TD d'Informatique MP2I

Julien Reichert

2024/2025

Ce document contient mes TD d'informatique pour l'année scolaire 2024/2025 en MP2I.

L'association entre les TD et un chapitre de cours, quasiment systématique, est précisée dans le TD et en cours.

Plus on s'avancera dans l'année, plus le travail sera en principe faisable sans utiliser d'ordinateur, il s'agira souvent de théorie, d'études d'algorithmes ou d'analyse de programmes déjà écrits. Pour autant, rien n'exclut de s'aider d'un environnement de développement intégré de l'éventuel langage concerné, si la consigne ne l'interdit pas explicitement.

Au vu de la suppression des TD-preuves à partir de 2024/2025, certains TD longs seront traités sur deux séances.

Table des matières

TD 0 : Algorithmique et programmation

En guise de (re)découverte de l'algorithmique, le TD ici proposé consiste à ressortir de l'oubli [1] un langage informatique, Logo [2], et d'en utiliser les principes pour écrire en pseudo-code des instructions basiques pour faire des dessins en déplaçant une tortue.

Pour les éventuels intéressés, il est possible d'utiliser un interpréteur en ligne du langage Logo à l'adresse `http://www.calormen.com/jslogo/`, une aide à la syntaxe étant fournie.

Comme signalé au premier paragraphe, dans notre langage, appelé sommairement « Logo-- », nous réalisons des figures en déplaçant une tortue dans un repère orthonormé du plan à l'aide de différentes instructions.

Contrairement au cas de la plupart des librairies graphiques des langages de programmation habituels, on s'interdira ici de forcer une nouvelle position par « téléport ». Tous les déplacements se feront en fonction de l'orientation de la tortue, à l'aide des instructions **avancer** (`A`), **reculer** (`R`), **tourner à gauche** (`G`) et **tourner à droite** (`D`). Tourner se fait sur place et à angle droit. Par commodité, on autorise la syntaxe `A(n)`, où `n` est un entier, qui représente le fait d'avancer de `n` unités de longueur. De même pour `R(n)`.

On constate une redondance entre les instructions `A(n)` et `R(n)`, de même qu'entre les instructions `G` et `D`. La plupart des langages disposent de ce qu'on appelle de manière informelle « sucre syntaxique », des instructions qui ne sont pas indispensables mais qui facilitent la vie du programmeur et n'ont pas à être redéfinies par lui-même avant d'écrire un programme.

À l'aide des instructions de déplacement, la tortue peut aller de n'importe quelle configuration (la donnée de l'abscisse, de l'ordonnée et de l'orientation) à n'importe quelle autre. La tortue disposant d'un crayon, son déplacement est alors tracé.

Sans restriction, les seules figures qui peuvent être obtenues sont celles qu'on peut représenter sans lever le crayon. Puisqu'on a dit qu'il était exclu de téléporter la tortue (ce qui est cependant possible en Logo), il faut trouver une solution.

1. tout est relatif, il existe tout de même un module en python qui lui rend hommage
2. `http://fr.wikipedia.org/wiki/Logo_(langage)`

Parler de saut en serait une, mais pour se conformer à l'esprit de Logo on préfèrera ajouter les instructions **lever le crayon** (`L`) et **baisser le crayon** (`B`). Au début, le crayon est baissé. On considère qu'il n'y a pas de souci si on lève un crayon levé ou si on baisse un crayon baissé.

Afin de disposer d'une puissance de calcul suffisante, on ajoute aux instructions les constructions de base de la programmation :

- `Si <condition> alors <instructions> sinon <instructions> Fin Si`. *La partie « sinon » n'est pas obligatoire.*
- `Pour <nom de variable> de <seuil> à <seuil> [par pas de <pas>] faire <instructions> Fin Pour`. *La variable choisie est augmentée d'un par défaut, ou du pas précisé (pouvant être négatif) à chaque passage dans la boucle, pour progresser du premier seuil au second.*
- `Tant que <condition> faire <instructions> Fin Tant que`.

Le balisage de fin n'est pas nécessaire dans les langages, ou les conventions d'écriture de pseudo-code, pour lesquels il existe un moyen de prévenir les ambigüités.

Exercice 1 : Réaliser un programme qui permet de tracer six lignes horizontales de `(0, i)` à `(5, i)` pour `i` entre `-5` et `0`, avec pour configuration de départ `(0, 0)` et une orientation vers la droite. Comparer avec les programmes d'autres étudiants.

Exercice 2 : Réaliser un programme qui permet de tracer une spirale partant de `(0, 0)` construite de la manière suivante : un pas à droite, un pas en haut, deux pas à gauche, deux pas en bas, trois pas à droite, etc. jusqu'à avoir fait douze étapes. Essayer de trouver une façon de commencer par `Pour i de 1 à 12`.

Exercice 3 : Réaliser un programme qui envoie la tortue en `(0, 0)` avec l'orientation vers le haut, quelle que soit sa position de départ. Il peut être pratique de commencer par forcer une orientation.

Pour ce faire, deux fonctions de repérage de la tortue sont offertes :

- `pos()` renvoie le couple de coordonnées de la tortue. On peut s'en servir en écrivant `(x,y) <- pos()`, ce qui a pour effet de mettre dans `x` l'abscisse de la tortue et dans `y` l'ordonnée de la tortue au moment de l'appel de la fonction ;
- `orientation()` renvoie l'orientation de la tortue, parmi les points cardinaux, encodés en tant que `"N"`, `"S"`, `"E"` et `"O"`.

Exercice 4 : Trouver le moyen d'écrire la fonction `orientation()` à l'aide de `pos()`. On prendra garde à ce que la tortue soit dans la même position et la même orientation au début et à la fin de l'exécution (sinon, il se produit ce qu'on appelle un effet de bord).

La syntaxe proposée pour l'écriture de fonctions est la suivante :

`Fonction <nom de fonction> <instructions> Fin Fonction`

... et pour les valeurs de retour, on utilisera le mot-clé `Retourner`.

Une fonction supplémentaire sera nécessaire pour que le travail soit complet, on l'appellera `etat_crayon()` et elle renverra **un booléen** indiquant si le crayon est baissé.

En guise d'introduction à la récursivité, utilisée dans le premier langage de programmation enseigné par la suite, les fonctions définies ici pourront s'appeler elles-mêmes.

Exercice 5 : Refaire les deux premiers exercices en écrivant une fonction faisant intervenir de la récursivité prenant en argument le nombre de lignes à tracer dans les deux cas. Constater qu'il y a une difficulté particulière dans le deuxième exercice.

Pour aller plus loin, on peut faire de magnifiques figures à l'aide de fonctions récursives, comme les fractales.

Exercice 6 : Après s'être renseigné sur le flocon de Von Koch, écrire une fonction récursive permettant de la faire tracer en fonction de la « profondeur » de récursion et de la taille de la ligne.

Pour ce dernier exercice, il faut étendre les instructions élémentaires de déplacement : avancer peut se faire sur une longueur non nécessairement entière, et les fonctions pour tourner à gauche et à droite sont paramétrées par un angle en degrés.

TD 1 : Limites de la pratique par rapport à la théorie

Dans ce TD, le travail théorique sur feuille précèdera la vérification pratique, et ceci exercice par exercice.

L'ordinateur ne sait pas s'arrêter !

Exercice 1 : Soit la fonction de Morris ci-après. Prouver qu'elle est censée toujours retourner `1`. Constater qu'elle ne termine pas quand on appelle `morris(1, 0)` et chercher à expliquer pourquoi.

```
Fonction morris(m, n)
    Si m est nul alors Retourner 1 Fin Si
    Retourner morris(m-1, morris(m, n))
Fin Fonction
```

Exercice 2 : Soit la fonction d'Ackermann ci-après. Prouver qu'elle termine. Déterminer par le calcul (avec une récurrence) la valeur retournée en fonction de `n` quand `m` vaut successivement les entiers de `0` à `3`. Deviner ce qui se passe au-delà et tester.

```
Fonction ackermann(m, n)
    Si m est nul alors Retourner n+1 Fin Si
    Si n est nul alors Retourner ackermann(m-1, 1) Fin Si
    Retourner ackermann(m-1, ackermann(m, n-1))
Fin Fonction
```

L'ordinateur ne sait pas calculer !

Exercice 3 : Trouver trois valeurs `x`, `y` et `z` telles que le résultat de la commande `x + (y + z)` soit différent de celui de `(x + y) + z`. Trouver aussi deux valeurs `x` et `y` telles que le résultat de la commande (Python spécifiquement) `int(x/y)` soit différent de celui de `x // y`.

Exercice 4 : Consulter la documentation de la fonction `round` de Python. Noter la convention surprenante pour l'arrondi à l'unité des nombres entiers plus un demi. Demander l'arrondi au dixième près des nombres `1.05`, `10.05`, `100.05` et `1000.05`.

Exercice 5 : Anticiper le résultat de $1 - \frac{1}{3} - \frac{1}{3} - \frac{1}{3}$ et de $1 - \frac{1}{5} - \frac{1}{5} - \frac{1}{5} - \frac{1}{5} - \frac{1}{5}$. Cela ne marcherait pas avec $1 - 5 * \frac{1}{5}$ d'ailleurs ! Qu'en serait-il de $1 - \frac{1}{4} - \frac{1}{4} - \frac{1}{4} - \frac{1}{4}$?

Exercice 6 : Écrire une fonction qui retourne le discriminant d'un polynôme du second degré à partir des trois coefficients `a`, `b` et `c`. Écrire ensuite une fonction qui retourne l'ensemble des solutions réelles d'une équation polynomiale définie par les trois mêmes coefficients. Tester la fonction pour les fonctions $x^2 + 1.4x + 0.49$, $x^2 + 0.2x + 0.01$ et $x^2 + x + \frac{1}{4} + 10^{-20}$. Commenter.

L'intérêt de ne pas faire de tests d'égalité sur des réels se dessine. Malheureusement, au vu du troisième exemple, on ne peut pas non plus se permettre d'assimiler à zéro une valeur certes proche mais différente. Les logiciels de calcul formel tentent de pallier ce genre de problèmes que l'on rencontre lorsque l'on manipule des réels comme ici sans outils adaptés. D'ailleurs, demander le sinus de π donnera un argument supplémentaire pour se méfier.

Exercice 7 : Exécuter les deux lignes suivantes en Python et les expliquer.

```
n = 2**61 - 2
n < n - 1.0
```

L'ordinateur ne sait pas. . . qu'il ne doit pas s'arrêter !

Exercice 8 : Prouver que le code ci-dessous ne doit pas terminer. L'exécuter dans n'importe quel langage. Constater qu'il termine tout de même. Compter le nombre de tours de boucle utilisés en modifiant le code.

```
Fixer x à 1.0
Tant que x est non nul faire
    Réduire x de moitié
Fin Tant que
```

Exercice 9 : Même exercice mais en écrivant un code où la valeur de départ de `x` est `2.0` et on fait tendre `x` vers `1.0` en comptant de nouveau le nombre de tours de boucle.

Exercice 10 : Encore la même chose avec `101.0` et `100.0` avec anticipation du nombre de tours de boucle.

Bien entendu, les problèmes soulevés ici ne dépendent pas du langage de programmation a priori tant que les nombres ont une gestion similaire. OCaml et C ne mélangeant pas les entiers et les flottants, il a fallu faire intervenir un autre langage pour certains tests.

TD 2 : Terminaison, correction, complexité

Pour commencer, une étape douloureuse mais nécessaire : écrire et prouver des algorithmes dichotomiques. On en fera deux : la recherche d'un élément dans un tableau croissant d'entiers (en C), et l'approximation d'un zéro d'une fonction continue à rechercher dans un intervalle entre deux points dont les images par la fonction sont de signe opposé (en Python).

Ensuite, pour les fonctions successives déjà écrites, prouver la terminaison et calculer la complexité en fonction des arguments, déterminer la signature et la spécification précise puis prouver la correction.

```
def FisherYates(l):
    n = len(l)
    ll = deepcopy(l)
    for i in range(n-1, 0, -1):
        j = randint(0, i)
        ll[j], ll[i] = ll[i], ll[j]
    return ll

# -----------------------------------

def r(s):
    rep = ""
    for i in range(len(s)):
        car = s[i]
        rep = car + rep
    return rep

# -----------------------------------

def c(s, m):
    rep = 0
    for i in range(len(s)):
        if s[i:i+len(m)] == m:
            rep += 1
    return rep
```

```
def verif1(l, ll):
    for x in l:
        if x not in ll:
            return False
    for x in ll:
        if x not in l:
            return False
    return True

# -----------------------------------

bool verif2(int* t, int n)
{
  int i;
  for (i = 0 ; i < n-1 && t[i] <= t[i+1] ; i += 1);
  return i == n-1;
}

# -----------------------------------

bool verif3(int* t, int n)
{
  for (int i = 2 ; i < n ; i += 1)
  {
    if (t[i] != t[0] + i*(t[1]-t[0])) return false;
  }
  return true;
}
```

TD 3 : Récursivité

Dans ce TD, des exercices d'algorithmique sur la récursivité sont proposés. Il s'agira de trouver les formules reliant chaque instance d'un problème et d'autres instances progressant vers des cas de base, après avoir identifié ceux-ci.

Une fois ce travail terminé pour tous les problèmes abordés, une implémentation en OCaml sera effectuée, idéalement sans l'aide de l'ordinateur dans un premier temps.

Un but annexe de ce TD est de travailler sur la complexité.

Arithmétique

Commençons cette section en calculant $x \times y$ et a^n en faisant une récursion sur y et sur n respectivement. Tous les nombres seront supposés entiers et positifs.

Exercice 1 : Commencer par donner la formule naïve pour les deux opérations. Ensuite, chercher une formule faisant intervenir une forme de dichotomie. Comparer les deux formules et commenter.

Un calcul plus avancé, et nécessitant éventuellement une introduction ou un rappel en fonction de ce qui a été vu en pré-bac, fait intervenir le PGCD de deux entiers, obtenu par l'algorithme d'Euclide. On étend cet algorithme pour déterminer un couple de Bézout. Les deux exercices suivants ont été par le passé la hantise de nombreux étudiants, pas forcément de manière justifiée.

Exercice 2 : Écrire une formule récursive donnant la valeur du PGCD de deux entiers naturels non nuls en fonction d'un autre PGCD progressant vers un cas de base à déterminer.

Exercice 3 : Faire tourner à la main le calcul d'un couple de Bézout pour les entiers 19 et 12. Généraliser à deux entiers quelconques.

Un peu d'ASCIIArt

Les figures à faire dessiner dans cette section seront composées de répétitions du symbole étoile pour le côté esthétique.

Pour le plaisir, on pourra inventer des exercices supplémentaires, éventuellement en faisant intervenir plusieurs symboles différents.

Exercice 4 : Écrire une formule récursive pour dessiner un « escalier » sur `n` lignes en fonction de `n`. Le rendu pour `n` valant `5` figure ci-après.

```
*
**
***
****
*****
```

Exercice 5 : Écrire une formule récursive pour dessiner un « triangle » sur `n` lignes en fonction de `n`. Le rendu pour `n` valant `3` figure ci-après. Après les étoiles, la ligne peut s'arrêter ou être complétée par des espaces pour avoir toujours le même nombre de caractères par ligne, au choix. Attention, ici la récursion est plus subtile, car la valeur initiale de `n` intervient à tout moment !

```
  *
 ***
*****
```

Exercice 6 : Écrire une formule récursive pour dessiner un « diamant » sur `n` lignes en fonction de `n`, forcément impair. Le rendu pour `n` valant `5` figure ci-après. Après les étoiles, la ligne peut s'arrêter ou être complétée par des espaces pour avoir toujours le même nombre de caractères par ligne, au choix. Ici c'est plus intuitif de remarquer que la valeur initiale de `n` intervient dans les appels récursifs.

```
  *
 ***
*****
 ***
  *
```

Exercice 7 : Écrire une formule récursive pour dessiner la forme suivante sur `n` lignes en fonction de `n`, forcément impair.

```
******
*****
****
***
**
*
**
***
****
*****
******
```

Permutations, sous-ensembles, etc.

La construction de l'ensemble des sous-ensembles à `k` éléments d'un ensemble à `n` éléments donne un aperçu de la preuve combinatoire de la formule des coefficients binomiaux. Il en va de même pour les autres objets de cette section.

Exercice 8 : Déterminer comment obtenir récursivement l'ensemble des listes de `p` éléments d'un ensemble à `n` éléments. Les répétitions sont ici possibles.

Exercice 9 : Déterminer comment obtenir récursivement l'ensemble des arrangements de `p` éléments d'un ensemble à `n` éléments. Les répétitions sont ici impossibles, en particulier on suppose que `p` est entre `0` et `n`.

Exercice 10 : En déduire comment obtenir récursivement l'ensemble des permutations d'un ensemble à `n` éléments.

Exercice 11 : Déterminer comment obtenir récursivement l'ensemble des sous-ensembles de `p` éléments d'un ensemble à `n` éléments. Les répétitions restent impossibles, et ici l'ordre n'importe plus.

Les tours de Hanoï

Le problème des tours de Hanoï, que nous devons à Édouard Lucas (fin du XIX^{e} siècle), consiste à déplacer une pile de `n` (en argument) anneaux de taille croissante d'un tas (matérialisé par un piquet) à un autre (parmi trois), les opérations élémentaires étant le déplacement d'un anneau du haut d'une pile sur le haut d'une autre pile, à condition qu'il soit plus petit que l'ancien sommet de la pile d'arrivée.

Exercice 12 : Déterminer une méthode de résolution récursive de ce problème.

Exercice 13 : Déterminer une méthode récursive pour passer d'une configuration quelconque (par exemple donnée sous la forme de trois listes croissantes) à une configuration finale possible (une liste croissante, quelle qu'elle soit, et deux listes vides).

Le retour de la tortue (à faire après la séance)

Pour cette section, on utilisera au choix une console Python avec le module `turtle` ou un interpréteur Logo (rappel : `http://www.calormen.com/jslogo/`), le travail se faisant sur un ordinateur pour profiter du joli rendu graphique.

Exercice 14 : Écrire une fonction récursive imprimant un flocon de von Koch après un nombre d'étapes donné en argument.

Des explications sur le flocon pourront être données avant la fin de la séance. À défaut, une recherche rapide sur internet fera l'affaire.

Exercice 15 : Chercher sur internet d'autres courbes fractales et les représenter à l'aide de fonctions récursives.

TD 4 : Les modules `List` et `Array` en OCaml

Les listes

Le module `List`, issu de la bibliothèque principale (*core library*), contient des fonctions déjà présentées en cours, à savoir `hd`, `tl` et `length`, ainsi que l'opérateur `@`.

D'autres fonctions de ce module font l'objet de ce TD. **Elles sont à maîtriser si leur spécification est rappelée dans un énoncé** :

- `rev` (pas au programme) renvoie la version retournée de la liste ;
- `for_all` et `exists` vérifient si tous les éléments (resp. au moins un élément) de la liste en deuxième argument satisfont (resp. satisfait) un prédicat, c'est-à-dire une fonction prenant en argument des objets du type commun des éléments de la liste, en premier argument, et retournant un booléen. La valeur `x` satisfait le prédicat `p` si, et seulement si, `p x` est vrai ;
- `mem` vérifie si le premier argument est un élément de la liste en deuxième argument.

Exercice 1 : Réécrire toutes ces fonctions (en tant que fonctions récursives).

Une importance particulière est accordée aux itérations de fonctions [3] :

- `map : ('a -> 'b) -> 'a list -> 'b list` applique son premier argument aux éléments de son deuxième argument dans l'ordre pour former une liste ;
- `iter : ('a -> unit) -> 'a list -> unit` fait la même chose, mais le premier argument est une fonction renvoyant un `unit` ;
- `filter : ('a -> bool) -> 'a list -> 'a list` renvoie la liste, dans le même ordre d'apparition, des éléments de la liste en deuxième argument qui satisfont le prédicat en premier argument.
- `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` applique son premier argument de manière imbriquée à tous les éléments de son deuxième argument conjointement à son troisième argument [4] ;
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` est similaire, mais l'ordre est inversé [5].

3. Malgré le nom, on peut tout de même faire ce genre de fonctions de manière récursive.
4. En clair, `fold_right f [a1; ...; an] b` correspond à `f a1 (f a2 (...(f an b) ...))`.
5. De même, `fold_left f b [a1; ...; an]` correspond à `f (...(f (f b a1) a2) ...) an`.

Exercice 2 : Réécrire également ces cinq fonctions.

Au passage, seules les fonctions `fold_left` et `fold_right` ne sont pas explicitement au programme. Mais elles sont tellement classes...

Exercice 3 : Écrire à l'aide d'itérateurs de listes des fonctions `print_type_list` pour différents types.

Exercice 4 : Écrire les fonctions `for_all`, `exists` et `mem` à l'aide de `fold_left`.

Exercice 5 : Écrire la factorielle à l'aide de `fold_left` ou `iter` en créant la liste sur laquelle faire une itération [6].

Les tableaux

Le module `Array` est le pendant du module `List` pour les tableaux. Il contient également des fonctions à maîtriser, bien qu'elles ne soient pas à connaître par cœur. On tâchera cependant de se souvenir de `length` et `make`, ainsi que `make_matrix` dans une moindre mesure, déjà documentées.

- `init : int -> (int -> 'a) -> 'a array` crée un tableau de taille son premier argument, dont les éléments sont donnés par l'application de la fonction en deuxième argument aux indices du tableau. [7]
- `copy : 'a array -> 'a array` crée un tableau dont les éléments sont les mêmes que les éléments du tableau en argument. En particulier, si les éléments du tableau sont mutables, ceux de la copie seront dépendants. Attention donc à ne pas copier de matrice sans précaution.
- `mem : 'a -> 'a array -> bool` est similaire à `List.mem`.
- `for_all` et `exists` : même remarque.
- `map` : toujours la même remarque.
- `iter` : on aura compris...

Exercice 6 : Adapter les exercices précédents aux tableaux.

6. conseil : ne jamais réécrire la factorielle ainsi
7. Une version récente a introduit la fonction `List.init`, d'effet analogue, mais toujours pas de `List.make` à l'horizon.

TD 5 : Tris

Le but de ce TD est de découvrir ou redécouvrir des algorithmes de tris classiques. Le programme officiel de MP2I ne prévoit pas de section dédiée aux tris, ces algorithmes étant alors plutôt prévus pour servir d'illustrations de paradigmes de programmation ou comme exercices quelconques en TP.

C'est un peu l'esprit du programme de tronc commun qui suggère de traiter les tris comme une notion à apprendre au premier semestre, tout en mettant le tri à bulles comme exemple à l'occasion de l'introduction des doubles boucles.

Bien que ce TD soit essentiellement algorithmique, une mise en œuvre est aussi attendue, en s'efforçant d'alterner entre C et OCaml.

Au vu de l'avancement dans l'année, on choisira de préférence la structure de liste quand elle pertinente, pour une programmation récursive en OCaml, et une structure de tableau sinon, pour une programmation impérative en C.

La proposition de l'énoncé n'est certes pas une obligation, mais la correction mise en ligne la respectera et se limitera à ceci.

Introduction

La notion de dichotomie est désormais connue, en attendant la théorie qui la généralise en un paradigme de programmation.

Nous avons abordé, au moment de donner les formules de complexité, que la dichotomie apportait un avantage considérable en termes de temps de calcul, mais elle nécessite des conditions particulières sur les données.

Ainsi, une structure est plus pratique si elle est triée. Nous allons désormais étudier divers algorithmes de tri dans cette optique.

Les tris de ce TD seront toujours dans l'ordre croissant, en gardant à l'esprit qu'il est immédiat de passer à la version décroissante, voire d'ajouter un booléen en argument précisant dans quel ordre le tri doit se faire.

Comparaisons

La plupart des tris classiques agissent par comparaison, c'est-à-dire que l'on regardera deux à deux certains couples d'éléments, et on prendra une décision en fonction de l'élément qui est le plus grand des deux.

En quelque sorte, parmi les $n!$ permutations d'une collection de n éléments distincts, une comparaison permet d'éliminer certaines (si possible de l'ordre de la moitié du total) : les permutations qui ne donnent pas le même résultat pour la comparaison en question.

Le meilleur tri par comparaison fait donc implicitement une dichotomie parmi l'ensemble des permutations candidates restantes.

Les tris par comparaison les plus simples seront en temps quadratique, en comptant comme unité de temps la comparaison (ou l'affectation, intuitivement plus coûteuse). C'est moins bien que le tri rapide et que le tri fusion, mais ces derniers ne sont pas aussi aisés à comprendre.

Il existe également des tris n'agissant pas par comparaison. Ils peuvent être plus simples, voire plus efficaces, mais nécessitent souvent des conditions particulières. Nous verrons dans ce TD le plus classique d'entre eux : le tri par dénombrement.

Tris en place

Un tri est dit en place si toute l'exécution de l'algorithme utilise une mémoire constante, matérialisée par exemple par les éléments sauvegardés dans l'optique d'un échange, les variables représentant les indices, etc.

En particulier, un tri en place agit par effet de bord car il ne peut pas se permettre de construire la structure à retourner.

Attention : un tri qui construit la structure à retourner puis remplace dans la mémoire la structure en entrée par celle qui a été construite aura certes agi par effet de bord, mais ne sera pas en place.

On peut écrire la plupart des tris en place ou non en place, sachant que la version en place n'est pas toujours aussi facile à écrire (surtout pour le tri rapide), voire à proscrire (cas du tri fusion).

Parfois c'est étonnamment l'inverse qui se passe (le tri par sélection est plus agréable en place, et des bugs menacent si on ne prend pas assez garde dans la version non en place).

Toujours est-il qu'un tri en place sera plus rapide en raison des allocations de mémoire qu'on économise.

Stabilité

Un tri est dit stable si deux éléments égaux au regard de la fonction de comparaison se retrouvent dans le même ordre dans la version initiale et dans la version triée de la structure.

Cette propriété n'est pas fondamentale si la relation de comparaison ne peut pas considérer comme égaux des éléments qui peuvent être distingués, mais peut présenter un intérêt si le tri n'étudie par exemple qu'un élément de chaque n-uplet d'une structure.

On se posera la question de la stabilité de tous les tris écrits au cours de la séance.

Exercices

Exercice 1 : Écrire une version non en place des tris par insertion et par sélection en OCaml, et une version en place de ces mêmes tris en C.

Le *tri par insertion* consiste à prendre chaque élément de la structure et de le placer au bon endroit parmi les éléments déjà considérés.

De manière duale, le *tri par sélection* consiste à prendre le plus petit des éléments non encore considérés et le mettre après le dernier élément placé.

Exercice 2 : Implémenter le tri cocktail, expliqué ci-après, en C.

Le classique mais très mauvais tri à bulles, qui consiste à échanger dans une double boucle deux éléments voisins s'ils sont mal ordonnés, pèche par sa complexité, et seules sa compréhension relativement facile et son originalité font qu'il est enseigné. Nous allons ici broder autour.

Vérifier si un tableau est déjà trié peut se faire de manière très pratique dans le tri à bulles : on utilise un booléen déterminant si un échange a été effectué dans un parcours de la boucle principale. Si le booléen reste à `False`, c'est que le tableau est déjà trié.

L'utilité est de ne pas perdre de temps quand le tableau de départ est presque trié, dans la mesure où par exemple quelques éléments parmi les plus grands sont mis au début du tableau.

Le principe du tri à bulles fait que de tels éléments seront vite envoyés à leur place, mais le souci est qu'au contraire des éléments parmi les plus petits sont peut-être à la fin du tableau, et ils devront être échangés au cours d'un nombre considérable de parcours de la boucle principale.

Le tri cocktail pallie ce souci en faisant des parcours du tableau dans les deux sens. Après chaque (double) parcours dans le tri cocktail, un élément de plus est à la bonne place aux deux extrémités.

Il est recommandé après la séance de comparer l'efficacité de tous les tris écrits ici sur une structure obtenue en prenant la liste des nombres de `1` à `100000` dans l'ordre croissant et en faisant une demi-douzaine d'échanges. Normalement, le tri cocktail avec vérification après chaque passage que la liste est bien triée devrait gagner.

Exercice 3 : Implémenter le tri fusion, expliqué ci-après, en OCaml.

Le *tri fusion* s'adapte à merveille à la récursivité : on prend la moitié gauche et la moitié droite de la liste, on construit la version triée par le tri fusion, puis on les fusionne en prenant à chaque fois l'élément minimum parmi les deux éléments de tête. La fusion proprement dite transforme alors effectivement deux listes triées en leur union triée.

Exercice 4 : Implémenter le tri par dénombrement, expliqué ci-après, en C. On supposera que le tableau en entrée contiendra des valeurs entre `0` et `k`, pour un `k` en argument.

Le **tri par dénombrement** ou **tri comptage** (*counting sort*) permet de trier une structure dont les éléments proviennent d'un ensemble fini, et si possible de taille très petite devant la taille de la structure.

Par exemple, on voudrait trier un million de caractères de la table ASCII, ce qui s'assimile à trier un million de nombres entre 0 et 127.

Pour ce faire, on crée un tableau de taille 128 dont les éléments sont le nombre d'occurrences de chacun des nombres de 0 à 127, que l'on va compter en parcourant la structure. Une fois le parcours terminé, on constitue directement la version triée de la structure. Attention, même si on trie un tableau en le mutant, l'utilisation des compteurs fait que l'on ne peut pas appeler ceci un travail en place.

TD 6 : Encore des tris et borne inférieure de complexité

Pour alléger le TD précédent, deux algorithmes de tris seront présentés en introduction de ce TD, dont le but principal est de prouver le théorème suivant :

Théorème

Considérons un algorithme de tri fonctionnant par comparaisons sur des listes quelconques. Le nombre de comparaisons effectuées par cet algorithme dans le pire des cas ne peut pas être négligeable devant $n \log_2(n)$, où n est la taille de la liste.

Tris supplémentaires

Exercice 1 : Implémenter le tri rapide, expliqué ci-après, en C dans sa version en place (la version non en place se fera dans le TD suivant).

Comme le tri fusion vu au TD précédent, le *tri rapide* est classiquement récursif et applique le principe de « diviser pour régner » qui sera vu au deuxième semestre.

Ici, les comparaisons se font au moment de créer les deux structures (de taille non contrôlée cette fois) sur lesquelles les appels récursifs doivent être exécutés, ce qui se fait autour d'un pivot qui sera un élément arbitraire (le premier, disons, même si ce choix peut occasionner des temps très longs quand la structure est déjà croissante ou décroissante).

Attention, pour agir en place, il faut échanger les éléments de la zone bien délimitée en cours d'étude pour les envoyer du côté droit s'ils sont supérieurs au pivot ou du côté gauche s'ils lui sont inférieurs.

Exercice 2 : Implémenter le tri par base pour des listes d'entiers positifs, expliqué ci-après, en OCaml.

Le **tri par base** ou **tri radix**, voire **tri par baquets** (*radix sort*), est un tri coûteux en espace mais dont la complexité en temps est un $\mathcal{O}(kn)$, où k est la taille (en tant que nombres ou chaîne de caractères) des éléments de la liste.

Ceci correspond à un $\mathcal{O}(n \log n)$ pour les nombres de 1 à n, mais on peut imaginer des cas où le nombre k est plus intéressant. Historiquement, ce tri a été introduit pour classer des cartes perforées.

Pour illustrer le principe, on considère une liste de n nombres entre 0 et 999. Le tri par base fait trois passages dans une boucle principale, consistant à insérer de gauche à droite les nombres dans 10 listes, une par chiffre des unités au premier passage, dizaines au deuxième, centaines au troisième, avant de fusionner à chaque fois les listes dans l'ordre de leur identifiant.

À la fin du deuxième passage dans la boucle principale, les nombres sont dans l'ordre croissant des dizaines, et à même dizaine, ils sont dans l'ordre croissant des unités en raison du résultat du premier passage, ce qui permet de déduire la correction de l'algorithme.

Il est important de noter que chaque passage correspond à un tri stable.

Exercice 3 : Implémenter le tri par base pour des listes de chaînes de caractères en OCaml, dont on supposera qu'elles sont constituées uniquement de lettres non accentuées, et où le tri ne sera pas sensible à la casse.

À la lumière de ce qui précède, on réfléchira bien à l'ordre dans lequel le tri s'effectuera ainsi qu'à la gestion du cas où des chaînes sont trop courtes au regard de l'indice du caractère à étudier sur le passage du tri en question.

On notera que la restriction aux chaînes constituées de lettres n'est pas nécessaire pour effectuer le tri par base, on peut tout aussi bien faire 256 paquets que 26.

La preuve du théorème

Considérons une séquence de taille n, dont on suppose les éléments deux à deux différents pour faciliter la preuve à venir, sans perte de généralité pour autant. En fait, puisque nous considérons un tri dont les actions dépendent de comparaisons entre deux éléments de la séquence, il ne coûte rien de supposer même que les éléments sont les entiers de 0 à $n - 1$.

On rappelle que le nombre de permutations d'un ensemble à n éléments est $n!$ (exercice de mathématiques).

Exercice 4 : Écrire en C une fonction prenant en argument un tableau d'entiers et sa taille ainsi qu'une permutation (de même taille) donnée en tant que tableau dont l'élément à chaque indice `i` correspond à l'image de `i` par la fonction associée et renvoyant le résultat de la permutation sur le tableau en premier argument.

Exercice 5 : Prouver que parmi toutes les permutations possibles de notre séquence considérée, une seule produit sa version triée.

Il s'agit formellement de déterminer quelle est la permutation qui convient, mais en pratique l'appliquer suffit, et même son application peut se décomposer en l'application de permutations qui la composent (en général des transpositions, car beaucoup d'algorithmes de tris procèdent par des échanges).

On rappelle au besoin qu'une transposition est une permutation σ telle qu'il existe $i \neq j$ vérifiant que $\sigma(i) = j$, $\sigma(j) = i$ et pour tout $k \notin \{i, j\}$, $\sigma(k) = k$, les valeurs i, j et k étant prises dans l'intervalle des indices possibles de la permutation. On note alors $(i\ j)$ la transposition en question, en écrivant en général la plus petite valeur en premier.

Exercice 6 : Dans notre séquence de départ, on compare les éléments à deux indices différents, notés `i` et `j`, et on observe que le premier est strictement inférieur au second. Que peut-on en déduire concernant la permutation qui trie la liste ?

Exercice 7 : Selon quel principe va-t-on déterminer de manière efficace quelle est la bonne permutation, par élimination des mauvaises en utilisant les comparaisons ?

La formule de Stirling donne un équivalent en l'infini de la factorielle de n, à savoir $(\frac{n}{e})^n\sqrt{2\pi n}$.

Exercice 8 : Par des manipulations d'équivalences, donner un équivalent du logarithme en base 2 de $n!$.

Exercice 9 : Rassembler tout ce raisonnement pour prouver le théorème.

Exercice 10 : Donner par ailleurs une preuve élégante et astucieuse du fait que toute permutation se décompose (pas de manière unique) en une composée de transpositions de la forme $(i\ j)$ avec $j = i + 1$.

TD 7 : Médiane des médianes

Ce TD traite d'un point technique sur les tris, donc n'entre pas spécifiquement dans le cadre du programme. Par des questions intermédiaires, on pourra être amené à suivre le raisonnement qui mène à un résultat magnifique.

L'algorithme que nous allons exposer ici permet de calculer la médiane d'une structure séquentielle en temps linéaire.

On appelle le principe utilisé « la médiane des médianes ».

Son utilité principale, beaucoup plus importante que la recherche de la médiane en tant que telle, est que cette médiane peut servir de pivot dans le tri rapide, garantissant par son utilisation que la complexité dans le pire des cas est tout de même un $\Theta(n \log n)$ avec n la taille de la structure triée.

Commençons par des questions d'introduction et de position du problème sur le tri rapide.

Exercice 1 : Donner un cas où l'utilisation systématique du premier élément de la zone étudiée comme pivot donne une complexité quadratique.

Exercice 2 : Commenter l'idée de mélanger d'abord avant d'utiliser le premier élément.

Exercice 3 : Commenter l'idée de prendre la moyenne entre le maximum et le minimum comme pivot.

Désormais, attelons-nous au calcul de la médiane d'une structure, médiane notée `M`.

Exercice 4 : Rappeler deux algorithmes possibles et en donner la complexité.

La première étape de notre algorithme en temps linéaire est de séparer la structure en autant de sous-structures de taille cinq (valeur un peu arbitraire) qu'il le faudra (la dernière structure sera potentiellement de taille moindre).

Exercice 5 : Pour une structure de taille cinq, dans quelle mesure le calcul de la médiane peut-il être fait comme on le souhaite, même de manière naïve ?

Exercice 6 : En considérant une structure contenant les médianes ainsi obtenues, comment veut-on faire le calcul de la médiane de cette structure ?

Pour se faciliter la vie dans les calculs de complexité, on suppose tous les éléments distincts deux à deux.

Exercice 7 : Soit `M5` la médiane calculée en suivant le principe de la question précédente. Correspond-elle à `M` ? Donner une borne inférieure et une borne supérieure de la proportion d'éléments de la structure de départ (avant constitution des sous-structures de taille cinq) qui sont inférieures à `M5`.

La suite n'est pas très intuitive, donc le principe va être révélé ici : avec `M5`, on va éliminer une partie des éléments de la structure de départ car soit on sait qu'ils sont trop grands pour être la médiane soit on sait qu'ils sont trop petits. Il s'agit de simuler (sur une copie de la structure de départ par exemple) une étape de filtrage du tri rapide pour déterminer la vraie place de `M5`, par rapport au « milieu » de la structure où doit trôner la médiane. La zone contenant le milieu sera traitée dans une étape suivante de l'algorithme appelant de nouveau toutes les fonctions impliquées, jusqu'à avoir effectivement trouvé la médiane.

Exercice 8 : Majorer la complexité de toutes les fonctions impliquées, en faisant intervenir des formules de récurrence.

On admettra le résultat suivant : Si une complexité est donnée sous la forme de $c_n = (\sum_{k=1}^{p} c_{n_k}) + \mathcal{O}(n)$ avec $\sum_{k=1}^{p} n_k < \alpha n$ pour un réel $\alpha < 1$, alors $c_n = \mathcal{O}(n)$. (Ce résultat est aussi valable avec Θ dans les deux cas.) Il permet de conclure.

Exercice 9 : Écrire maintenant tout ceci en C ou en OCaml.

Exercice 10 : [Cerise sur le gâteau] Écrire un tri rapide en C ou en OCaml utilisant la médiane calculée en temps linéaire.

TD 8 : Algorithmes gloutons, diviser pour régner

Dans ce TD, des exemples de problèmes classiques pour illustrer les algorithmes gloutons (optimaux dans un premier temps, pour se conformer au programme de première année) ou les algorithmes diviser pour régner sont présentés et traités.

Comme d'habitude en TD, l'important est l'algorithmique, et la programmation se fait dans un deuxième temps. On pourra écrire les programmes de la première section en C et en OCaml, en se posant la question des structures de données à employer.

Ceci est un TD long et deux séances y seront consacrées ; le déséquilibre entre les deux parties fait qu'il n'était pas pertinent de le scinder en deux TD.

Algorithmes gloutons

Exercice 1 : Écrire un algorithme glouton pour le problème du rendu de monnaie en euros (en centimes d'euros, en pratique, pour rester dans les entiers).

Exercice 2 : Écrire un algorithme glouton pour le problème de sélection des activités, détaillé ci-après. On ne mettra pas en œuvre d'algorithme de tri pour cet exercice.

Le problème de sélection des activités consiste à choisir dans une liste d'activités, chacune représentée par un temps de début et un temps de fin, une sous-liste d'activités ne se chevauchant pas et de la plus grande taille possible.

On ne considère pas que deux activités se chevauchent si le temps de fin de l'une est égal au temps de début de l'autre, seulement s'il est strictement supérieur.

Ici, l'algorithme glouton est optimal à condition de trouver le bon critère discriminant les choix d'activité…

Exercice 3 : Écrire un algorithme glouton pour le problème d'allocation des salles de cours, détaillé ci-après.

Le problème d'allocation des salles de cours, qu'on pourra rapprocher du problème précédent, consiste à sélectionner pour chaque cours, représenté par un temps de début et un temps de fin, une salle où il peut avoir lieu, sans que deux cours n'aient lieu en même temps dans une même salle, et en minimisant le nombre de salles.

Là aussi, on ne considère pas que deux cours ont lieu en même temps si le temps de fin de l'un est égal au temps de début de l'autre.

Le problème classique en informatique correspondant est celui de la coloration de graphes (cette structure sera vue plus tard dans ce semestre), et bien que pour ce problème l'algorithme glouton n'est pas optimal en général, il s'avère que c'est tout de même le cas dans les graphes obtenus en recensant les incompatibilités entre cours, en affectant à chaque cours la salle du plus petit identifiant possible.

Pour aller plus loin, ce problème a été mis à l'honneur par le sujet d'option informatique au concours Centrale en 2015.

Diviser pour régner

Exercice 4 (Méthode de Karatsuba pour la multiplication de deux polynômes) : Pour calculer $P \times Q$, où P et Q sont deux polynômes de degré $2n-1$ (au plus), on écrit $P = P_1 + X^n P_2$ et $Q = Q_1 + X^n Q_2$, où les quatre nouveaux polynômes sont de degré au plus $n-1$. On sait que $PQ = P_1Q_1 + X^n(P_1Q_2 + P_2Q_1) + X^{2n}P_2Q_2$, cependant on peut se limiter à trois multiplications en calculant $(P_1 + P_2)(Q_1 + Q_2)$. Prouver qu'on obtient bien le produit souhaité en trouvant les deux autres produits à faire, et donner la complexité de l'algorithme (puis l'écrire en C et en OCaml). [8] Consulter le web pour y découvrir l'algorithme de Strassen permettant la multiplication de deux matrices en un temps meilleur que cubique, précisément en $\mathcal{O}(n^{\log_2(7)})$.

Exercice 5 : Trouver un algorithme DPR pour déterminer l'enveloppe convexe d'un nuage de points. L'écrire en OCaml.

Le principe est de calculer l'enveloppe convexe des deux moitiés du nuage contenant les points les plus à gauche et les points les plus à droite (par exemple), puis de rassembler ces enveloppes en ajoutant exactement deux lignes (et en en supprimant un certain nombre).

Pour aller plus loin, ce problème a fait l'objet du concours blanc d'option première année en 2018 (et du sujet d'informatique B au concours X-ENS de 2015, avec deux algorithmes DPR différents dans les deux sujets).

8. Pour aller plus loin, un principe encore plus efficace et utilisant aussi le DPR est de passer par la transformée de Fourier rapide, la complexité en temps sera alors un $\mathcal{O}(n\log(n))$.

Exercice 6 : Trouver un algorithme DPR qui calcule le nombre d'inversions dans un tableau ou dans une liste (au choix). L'écrire en OCaml.

Une inversion dans `t` est un couple `(i,j)` tel que `i < j` et `t.(i) > t.(j)`.

Le principe est de faire un tri fusion et de compter les inversions au moment de fusionner : les inversions d'un tableau sont les inversions de sa moitié gauche, les inversions de sa moitié droite et pour chaque élément de la moitié droite le nombre d'éléments supérieurs dans la moitié gauche, nombre qu'on peut déterminer en temps constant par le principe annoncé.

Pour aller plus loin, ce problème a fait l'objet du concours blanc d'option première année en 2019.

Exercice 7 : Trouver un algorithme DPR qui détermine les deux points les plus proches dans un nuage de points du plan. L'écrire en OCaml.

Il s'agit de disposer d'informations suffisantes sur le nuage de points, on créera donc deux tableaux : l'un trié par abscisses croissantes (et par ordonnées croissantes à même abscisse) et l'autre trié par ordonnées croissantes.

On peut séparer le tableau trié en abscisses en deux moitiés de taille égale sur lesquelles on applique récursivement ce principe jusqu'à ce que la taille du tableau soit assez petite pour qu'un algorithme naïf suffise (moins de 6 points par exemple).

Si on dispose d'un couple de points minimisant la distance pour chaque moitié (disons que `d` est la plus petite des deux distances obtenues), il faut vérifier qu'aucun couple dont les éléments sont chacun dans une moitié ne donne une distance inférieure à `d`, mais si c'était le cas ils seraient dans une bande d'abscisse de largeur inférieure ou égale à `2*d`, et le nombre de points du nuage dans cette bande est assez faible (car ils sont espacés d'au moins `d`).

En extrayant les points dont l'abscisse est dans la bande du tableau trié par ordonnées croissantes, on peut déterminer en temps linéaire s'il y a une distance inférieure à `d` dans le tableau obtenu, car l'étroitesse de la bande permet de limiter la recherche à une zone constante du tableau (faire un dessin pour le prouver).

TD 9 : Master theorem

Comme le TD 7, ce TD traite d'un point technique qui n'est pas au programme (en pratique explicitement hors-programme), mais le raisonnement est suffisamment découpé pour être à la portée de la classe.

Le théorème de récurrence des partitions

Nous allons prouver la version la plus simple du théorème permettant de donner directement la complexité en temps de la plupart des algorithmes DPR.

En voici l'énoncé :

Théorème

Soit un algorithme dont la complexité est définie par une relation de la forme $c_n = ac_{\frac{n}{b}} + \Theta(n^\alpha)$, *avec* $a \in \mathbb{N} \setminus \{0\}, b \in \mathbb{N} \setminus \{0,1\}$ *et* $\alpha \in \mathbb{R}_+$. *Alors :*

- *Si* $\alpha < \log_b(a)$, *alors* $c_n = \Theta(n^{\log_b(a)})$.
- *Si* $\alpha > \log_b(a)$, *alors* $c_n = \Theta(n^\alpha)$.
- *Si* $\alpha = \log_b(a)$, *alors* $c_n = \Theta(n^\alpha \log_b(n))$.

Remarque : Ce théorème se généralise aux relations de type $c_n = ac_{\frac{n}{b}} + f(n)$, où il s'agit de comparer $n^{\log_b(a)}$ et $f(n)$ en termes de domination.

Dans l'ensemble de cette partie, on étudie la formule de récurrence $c_n = ac_{\frac{n}{b}} + \Theta(n^\alpha)$ avec les contraintes ci-avant pour les variables et d'éventuelles restrictions sur certaines questions.

La valeur de c_1 sera fixée arbitrairement à 1.

Jusqu'à la dernière question, on supposera que n puisse s'écrire sous la forme d'une puissance de b, donc $\exists k \in \mathbb{N}, n = b^k$.

On introduira alors la suite $(T_k)_{k\in\mathbb{N}}$ définie par $T_k = c_{b^k}$.

Les exercices commencent par un peu d'échauffement mathématique puis les raisonnements s'étoffent (mais restent mathématiques).

Exercice 1 : Plaçons-nous dans un cas annexe où le $\Theta(n^\alpha)$ a été retiré. Donner la formule de c_n et la prouver par récurrence.

Exercice 2 : Montrer que $n^{\log_b a} = a^{\log_b n}$.

Exercice 3 : Écrire une équation sans logarithme équivalente à l'équation $\log_b a = \alpha$.

Exercice 4 : Supposons que $\log_b a < \alpha$. Donner les croissances comparées de $a^{\log_b n}$ et n^α pour $n \to +\infty$.

Exercice 5 : Supposons encore que $\log_b a < \alpha$. Soit $k \in \mathbb{N}$ non nul. Montrer qu'il existe une constante $C \in]0, 1[$ telle que $a(b^{k-1})^\alpha < C(b^k)^\alpha$.

Exercice 6 : Supposons cette fois-ci que $\log_b a > \alpha$. Adapter le résultat des deux questions précédentes.

Exercice 7 : Supposons que le $\Theta(n^\alpha)$ est exactement n^α. Montrer que T_k s'exprime sous la forme $\sum_{i=0}^{k} a^{k-i}(b^\alpha)^i$.

Exercice 8 : Transformer la somme précédente pour qu'on voie clairement apparaître une somme de termes d'une suite géométrique, et écrire alors la valeur de cette somme. Attention à la distinction de cas à ne pas oublier !

Exercice 9 : En rassemblant tous les résultats précédents, déduire le théorème dans le cas restreint où n est une puissance de b et où le $\Theta(n^\alpha)$ est exactement n^α.

Exercice 10 : Par un encadrement pertinent, lever la restriction sur le $\Theta(n^\alpha)$.

Exercice 11 : En partant du principe que c_n est croissante, généraliser à toute valeur de n par un encadrement entre deux puissances de b.

Exercice 12 : Se servir du théorème nouvellement prouvé pour justifier la complexité du produit de deux polynômes par l'algorithme de Karatsuba et celle du produit de deux matrices par l'algorithme de Strassen, tous deux issus du TD précédent.

TD 10 : Programmation dynamique

Dans ce TD, des applications de la programmation dynamique sont présentées.

Les programmes sont à écrire en C et en OCaml. Pendant la séance, ce sera un langage par exercice, pour avancer.

Exercice 1 : Écrire un programme dynamique déterminant le nombre minimal de multiplications de scalaires à faire pour multiplier n matrices de dimensions variées.

Par exemple, pour calculer $M_1M_2M_3M_4$, où les dimensions respectives des matrices sont $(4,6)$, $(6,2)$, $(2,10)$ et $(10,3)$, les multiplications peuvent être faites ainsi [9] :

- $M_1(M_2(M_3M_4))$ $(2\times 10\times 3+6\times 2\times 3+4\times 6\times 3$ soit 168 multiplications) ;
- $(M_1M_2)(M_3M_4)$ $(4\times 6\times 2+2\times 10\times 3+4\times 2\times 3$ soit 132 multiplications) ;
- $M_1((M_2M_3)M_4)$ $(6\times 2\times 10+6\times 10\times 3+4\times 6\times 3$ soit 372 multiplications) ;
- $((M_1M_2)M_3))M_4$ $(4\times 6\times 2+4\times 2\times 10+4\times 10\times 3$ soit 248 multiplications) ;
- $(M_1(M_2M_3))M_4$ $(6\times 2\times 10+4\times 6\times 10+4\times 10\times 3$ soit 480 multiplications).

Exercice 2 : Écrire un programme dynamique pour le problème du rendu de monnaie dans le cas général où on ne peut pas garantir qu'un algorithme glouton soit optimal. On pourra là aussi calculer simplement le nombre optimal de pièces et billets à rendre ou donner le détail.

Exercice 3 (Ordonnancement de tâches pondérées - *weighted interval scheduling*) : Écrire un programme dynamique déterminant la valeur maximale d'un ensemble de tâches effectuables, les tâches étant définies par leur heure de début, leur heure de fin et leur valeur, et une seule tâche pouvant être accomplie à la fois. Il s'agit de l'extension du problème de sélection d'activités au cas où on n'optimise pas forcément le nombre de tâches.

---

9. Pour information, le nombre de façons d'organiser les multiplications de $n+1$ matrices est le n-ième nombre de Catalan, donné par les formules équivalentes

$$C_n = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!} = \binom{2n}{n} - \binom{2n}{n-1}.$$

Le nombre de Catalan se retrouve très souvent en combinatoire avancée.

Exercice 4 (Distance d'édition) : Soient deux chaînes de caractères s_1 et s_2. La distance d'édition de s_1 à s_2 correspond au coût minimal pour passer de s_1 à s_2 (coût qui n'est pas symétrique a priori) en nombre d'insertions de caractères, de modifications de caractères et de suppressions de caractères (n'importe où dans les trois cas). On associe à chacune de ces opérations un coût constant, mais pas nécessairement égal entre les opérations (on considère dans le cas simple que les coûts sont tous d'un). Écrire un programme dynamique qui calcule la distance d'édition entre deux chaînes de caractères dans le cas simple puis dans le cas général. Par la suite, il est possible d'aligner les séquences (comme on le fait en génétique) pour voir comment cette distance d'édition est minimisée.

Exercice 5 : Écrire un programme dynamique pour le problème du sac à dos.

Pour information ou rappel, le problème du sac à dos est le suivant : étant donné un ensemble d'objets d'un certain poids et d'une certaine valeur, comment remplir un sac à dos avec un ou plusieurs exemplaire(s) (certaines variantes excluent cependant d'en prendre plusieurs) de certains objets de sorte que la valeur transportée soit maximale et le poids soit inférieur à un seuil donné ?

L'argument de la fonction sera donc une liste de couples (poids puis valeur pour se mettre d'accord) et la valeur de retour la liste des objets retenus. Une autre possibilité est d'ajouter un nom aux couples, pour ne renvoyer que la liste des noms des objets retenus.

Le problème demande normalement de renvoyer la valeur maximale sans détailler les objets à transporter, mais le programme devra retourner les deux (ou au moins la liste des objets, dont poids et valeur seront déduits).

Exercice 6 (long, à faire chez soi) : Écrire en OCaml un programme dynamique pour résoudre toutes les instances du jeu 24, détaillé ci-après.

Le jeu 24 se rapproche du coup de chiffres de l'émission « Des chiffres et des lettres ».

Seule la partie qui a un intérêt algorithmique est présentée ici (pour en dire légèrement plus, on utilise les 40 cartes numériques d'un paquet de 52 dans la pratique) : il s'agit d'obtenir le nombre `24` grâce aux quatre opérations arithmétiques usuelles (d'autres sont possibles dans des variantes) en partant de quatre entiers entre `1` et `10` fournis à l'avance et non nécessairement distincts.

Il est obligatoire d'utiliser les quatre entiers fournis, et il est autorisé de passer par des nombres négatifs voire non entiers.

En l'occurrence, comme les nombres resteront rationnels (hors variante...), on utilisera des couples d'entiers représentant tous les nombres impliqués, en tant que fraction simplifiée (le premier élément du couple représentera le numérateur et le deuxième, nécessairement strictement positif et premier avec l'autre, représentera le dénominateur).

Le travail se décomposera en plusieurs étapes :

- Écrire les opérations arithmétiques sur les couples d'entiers en tant que fonctions annexes.
- Créer à l'aide du module `Hashtbl` un dictionnaire `d` dont les clés seront tous les 1-uplets, couples, triplets et quadruplets d'entiers (croissants pour éviter l'ambiguïté) entre `1` et `10` et les valeurs seront des dictionnaires vides.
- Initialiser les valeurs dans `d` pour tous les 1-uplets `(k, )` en y mettant `{ (k, 1) : "k"}`, signifiant qu'on peut obtenir le nombre `k`, représenté par le couple `(k, 1)`, en écrivant simplement ce nombre.
- En déduire les valeurs dans `d` pour tous les couples formés à partir de deux 1-uplets par l'application des opérations élémentaires.
- En déduire les valeurs dans `d` pour tous les triplets formés à partir d'un couple et d'un 1-uplet de la même façon.
- En déduire les valeurs dans `d` pour tous les quadruplets formés à partir de deux couples ou d'un triplet et d'un 1-uplet, mais cette fois-ci seul le couple `(24, 1)` sera intéressant à stocker.
- Retourner `d` dont seuls les quadruplets seront conservés en tant que clés.

TD 11 : Applications des arbres

Les trois applications des arbres évoquées dans le chapitre 11 sont abordées ici, avec quelques exercices faisant intervenir ou non de la programmation. **Tout ce TD se fera en OCaml exclusivement.**

Ceci est un TD long et deux séances y seront consacrées.

Expressions arithmétiques

Considérons un fragment simple de l'ensemble des expressions arithmétiques. On se limitera à l'utilisation d'entiers, aux quatre opérations de base (la division étant le quotient dans la division euclidienne) et aux parenthèses, en s'assurant que la syntaxe soit respectée.

En OCaml, un type permettant de représenter une expression arithmétique sous cette restriction est alors :

```
type ea = Nombre of int | Plus of ea * ea | Moins of ea * ea |
Fois of ea * ea | Div of ea * ea;;
```

Ici, il n'y a pas de parenthèses, celles-ci servent à lever l'ambigüité dans une chaîne de caractères à partir de laquelle on construit l'expression, qui sera alors représentée sous forme d'un arbre, dont les nœuds internes seront associés à un constructeur récursif, les sous-arbres étant obtenus en représentant les paramètres du constructeur, et les feuilles au constructeur `Nombre` avec son paramètre.

Le type associé pour les arbres sera alors (le caractère au niveau du nœud étant par exemple l'opérateur tel qu'on l'écrit usuellement) :

```
type aea = Noeud of char * aea * aea | Feuille of int;;
```

Exercice 1 : Écrire une expression arithmétique représentant l'opération `2 + 3 * 5` avec les priorités opératoires usuelles. Écrire et dessiner l'arbre correspondant. Faire ensuite de même pour `(2 + 3) * 5`.

Exercice 2 : Écrire une fonction qui évalue une expression arithmétique et une fonction qui évalue un arbre d'expression arithmétique.

Exercice 3 : Écrire les deux fonctions de conversion d'un type vers l'autre.

Exercice 4 : Écrire une fonction d'impression d'une expression pour chacun des deux types. L'expression imprimée doit pouvoir être évaluée correctement.

Dans le TP 8, nous avons vu la mention de la notation polonaise inversée. Il s'avère qu'on peut passer d'une expression en notation polonaise inversée à un arbre de syntaxe abstraite (ou à une expression au sens du premier type donné) en utilisant une pile comme pour la fonction d'évaluation du TP susmentionné, où là aussi les nombres sont mis en attente et les opérateurs permettent de construire depuis les feuilles (ou l'intérieur) la version finale de l'expression.

Exercice 5 : Écrire une fonction de conversion d'une expression en notation polonaise inversée (liste de chaînes de caractères avec les mêmes restrictions que ci-avant) vers l'un ou l'autre des types de cette section.

La construction d'une expression en notation polonaise inversée peut se faire avec l'algorithme dit Shunting-yard (c'est-à-dire « gare de triage »), lui aussi dû à Dijkstra. Cet algorithme consiste à utiliser une pile (pour changer) d'opérateurs en attente pour matérialiser les priorités supérieures d'opérateurs éventuellement à venir.

Son argument sera une liste de « jetons », c'est-à-dire d'unités lexicales déjà identifiées, pour simplifier. Sa valeur de retour sera une autre liste de jetons pouvant être utilisée dans l'exercice précédent.

Le fonctionnement de l'algorithme restreint aux conditions de l'exercice est le suivant. Pour tous les jetons :

- Si c'est une chaîne représentant un entier, l'ajouter à la sortie.
- Si c'est un opérateur, dépiler et ajouter à la sortie tous les opérateurs de priorité égale ou supérieure (attention à s'arrêter si on rencontre une parenthèse ouvrante) puis empiler l'opérateur lu. Le seul cas sans dépilement est quand un opérateur multiplicatif est lu alors qu'un opérateur additif est dans la pile.
- Si c'est une parenthèse ouvrante, l'empiler.
- Si c'est une parenthèse fermante, dépiler tous les opérateurs jusqu'à la première parenthèse ouvrante rencontrée et les mettre dans la sortie (pas la parenthèse, évidemment).

À la fin, dépiler et ajouter à la sortie tous les opérateurs.

Si une parenthèse existe encore dans la pile, il y a une erreur de syntaxe. De même si le nombre total d'opérateurs n'est pas exactement un de moins que le nombre de chaînes représentant des entiers.

Exercice 6 : Écrire une fonction réalisant cet algorithme, en déclenchant une erreur si la syntaxe est incorrecte.

Tries

Dans cette section, nous allons nous intéresser aux arbres préfixes, autre nom des tries. La structure employée sera celle des arbres d'arité quelconque, où chaque nœud est étiqueté par un booléen et un caractère (le caractère associé à la racine étant ignoré) de sorte que deux fils d'un même nœud soient étiquetés par des caractères différents, et en suivant une branche la concaténation des caractères à partir du premier fils de la racine donne soit un mot inséré dans l'arbre (le booléen du nœud actuel est vrai), soit le préfixe d'un tel mot (le booléen est faux et le nœud actuel doit avoir au moins un fils).

Les opérations élémentaires sont la création d'un trie, l'ajout d'un mot, la suppression d'un mot (attention, ce n'est pas évident) et le test d'appartenance d'un mot.

Exercice 7 : Écrire un type associé.

Exercice 8 : Écrire et dessiner l'arbre correspondant à la liste de mots `"BAR"`, `"BARBE"`, `"BARBU"`, `"BARRE"`, `"BARRER"`, `"LOU"`, `"LOUP"`, `"LOUVE"`, `"LOUER"` et `"LOUVOYER"`.

Exercice 9 : Écrire les quatre opérations élémentaires sur les tries.

Un arbre PATRICIA est un trie dont tous les nœuds pour lesquels le booléen de l'étiquette est faux et n'ayant qu'un fils sont court-circuités. Il y aura alors une chaîne de caractères de taille au moins un au lieu du caractère d'un trie.

Exercice 10 : Écrire une fonction de conversion d'un trie à un arbre PATRICIA et la fonction réciproque.

Arbres de décision

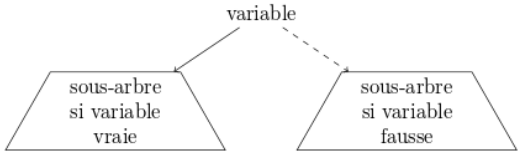
Comme annoncé en cours, les arbres de décision étaient à l'honneur dans le sujet d'option informatique en 2013 à Centrale. En voici une transcription.

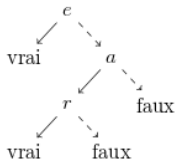
Un *arbre de décision* est un arbre binaire dans lequel un nœud interne est associé à une variable, parmi un ensemble V de variables et une feuille est associée à un booléen (vrai ou faux).

Si chaque variable de l'ensemble V reçoit une valeur booléenne, un tel arbre permet de prendre une décision en parcourant l'arbre : on part de la racine, quand on arrive sur un nœud interne (racine comprise), on regarde quelle est la valeur de la variable associée au nœud, si elle vaut vrai on poursuit le parcours dans le sous-arbre gauche, sinon on poursuit le parcours dans le sous-arbre droit, quand on arrive sur une feuille, le booléen associé constitue la décision.

On représente par convention l'arête menant au sous-arbre pour le « cas vrai » en trait plein, l'arête menant au sous-arbre « cas faux » en pointillés.

Schématiquement, un arbre est structuré comme indiqué ci-après à gauche.





Le schéma de droite ci-avant illustre l'exemple : un module de cours est validé si l'examen est réussi (e), ou sinon, si l'étudiant a été assidu en cours (a) et qu'il réussit un examen de rattrapage (r).

On envisage une représentation simple d'un arbre de décision, à l'aide d'un tableau. On numérote les nœuds : la racine reçoit le numéro `0`, les autres nœuds sont numérotés arbitrairement par des entiers consécutifs à partir de `1`.

Le tableau contient donc autant de cases que de nœuds. La case d'indice `i` contient soit un triplet (nom de variable, numéro du fils gauche, numéro du fils droit) si le nœud numéro `i` est un nœud interne, soit un booléen si le nœud numéro `i` est une feuille.

En OCaml, on définit le type :

```
type noeud = Feuille of bool | Decision of string * int * int;;
```

Un arbre de décision est donc représenté par un tableau de nœuds (type `noeud array`).

Exercice 11 : Créer une variable `monAD` représentant l'arbre de décision illustré précédemment.

Dans les deux questions suivantes, on veut faire déterminer une décision en fournissant la liste des seules variables qui sont vraies dans l'évaluation.

Exercice 12 : Écrire une fonction `eval_var` qui, étant donnés le nom d'une variable (`string`) et une liste des seules variables vraies, renvoie un booléen correspondant à la variable indiquée.

Exercice 13 : Écrire une fonction `eval` qui, étant donnés un arbre de décision et une liste des seules variables vraies, renvoie un booléen correspondant à la décision finale.

En pratique, les *diagrammes de décision* supplantent les arbres de décision. Il s'agit de compacter la représentation en mémoire : si plusieurs sous-arbres sont identiques, on n'a pas envie de les stocker plusieurs fois.

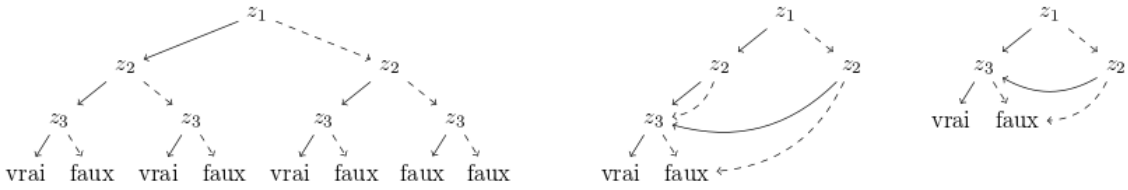
En raisonnant sur la représentation informatique des arbres de décision, on voit assez facilement une façon de procéder : si les arbres de numéros `i` et `j` sont identiques, on peut (par exemple) au niveau du parent `p` de `j` indiquer comme numéro de fils `i` au lieu de `j` et ainsi éliminer `j` de la représentation.

Ce faisant, on ne représente plus un arbre (car `i` a maintenant deux parents), mais un graphe orienté.

On dira qu'il existe un arc de `p` vers `i` et on le notera $\texttt{p} \xrightarrow{b} \texttt{i}$, avec b booléen, selon que l'arc est suivi dans le cas où `p` est vrai ou faux (ce qui correspondait aux fils gauche et droit).

On note de manière équivalente $\texttt{i} = \text{succ}_b(\texttt{p})$.

Exemple : l'expression $(z_1 \wedge z_3) \vee (z_2 \wedge z_3)$ admet (entre autres) les diagrammes de décision ci-dessous.



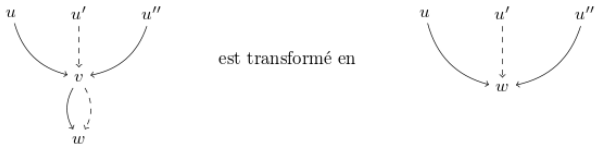
Exercice 14 : Écrire une fonction `redirige` à trois paramètres, un diagramme ainsi que deux indices `v` et `w`, qui supprime le nœud `v` dans le graphe et transforme tous les arcs $\mathtt{u} \xrightarrow{b} \mathtt{v}$ en $\mathtt{u} \xrightarrow{b} \mathtt{w}$. Les cases du tableau qui deviennent inoccupées sont remplies avec la valeur spéciale `Vide`.

Pour ce faire en OCaml on complète :

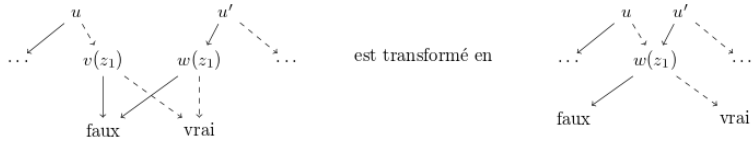
```
type noeud = Feuille of bool | Decision of string * int * int | Vide;;
```

Pour transformer un arbre en diagramme sans répétition, on applique deux règles de simplification.

Élimination : Si pour un nœud `v` on a $\mathrm{succ}_F(\mathtt{v})=\mathrm{succ}_T(\mathtt{v})=\mathtt{w}$ alors on élimine `v` et on transforme les arcs $\mathtt{u} \xrightarrow{b} \mathtt{v}$ en $\mathtt{u} \xrightarrow{b} \mathtt{w}$.



Isomorphisme : Soient `v` et `w` deux nœuds différents. Si ce sont des feuilles avec la même valeur de vérité ou si ce sont des nœuds internes associés à la même variable et tels que $\mathrm{succ}_b(\mathtt{v})=\mathrm{succ}_b(\mathtt{w})$ quel que soit le booléen b, alors on élimine `v` et on transforme les arcs $\mathtt{u} \xrightarrow{b} \mathtt{v}$ en $\mathtt{u} \xrightarrow{b} \mathtt{w}$.



Exercice 15 : Écrire une fonction `trouve_elimination`, prenant en paramètre un diagramme et renvoyant l'indice d'un nœud pouvant être supprimé par élimination, s'il en existe un. Sinon, elle doit renvoyer `-1`.

Exercice 16 : De même, écrire une fonction `trouve_isomorphisme`, prenant en paramètre un diagramme et renvoyant un couple d'indices correspondant à deux nœuds pouvant être simplifiés par isomorphisme, s'il en existe un. Sinon, elle doit renvoyer le couple `(-1,-1)`.

On dit que le diagramme est *sous forme réduite* s'il n'existe pas de nœuds différents qui correspondent à la même formule logique.

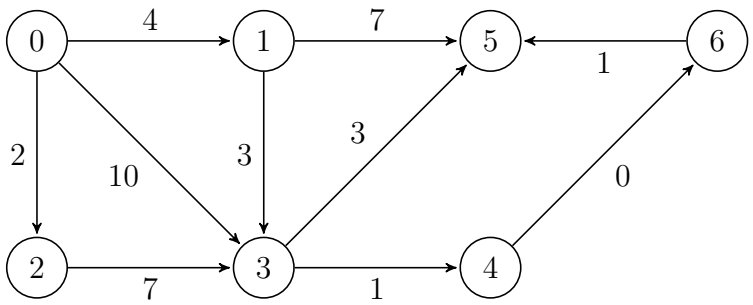
Exercice 17 : Écrire une fonction `reduit` prenant en paramètre un diagramme, qui détecte les deux simplifications possibles et effectue les redirections correspondantes, jusqu'à ce qu'il ne soit possible de faire aucune simplification supplémentaire.

On obtient à ce stade une représentation du diagramme simplifié sous forme d'un tableau dans lequel certaines cases ne sont plus utilisées : elles sont marquées `Vide`.

TD 12 : Un graphe sommaire

L'objectif de ce TD est de se familiariser avec les notions vues en cours sur les graphes à partir d'un exemple simple.

Nous travaillerons sur ce graphe pondéré :



Pour les premières questions, la pondération des arcs sera ignorée.

Exercice 1 : Combien le graphe a-t-il de sommets, d'arcs ? Donner une expression de ce graphe selon les trois représentations vues en cours.

Exercice 2 : Donner un chemin de 0 à 6 dans le graphe. Ce graphe est-il fortement connexe ?

Exercice 3 : Ce graphe admet-il des chemins hamiltoniens ? eulériens ?

Exercice 4 : Dessiner la fermeture transitive du graphe. En déduire ses composantes fortement connexes.

Exercice 5 : Réaliser à la main un parcours en profondeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet en fonction du numéro associé (les petits en premier).

Exercice 6 : Réaliser à la main un parcours en largeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet en fonction du numéro associé (les petits en premier).

Désormais, on considèrera les poids des arcs.

Exercice 7 : Est-il possible de lancer l'algorithme de Dijkstra sur ce graphe et pourquoi ?

Exercice 8 : À l'aide d'un algorithme au choix, déterminer à la main un chemin de poids minimal depuis le sommet 0 et vers chaque autre sommet, ainsi que le poids du chemin en question. En profiter pour restreindre le graphe à un arbre réalisant des chemins optimaux.

Exercice 9 : Écrire dans un langage au choix l'algorithme de Dijkstra et tester cet algorithme sur le graphe de ce TD.

Exercice 10 : Écrire dans un langage au choix l'algorithme de Floyd-Warshall et tester cet algorithme sur le graphe de ce TD.

Exercice 11 : Écrire dans un langage au choix l'algorithme de Bellman-Ford et tester cet algorithme sur le graphe de ce TD.

TD 13 : Algorithmique du texte

Ce TD est l'occasion de faire tourner à la main les algorithmes du chapitre 15, mais aussi de les écrire (sauf Huffman, qui a son TP à part).

Ceci est un TD long et deux séances y seront consacrées.

Recherche dans un texte

Algorithme naïf

Exercice 1 : Écrire en C une fonction prenant en argument deux chaînes de caractères, la deuxième étant le motif, et retournant la liste des positions où le motif est trouvé dans l'autre chaîne.

Algorithme de Boyer-Moore

Exercice 2 : Dessiner la table de décalage pour l'algorithme de Boyer-Moore simplifié associé au motif `"CHERCHEZ"`, en tant que tableau à double entrée indexé en ligne par les nombres de `0` à `7` et en colonne par les cinq caractères apparaissant dans le motif.

Exercice 3 : Donner le nombre de comparaisons entre caractères de l'algorithme naïf et de l'algorithme de Boyer-Moore avec le motif `"CHERCHEZ"` sur les trois chaînes contenant $n \geq 1$ répétitions de `"CHERCHER"`, `"CHERCHEZ"` et `"CHEZCHEZ"`, respectivement. Pour les deux algorithmes, la liste des positions de départ du motif étant attendue, on n'arrêtera pas le calcul sur le premier succès.

Exercice 4 : En pratique, on peut améliorer la table de décalages en profitant du fait qu'aucun suffixe de `"CHERCHEZ"` ne se retrouve ailleurs dans la chaîne. Donner la version optimisée de cette table et la donner aussi pour le motif `"CHERCHER"`, où l'optimisation est plus subtile. Recalculer le nombre de comparaisons entre caractères sous les mêmes conditions que dans l'exercice 3 mais avec la table optimisée.

L'optimisation en question ne sera pas mise en œuvre dans le programme, cependant !

Exercice 5 : Écrire en OCaml une fonction réalisant l'algorithme de Boyer-Moore. Les conditions sont les mêmes que pour l'algorithme naïf.

Algorithme de Rabin-Karp

Exercice 6 : Calculer le haché de `"mot"` pour $b = 31$. La calculatrice est interdite ! On rappelle que la position de `'a'` dans la table ASCII est 97.

Exercice 7 : En admettant que chaque facteur soit possible, donner pour le même b et la même taille de motif le plus petit haché possible (même si la chaîne associée est absurde), le plus grand haché possible et le nombre de hachés possibles.

On en déduit l'injectivité du hachage quand on calcule un résidu modulo un nombre premier « assez grand » pour des motifs « assez petits », surtout si la diversité des caractères est limitée (on peut imaginer que seulement un quart de la table ASCII-8-bits est vraiment utilisé).

Exercice 8 : Écrire en C la fonction de hachage du cours pour le motif, avec les arguments nécessaires.

Exercice 9 : Écrire en C la fonction de passage du haché d'une tranche au haché de la tranche suivante, toujours avec les arguments nécessaires.

Exercice 10 : Écrire en C une fonction réalisant l'algorithme de Rabin-Karp. Les conditions sont les mêmes que pour l'algorithme naïf. On considèrera les arguments utilisés pour la fonction de hachage comme des arguments du programme, mais la chaîne et le motif seront écrits dans la fonction `main`.

Compression

Algorithme de Huffman

Exercice 11 : Écrire l'arbre de Huffman associé à seize caractères ayant tous la même fréquence d'apparition. Commenter.

Exercice 12 : Écrire l'arbre de Huffman associé à six caractères ayant des fréquences d'apparition en progression géométrique de raison 2 avec deux occurrences du minimum. Commenter.

Exercice 13 : Dessiner l'arbre de Huffman associé au premier paragraphe d'un roman au choix.

Exercice 14 : Montrer que l'arbre obtenu par l'algorithme de Huffman minimise la somme sur tous les caractères des produits du nombre d'apparitions de ce caractère et du nombre de bits utilisés pour l'écrire, parmi tous les codes préfixes possibles.

Algorithme de Lempel-Ziv-Welch

Exercice 15 : Faire tourner à la main l'algorithme de Lempel-Ziv-Welch sur une chaîne contenant un certain nombre d'occurrences du même caractère et donner le résultat en fonction de la taille.

Exercice 16 : Faire tourner à la main l'algorithme de Lempel-Ziv-Welch sur une chaîne contenant une alternance stricte de deux caractères et donner « la tendance » en fonction de la taille.

Exercice 17 : Donner une chaîne de caractères pour laquelle l'algorithme de Lempel-Ziv-Welch ne réduit pas la taille du résultat par rapport à l'entrée. Peut-on trouver des exemples arbitrairement grands ? (Et en pratique, en est-on facilement capable ?)

Exercice 18 : Écrire en OCaml une fonction réalisant la compression dans n'importe quelle variante de l'algorithme de Lempel-Ziv-Welch.

Exercice 19 : Écrire en OCaml une fonction réalisant la décompression correpondante.

Exercice 20 : Tester sur diverses chaînes la fonction de l'exercice 18 et constater le taux de compression.

TD 14 : Autour de la logique

Dans ce TD, divers exercices indépendants permettent de tourner autour des compétences en logique acquises en première année.

Ceci est un TD long et deux séances y seront consacrées.

Exemple de sujet de CCINP

Au début des années 2010, la tendance du sujet d'option informatique de ce qui s'appelait alors CCP était de commencer par une partie consistant en une énigme logique mise en contexte.

Voici une transcription de cette partie dans le sujet de l'année 2012.

Exercice 1 : Résoudre les questions à suivre.

Vous participez à un concours de mathématiques comportant une partie de raisonnement logique. Plusieurs orateurs font des déclarations et vous devez répondre à des questions en vous appuyant sur des informations déduites de ces déclarations. La règle suivante s'applique : « Les orateurs sont de trois natures : les véridiques, les menteurs et les changeants. Les véridiques disent toujours la vérité, les menteurs mentent toujours, et les changeants disent en alternance une vérité et un mensonge (c'est-à-dire, soit une vérité, puis un mensonge, puis une vérité, etc. ; soit un mensonge, puis une vérité, puis un mensonge, etc.). Pendant tout le concours, les orateurs ne peuvent pas changer de nature. »

Les épreuves comportent deux phases :

— Les différents orateurs font plusieurs déclarations dont l'analyse permet de déterminer la nature de chaque orateur (véridique, menteur, changeant commençant par dire la vérité, ou changeant commençant par dire un mensonge).

— Les orateurs font une seconde série de déclarations. Puis, vous devez répondre à des questions en exploitant les informations contenues dans ces déclarations.

Question 1.1 : Dans la première phase, quel est le nombre minimum de déclarations que doit faire chaque orateur pour qu'il soit possible de déterminer sa nature ? Justifier.

Question 1.2 : Soit un orateur A qui fait une suite de n déclarations A_i. Proposer des formules du calcul des propositions A_V, A_M, A_{CV} et A_{CM} qui permettent de caractériser la nature de A (respectivement véridique, menteur, changeant commençant par dire la vérité, ou changeant commençant par dire un mensonge).

Vous participez à une première épreuve avec un orateur A qui fait les déclarations suivantes :

— J'aime le rouge mais pas le bleu.
— Soit j'aime le rouge, soit j'aime le vert.
— Si j'aime le rouge et le vert, alors j'aime le bleu.

Nous noterons R, V et B les variables propositionnelles associées au fait que l'orateur aime le rouge, le vert ou le bleu.

Nous noterons A_1, A_2 et A_3 les formules propositionnelles associées aux déclarations de A.

Question 1.3 : Représenter les déclarations de l'orateur sous la forme de formules du calcul des propositions A_1, A_2 et A_3 dépendant des variables R, V et B.

Question 1.4 : Appliquer les formules permettant de caractériser la nature des orateurs proposées pour la question 1.2 pour l'orateur A dépendant des variables A_1, A_2 et A_3 (ou directement de R, V et B).

Question 1.5 : En utilisant le calcul des propositions (résolution avec les tables de vérité, par exemple), déterminer la nature de l'orateur A. Quelles sont les couleurs qu'aime A ?

Vous participez à une seconde épreuve avec trois orateurs G, H et I. Vous avez déterminé dans la première phase avec succès que G est un menteur, que H est un véridique et que I est un changeant sans savoir s'il doit dire la vérité ou un mensonge pour sa déclaration suivante. Ceux-ci font les déclarations :

— I : Le losange est visible
— G : Le cercle n'est visible que si le losange est visible.
— I : Le triangle n'est pas visible.
— H : Soit le cercle est visible, soit le triangle est visible.

Nous noterons G_1, H_1, I_1 et I_2 les formules propositionnelles associées aux déclarations des orateurs dans cette épreuve. Nous noterons C, L et T les variables propositionnelles associées au fait que le cercle, le losange ou le triangle soit visible.

Question 1.6 : Représenter les déclarations des orateurs sous la forme de formules du calcul des propositions G_1, H_1, I_1 et I_2 dépendant des variables C, L et T.

Question 1.7 : Représenter les informations sur la nature des orateurs sous la forme d'une formule du calcul des propositions dépendant des variables G_1, H_1, I_1 et I_2.

Question 1.8 : En utilisant le calcul des propositions (résolution avec les formules de De Morgan, par exemple), déterminer quelle est (ou quelles sont) la (ou les) figure(s) visible(s) ainsi que la nature exacte de l'orateur changeant I.

n-SAT

En cours, le problème SAT a été présenté, ainsi que ses variantes n-SAT. Voyons ici comment transformer une clause en une conjonction de clauses équivalentes de taille au plus n, avec $n \geq 3$.

On considère une clause $C = l_1 \vee l_2 \vee \cdots \vee l_m$ et on suppose que $m > n$.

On pose $k = \lceil \frac{m-2}{n-2} \rceil$ (ramené à 1 si $m \leq 2$) et on va créer les clauses C_i pour i entre 1 et k (en pratique l'arrondi brutal fait qu'on peut avoir besoin d'une clause de moins), utilisant des variables γ_i (dont on suppose qu'aucune n'apparaît dans un littéral de C) pour i entre 1 et $k-1$, de sorte que :

- $C_1 = l_1 \vee l_2 \vee \cdots \vee l_{n-1} \vee \gamma_1$.
- Pour i entre 2 et $k-1$, $C_i = \neg\gamma_{i-1} \vee l_{n+(n-2)(i-1)} \vee \cdots \vee l_{n+(n-2)i-1} \vee \gamma_i$.
- $C_k = \neg\gamma_{k-1} \vee l_{n+(n-2)(k-1)} \vee \cdots \vee l_m$ (si on veut exactement n littéraux, on en répète un autant qu'il en faudra pour le *padding*).

Exercice 2 : Prouver que C est équivalente à $\exists\gamma_1, \ldots \exists\gamma_{k-1}, \bigwedge_{i=1}^{k} C_i$.

Exercice 3 : Prouver que la valeur de k fournie ci-avant est le nombre exact de n-clauses qui remplacent la clause C.

Remarquer que si $n = 2$, on aurait un problème.

Exercice 4 : En assimilant une variable propositionnelle à une chaîne de caractères quelconque, un littéral à un couple formé par un booléen et une variable propositionnelle (donc une chaîne) et une clause à une liste de littéraux, écrire en OCaml une fonction prenant en argument une clause et la taille maximale autorisée pour les clauses et retournant une liste de clauses dont la conjonction est équivalente à la clause en argument.

Dans le cas particulier de 2-SAT, il existe un algorithme en temps polynomial pour déterminer la satisfaisabilité d'une formule φ (alors qu'on rappelle que pour 3-SAT le problème est NP-complet).

Une proposition d'algorithme revient à construire ce que la littérature appelle le *graphe d'implication* de φ.

Concrètement, on va créer deux sommets par variable propositionnelle apparaissant dans φ (disons qu'il y a n noms de variables distincts au total), un pour le littéral égal à la variable, un pour le littéral égal à sa négation.

De plus, pour chaque clause faisant intervenir deux variables (distinctes ou non, et s'il n'était censé y avoir qu'un littéral on considère qu'il est répété une deuxième fois), on va créer deux arcs, à savoir pour une clause $l \vee l'$ on crée un arc partant du sommet associé au littéral équivalent à $\neg l$ vers le sommet associé au littéral l' et un arc partant du sommet associé au littéral équivalent à $\neg l'$ vers le sommet associé au littéral l.

Ainsi, il existe un arc depuis le sommet associé littéral l_1 vers le sommet associé au littéral l_2 si, et seulement si, une clause permet de déduire que l_1 implique l_2.

Dans ce cas, s'il existe un chemin depuis le sommet associé au littéral l_1 vers le sommet associé au littéral l_2, c'est que l_1 implique l_2 par transitivité de l'implication (on suit les arcs du chemin).

En particulier, s'il existe un chemin depuis le sommet associé à un littéral positif v_k vers le sommet associé au littéral $\neg v_k$, c'est que la variable v_k doit être fausse pour que la formule ait encore une chance d'être satisfaite.

On a un résultat analogue en déplaçant le symbole de négation.

Ceci permet d'énoncer la propriété suivante : une formule est satisfaisable si, et seulement si, il n'existe pas de variable propositionnelle v y apparaissant tel que, dans le graphe d'implication associé à la formule, on trouve un chemin du sommet associé à v vers le sommet associé à $\neg v$ et un chemin du sommet associé à $\neg v$ vers le sommet associé à v.

Exercice 5 : Prouver la réciproque (donc la condition suffisante de satisfaisabilité).

Remarque : En deuxième année, le calcul de composantes fortement connexes rendra la décision plus efficace et permettra même de trouver une interprétation qui satisfait la formule.

Exercice 6 : Créer en OCaml un type adapté pour des clauses de taille exactement 2 et pour les graphes d'implication.

Exercice 7 : Écrire une fonction prenant en argument une formule en FNC avec des clauses de taille exactement 2 et renvoyant le graphe d'implication associé.

Exercice 8 : Écrire une fonction prenant en argument un graphe d'implication et déterminant si la formule qui l'a produit est satisfaisable.

SAT et coloration

Dans la section précédente, l'intuition de la notion de réduction (dans un cas utile) avait été donnée.

Nous allons en voir une autre (moins utile) ici en construisant une formule en forme normale conjonctive qui est vraie si, et seulement si, un graphe non orienté donné est coloriable avec au plus k couleurs. Bien entendu, la formule dépend du graphe selon une méthode donnée ci-après.

Soient G un graphe non orienté, n son nombre de sommets (on assimile les sommets aux premiers entiers naturels), A l'ensemble de ses arêtes dont la taille est notée m et qui sont représentées comme des couples d'entiers (le premier étant conventionnellement strictement inférieur au deuxième) entre 0 et $n-1$ pour la cohérence. Soit aussi k un entier naturel précisant un nombre de couleurs disponibles, les couleurs étant les nombres de 0 à $k-1$.

On construit l'ensemble de variables propositionnelles $\{c_{i,j}, 0 \leq i < n, 0 \leq j < k\}$, sachant qu'on donnera pour signification à $c_{i,j}$ que « le sommet i est de couleur j ».

Alors le graphe G admet une coloration si, et seulement si, on peut attribuer à chaque sommet une et une seule couleur de sorte que deux sommets reliés soient de deux couleurs différentes.

Exercice 9 : Écrire une formule en forme normale conjonctive qui est vraie si, et seulement si, chaque sommet a au moins une couleur.

Exercice 10 : Écrire une formule en forme normale conjonctive qui est vraie si, et seulement si, chaque sommet a au plus une couleur.

Exercice 11 : Écrire une formule en forme normale conjonctive qui est vraie si, et seulement si, deux sommets reliés sont de couleurs différentes.

Exercice 12 : Écrire avec des types adaptés une fonction qui construit à partir d'un graphe et d'un entier naturel non nul une formule en forme normale conjonctive qui est satisfaisable si, et seulement si, le graphe admet une coloration avec au plus le nombre de couleurs précisé.

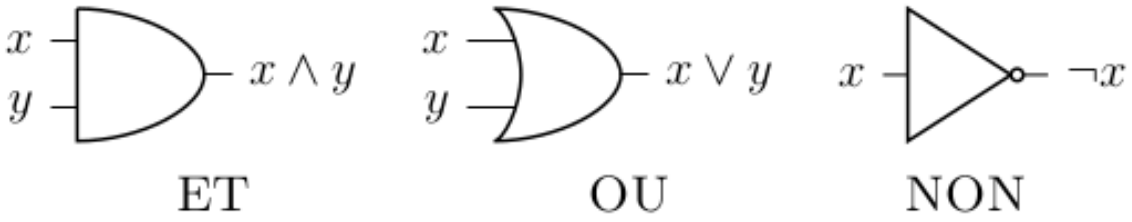
Par la suite, cette formule peut être résolue, et de manière constructive on peut en déduire une coloration une fois qu'une interprétation satisfaisant la formule est obtenue. Mais cette partie est laissée de côté.

Circuits logiques

Au début des années 2000, les circuits logiques étaient au programme de l'option informatique de MPSI. Entre temps, ils sont censés être abordés en physique en deuxième année.

Exercice 13 : Résoudre les questions à suivre, issues du concours ENS Ulm sélection internationale 2014.

Les portes logiques sont des éléments d'un circuit dont les entrées et les sorties sont binaires. Les trois portes logiques élémentaires ET, OU et NON sont décrites ci-dessous.



Dans les questions suivantes, indiquer clairement les entrées et les sorties dans tous les diagrammes de circuit. Éviter autant que possible d'utiliser des symboles supplémentaires et si c'est le cas, indiquer dans votre réponse ce qu'ils représentent.

Question 13.1 : En utilisant uniquement les portes ET, OU et NON, dessiner un diagramme de circuit pour un additionneur de 1 bit qui prend en entrée deux bits a_0, b_0 et (éventuellement) une retenue r et retourne la somme s_0 de a_0, b_0 et r et la retenue s_1, dont la formule mathématique est rappelée ci-dessous.

$$s_0 = (a_0 + b_0 + r) \bmod 2 \text{ et } s_1 = \lfloor (a_0 + b_0 + r)/2 \rfloor$$

Question 13.2 : En utilisant uniquement les portes ET, OU et NON et le circuit de la question précédente, dessiner un diagramme pour un additionneur de 3 bits qui prend en entrée a_2, b_2, a_1, b_1, a_0, b_0, r representant deux nombres de 3 bits a and b en binaire (notés $a_2a_1a_0$ et $b_2b_1b_0$), et un bit de retenue r. Il devra retourner la somme s de ces trois nombres en binaire sous la forme $s_3s_2s_1s_0$.

Question 13.3 : Expliquer comment étendre l'additionneur de la question précédente en un additionneur pour n bits et déterminer le nombre de portes nécessaires en fonction de n.

Question 13.4 : Supposons que le temps de propagation d'un signal logique à travers une porte élémentaire est de t secondes (indépendamment de la porte). Si tous les signaux d'entrée sont altérés pour l'additionneur pour n bits de la question précédente au temps 0, quel est le temps nécessaire pour l'obtention du résultat en sortie du circuit (en fonction de n et t) ?

Nous appelons un additionneur de 2^k bits « diviser pour régner » le circuit défini récursivement de la façon suivante :

- L'additionneur de 1 bit « diviser pour régner » est l'additionneur de 1 bit de la première question.
- Pour $k \geq 1$, l'additionneur de 2^k bits « diviser pour régner » est obtenu en fusionnant deux additionneurs de 2^{k-1} bits « diviser pour régner » A et B en envoyant les 2^{k-1} bits de poids faible des entrées a et b et l'éventuelle retenue r à A et tous les bits restants à B. La retenue de la sortie de A est envoyée à B.

Question 13.5 : Déduire le délai pour l'obtention du résultat pour ce circuit en fonction de $n = 2^k$ et de t.

(La fin du sujet est disponible sur internet et présente une optimisation.)