s'il existe un algorithme qui permet de le résoudre dans le modèle de calcul choisi, c'est-à-dire un algorithme qui termine toujours en temps fini et répond correctement à la question du problème. Il est **indécidable sinon**.
- Le **problème de l'arrêt**, qui consiste à décider si une fonction termine sur un argument donné, est indécidable. On le prouve par l'absurde : si on suppose qu'il existe une fonction `arret` qui décide le problème, alors la fonction suivante ne peut pas exister, car elle termine et ne termine pas à la fois lorsqu'on l'appelle avec son propre code :

```
let rec paradoxe f = if arret f f then paradoxe f
```

- Si on suppose $A \leqslant_{\mathsf{m}} B$ et A indécidable, alors B est indécidable.

12.3 Classes de complexité

- P (*deterministic Polynomial*) est l'ensemble des problèmes de décision résolubles en temps polynomial en la taille de leur entrée.
- NP (*Nondeterministic Polynomial*) est l'ensemble des problèmes de décision A tels qu'il existe :
  - un problème $B \in \mathsf{P}$ tel que $I_B = I_A \times \{0,1\}^*$;
  - une fonction polynomiale f telle que pour $x \in I_A$:

$$x \in A \Leftrightarrow \exists c \in \{0,1\}^*, |c| \leqslant f(|x|) \wedge (x,c) \in B$$

  Autrement dit, on peut vérifier **en temps polynomial** que $x \in A$ en utilisant un certificat **de taille polynomiale**. Généralement, on utilise une description plus explicite du certificat plutôt qu'un élément de $\{0,1\}^*$ (qui peut permettre d'encoder toute description finie).
- Si $A \leqslant^{\mathsf{p}}_{\mathsf{m}} B$ et $B \in \mathsf{P}$ (resp. $B \in \mathsf{NP}$), alors $A \in \mathsf{P}$ (resp. $A \in \mathsf{NP}$).
- On dit que B est NP-**difficile** si pour tout $A \in \mathsf{NP}$, $A \leqslant^{\mathsf{p}}_{\mathsf{m}} B$. B est NP-**complet** s'il est NP-difficile et si $B \in \mathsf{NP}$.
- Théorème de **Cook-Levin** : `SAT` est NP-complet.
- $\mathsf{P} \subseteq \mathsf{NP}$; on ne sait actuellement pas conclure sur l'inclusion réciproque. Si on trouve un problème A qui est NP-complet et dans P, alors $\mathsf{P} = \mathsf{NP}$.
- Problèmes NP-complets classiques : `3-SAT`, `CLIQUE`, `COUVERTURE PAR SOMMETS`, `CYCLE HAMILTONIEN`, `SOMME PARTIELLE`, `3-COLORATION`, `VOYAGEUR DE COMMERCE`, `SAC À DOS`, ...
- Certains problèmes NP-complets qui ont comme instances des entiers, comme `SOMME PARTIELLE`, ont des algorithmes qui les résolvent de complexité polynomiale en la **valeur** de leur entrée. Cela ne montre pas que $\mathsf{P} = \mathsf{NP}$, car la valeur d'un entier est exponentielle en la mémoire nécessaire pour le représenter.

13 Bases de données

13.1 Représentation

- Un **schéma relationnel** est un n-uplet d'**attributs** (ou **champs**, ou **colonnes**), chaque attribut étant défini sur un **domaine** (exemples : entiers, chaînes de caractères, date, ...).
- Un **enregistrement** (ou **ligne**) est un n-uplet de valeurs associées à un schéma relationnel.
- Une **table** est un ensemble d'enregistrements, tous associés au même schéma relationnel.
- Une **base de données** est un ensemble de tables.
- Une clé **primaire** dans une table est un ensemble d'attributs minimal tel que deux enregistrements ne coïncident jamais sur la clé. Une **clé étrangère** est une clé qui est une clé primaire dans une autre table.
- Un **diagramme Entités-Associations** est une représentation graphique qui décrit des **entités** et les **relations** qui existent entre elles. À chaque relation est associée une **cardinalité** indiquant combien d'éléments de la première entité peuvent être en relation avec combien d'éléments de la deuxième. Les trois cardinalités à considérer sont $1-1$, $1-n$ et $n-n$ (n étant générique), une cardinalité $p-q$ signifiant qu'un élément de la première entité est associé au plus p fois et un élément de la deuxième au plus q fois.

- On peut transformer une relation $n-n$ en deux relations $1-n$ en créant une entité fictive intermédiaire.

13.2 Requêtes SQL

Ne sont au programme que les requêtes SQL qui renvoient des tables (pas de création/modification).

- Le schéma général d'une requête SQL est le suivant. On met entre crochets les mots-clés facultatifs. Seules les lignes `SELECT` et `FROM` sont obligatoires.

```
SELECT [DISTINCT] champ, expression, *, ... [AS nouveau nom]
FROM table, produit cartésien, sous-requête, ... [AS nouveau nom]
[LEFT] JOIN table, produit cartésien, sous-requête, ...
ON condition de jointure
WHERE condition sur les champs pris individuellement
GROUP BY champ, expression, ...
HAVING condition sur les groupes, sur des fonctions d'agrégation
ORDER BY champ, expression, fonction d'agrégation, sous-requête renvoyant une seule valeur... [DESC]
LIMIT nombre OFFSET nombre
```

- En cas d'ambiguïté sur un même nom de champ, on peut lever l'ambiguïté par `table.champ`.
- Un produit cartésien s'écrit simplement comme des noms de tables séparés par des virgules.
- `LEFT JOIN` correspond à une jointure gauche. La table résultante contient les entrées de la première table qui n'ont pas d'entrées dans la deuxième table respectant la condition de jointure, avec des `NULL` pour les champs de la deuxième table à la place.
- Les fonctions d'agrégation à connaître sont `COUNT`, `SUM`, `MIN`, `MAX` et `AVG`. On ne peut les utiliser que sur des données groupées. En l'absence de `GROUP BY`, le groupe est l'ensemble de la table. On ne peut pas imbriquer les fonctions d'agrégation, car elles s'appliqueraient sur des groupes différents.
- `COUNT(*)` compte toutes les lignes, `COUNT(champ)` compte toutes les lignes dont `champ` n'est pas `NULL`, et `COUNT(DISTINCT champ)` compte toutes les valeurs distinctes (non `NULL`) de `champ`.
- Une fois des données groupées, il n'est possible d'obtenir des valeurs d'un champ que si toutes les données d'un même groupe coïncident sur ce champ.
- `LIMIT p OFFSET q` supprime les q premiers enregistrements, puis ne garde que les p suivants.
- On peut faire des opérations ensemblistes entre deux tables (ou deux requêtes) ayant le même schéma relationnel avec `UNION`, `INTERSECT` et `EXCEPT` (ou `MINUS`). Ces opérations suppriment les doublons.
- On peut tester l'appartenance (ou non appartenance) avec `IN` (ou `NOT IN`).
- On teste si un champ est vide ou non avec `IS NULL` ou `IS NOT NULL`.

14 Programmation en C

`t` désigne un type générique, `v` une variable de type `t`, `ptr` un pointeur vers un objet de type `t`, `tab` un tableau d'éléments de type `t`, `x` une expression de type `t`, `n` un entier, `b` un booléen, `f` une fonction.

- Commentaires : après `//` ou entre `/*...*/`
- Entêtes : rajouter `#include <nom_entete.h>` en début de fichier. Entêtes usuelles : `assert.h`, `stdbool.h`, `stdio.h`, `stdlib.h`, `limits.h`.
- Types de base :
  - entiers : `int`, `unsigned int`, `uint_8/32/64` (dans `stdint.h`)
  - flottants : `double`
  - booléens : `bool`, valeurs `true` et `false`
  - caractères : `char`, valeurs `'a'`, `'b'`, ...
- Opérations élémentaires :
  - opérations arithmétiques : `+`, `-`, `*`
  - division entière ou flottante : `/`
  - modulo : `%`
  - opérateurs booléens : `&&`, `||`, `!`
  - comparaisons : `<`, `>`, `<=`, `>=`, `==`, `!=`
- Déclaration et initialisation : `t v = x;`
- Transtypage : `t v = (t) x;`
- Pointeurs :
  - pointeur d'un objet de type `t` : `t*`
  - déréférencement (pointeur → valeur) : `*ptr`
  - référencement (valeur → pointeur) : `&v`
  - allocation sur le tas : `malloc(`*nb octets*`)` renvoie un pointeur vers la zone
  - taille d'un type ou objet : `sizeof(t)`, `sizeof(v)`
  - libération d'une zone mémoire : `free(ptr)`
- Tableaux :
  - initialisation sur la pile : `t tab[N] = {0};` ou `t tab[N] = {x0, x1, ...};`, `N` doit être une constante entière littérale
  - allocation sur le tas : `t* tab = malloc(n * sizeof(t));`
  - élément d'indice i : `tab[i]`
  - chaînes de caractères : type `char*`, valeurs `"hello"`, `"world"`, ..., ce sont des tableaux de `char` dont le dernier élément est `'\0'`
- Structures de contrôle : délimitées avec `{...}`, les variables déclarées sont locales
  - test : `if (b) {...} else {...}`
  - boucle : `for (`*init*`;`*fin*`;`*incr*`){...}` ou `while (b) {...}`
  - sortie de boucle : `break`
  - passage à l'itération suivante : `continue`
  - fonction : `t f(t1 v1, ...){...}`, `int main(void)`, ou avec des arguments `int main(int argc, char* argv[])`
- Types structurés :
  - déclaration : `struct T {t1 champ1; ...};`
  - renommage : `typedef struct T t;`
  - initialisation sur la pile : `t v = {.champ1 = x1, ...};`
  - accès à un champ : `v.champ1` ou `ptr->champ1`
- Affichage et lecture :
  - affichage d'une chaîne formatée : `printf("chaîne", v1, v2, ...)`
  - lecture d'une entrée : `scanf("%xxx", ptr)`
  - formattage : `%d` (entier), `%lf` (double), `%s` (chaîne de caractères), `%p` (pointeur)
  - caractères spéciaux : `\n` (retour à la ligne), `\t` (tabulation)
- Compilation et exécution :
  - syntaxe générale : `gcc` *options* `source.c`
  - options :
    - nom d'exécutable : `-o nom_exec`
    - avertissements : `-Wall`, `-Wextra`
    - alerte mémoire : `-fsanitize=address`
  - exécution : `./nom_exec`

15 Programmation en OCaml

`t` désigne un type générique, `v` une variable de type `t`, `tab` un tableau d'éléments de type `t`, `lst` une liste d'éléments de type `t`, `x` une expression de type `t`, `n` un entier, `b` un booléen, `f` une fonction, `e` une expression, `E` une exception.

- Commentaires : entre `(*...*)`
- Importation de bibliothèque : `#load "biblio"`
- Ouverture de module : `open Module`
- Types de base :
  - *void* : `unit`, valeur `()`
  - entiers : `int`
  - flottants : `float`
  - booléens : `bool`, valeurs `true` et `false`
  - caractères : `char`, valeurs `'a'`, `'b'`, ...
  - chaînes de caractères : `string`, valeurs `"hello"`, ...
- Opérations :
  - entiers : `+`, `-`, `*`, `/`
  - flottants : `+.`, `-.`, `*.`, `/.`, `**`
  - modulo : `mod`
  - opérateurs booléens : `&&`, `||`, `not`
  - comparaisons : `<`, `>`, `<=`, `>=`, `=`, `<>`
- Déclaration et initialisation : `let v = x` (`v` globale), `let v = x in e` (`v` locale à `e`)
- Tableaux :
  - type : `t array`
  - création : `Array.make n x`, `[|x0; x1; ...|]`
  - taille : `Array.length tab` (en $\mathcal{O}(1)$)
  - copie superficielle : `Array.copy tab`
  - initialisation : `Array.init n f` calcule `[|f 0; f 1; ...|]`
  - élément d'indice i : `tab.(i)`
  - modification : `tab.(i) <- x`
  - parcours : `Array.iter f tab` exécute `f tab.(0); f tab.(1); ...`
  - image par une fonction : `Array.map f tab` renvoie `[|f tab.(0); f tab.(1); ...|]`
- Listes :
  - type : `t list`
  - création : `[]`, `x :: lst`, `[x0; x1; ...]`
  - taille : `List.length lst` (en $\mathcal{O}(n)$)
  - tête et queue : `List.hd lst`, `List.tl lst`
  - parcours et image `List.iter` et `List.map`
- Uplets :
  - type : `t1 * t2 * ...`
  - création : `(x1, x2, ...)`
  - accès à la ième composante : `let (x1, x2, ...) = v in xi`
  - couples uniquement : `fst v`, `snd v`
- Types enregistrements :
  - définition (avec champs mutables) : `type t = {champ1 : t1; mutable champ2 : t2; ...}`
  - création : `{champ1 = x1; ...}`
  - accès à un champ : `v.champ1`
  - modification : `v.champ1 <- x`
- Références :
  - création : `let r = ref x`
  - accès à la valeur : `!r`
  - modification : `r := x`
- Types construits :
  - définition : `type t = Cons1 | Cons2 of t2 | Cons3 of t3 * t4 | ...`
  - création : `Cons1`, `Cons2 x`, `Cons3 (x3, x4)`
  - type option (existe par défaut) : défini par `type 'a option = None | Some of 'a`
- Structures de contrôle :
  - test : `if b then (...) else begin...end`, les `( )` et `begin end` sont interchangeables
  - filtrage : `match v with | motif1 -> ... | motif2 -> ...` un motif peut contenir types construits, enregistrements, variables distinctes ou constantes
  - boucle Pour : `for i = n1 to n2 do ... done`, `n1` et `n2` sont atteintes, on remplace `do` par `downto` pour une boucle décroissante
  - boucle Tant que : `while b do ... done`
  - fonction : `let f v1 v2 ... = ...`
  - fonction récursive : `let rec f v = ...`
- Exceptions :
  - définition : `exception E1`, `exception E2 of t`
  - levée : `raise E1`, `raise E2(v)`
  - rattrapage : `try ... with | E1 -> ... | E2(v) -> ...`
  - message d'erreur : `failwith "erreur"`
- Affichage et lecture :
  - types de base : `print_int`, `print_float`, `print_string`
  - affichage de chaîne formatée : `Printf.printf`
  - lecture : `read_int`, `read_float`, `read_line`
- Compilation et interprétation :
  - dans un interpréteur : `#use "source.ml";;`
  - dans une console : `ocamlc source.ml` (byte-code) ou `ocamlopt source.ml` (natif)