Avant la MP2I - Informatique (2024)

Document encore en écriture ! Rejoignez le discord (https://discord.gg/wrAx8B96Jy)pour signaler les fautes d'orthographe (elles doivent être nombreuses), les fautes dans les exercices, etc. Si vous n'avez pas discord, un formulaire est disponible sur le site de ce document.

Objectifs Ce document a été rédigé par un élève de classe MPI (désormais à Ulm) et non un professeur, il ne contient pas un "must-have" avant de rentrer en prépa. Il est en fait une compilation de points de cours et d'exercices classiques que j'aurais voulu avoir vu avant de rentrer en classe préparatoire. Le meilleur moyen d'utiliser ce document est de lire un chapitre par jour, de bien comprendre les points de cours et les notions présentées et ensuite d'attaquer les exercices. Si vous ne trouvez pas la solution à certains exercices, pas de panique : c'est normal. C'est la principale différence avec la terminale, le but n'est pas de trouver mais de chercher. En cherchant un exercice vous mobilisez des connaissances (tiens ça ressemble à tel théorème / telle propriété, oh et si j'essayais de faire ça ...) et peut-être que ça ne suffira pas pour résoudre l'exercice, mais vous aurez appris bien plus qu'en ayant trivialisé *(vous entendrez souvent ce mot)* un exercice d'application simple.

Sources La plupart des exercices présentés sont des exercices auxquels j'ai pensé en écrivant le cours ou que l'on m'a envoyé (je remercie Wyrdix pour certains exercices dans la partie `Exercices sans thèmes précis`). Certains peuvent aussi venir de mon parcours en classe préparatoire, je n'ai pas forcément leur source précise mais les exercices viendront souvent de :

- Algorithms (J. Erickson)
- Leetcode (site de programmation compétitive)

Les différents thèmes abordés Ce document est volontairement léger en cours pour ne pas vous noyer pendant les vacances, il faut principalement vous reposer. Cependant, quelques exercices ne peuvent pas faire de mal et quelques définitions non plus (juste pour s'habituer aux notations de prépa). Voici la liste des points abordés dans l'ordre:

1. Qu'est-ce qu'un algorithme
2. Récursivité
3. Tableaux ou listes ?
4. Représentation des réels
5. Raisonner inductivement
6. Retour sur trace
7. Introduction aux graphes
8. Travailler avec des mots
9. Comment débuguer
10. Exercices sans thème précis
11. Corrections de certains exercices

Gardez ce polycopié à jour en téléchargeant régulièrement la dernière version sur le site https://avantlampii.cr-dev.io/ ou directement depuis le github (https://github.com/crdevio/Livres)

Contribuez vous pouvez utiliser le repo GitHub pour proposer vos Pull Requests (exercices, corrections, typos, etc).

Nouveautés de la version 2024

- Ajouter de certaines corrections
- Ajout d'exercices
- Plus de margin-notes pour centrer le texte
- Ajout du cours sur le retour sur trace
- Changements mineurs de style

I - Qu'est-ce qu'un algorithme ?

I.1 - Définition

Vous avez sans doute souvent entendu parler d'algorithme, que ce soit en NSI ou lors de vos cours de mathématiques au lycée, mais qu'est-ce que c'est exactement ? On serait tenté de dire qu'un programme Python est un algorithme… Mais non ! Ce serait plutôt une implémentation d'un algorithme. Pour le comprendre, voici la définition formelle d'un algorithme :

Définition 1-0 — Algorithme

Un algorithme est une suite d'instructions précises réalisant une tâche.

Remarque: Instruction "précise" signifie qu'on ne doit pas douter, elle doit être claire ! Un ordinateur exécute le code de manière séquentielle *en général*, il ne doit pas y avoir d'ambiguïté sur la tâche à exécuter.

Cette définition peut surprendre si on ne l'a jamais vue : pas de notion de programmation et pas de notion d'ordinateur (trier votre main au Uno est un algorithme selon cette définition). C'est tout à fait normal puisqu'à l'époque où ce mot a été prononcé pour la première fois (IXème siècle) il n'y avait pas d'ordinateur pour implémenter les algorithmes, cependant on possède désormais cette définition :

Définition 1-0 — Implémentation d'un algorithme

On appelle implémentation d'un algorithme son écriture dans un langage de programmation.

Cette définition correspond donc à ce que l'on vous a probablement qualifié d'algorithme au lycée. Vous pouvez maintenant comprendre, par exemple, que l'**algorithme du tri fusion** est une suite d'instructions permettant de trier une liste et que votre fichier `tri.py` est son **implémentation en python**.

I.2 - Exemple d'algorithme

Pour le premier exemple d'algorithme, on va considérer le plus classique (que vous avez probablement vu en Maths Exp.) : l'algorithme d'Euclide. Voici un rappel du principe *c'est volontairement confus pour introduire l'intérêt d'avoir des instructions claires* :

Pour deux nombres `a` $\geq$ `b`, si `b` est nul on renvoie `a` sinon on rappelle l'algorithme avec `a % b` et `b`.

La preuve mathématiques est sans intérêt ici (vous la ferez sûrement en début de première année) donc on va admettre que l'algorithme fonctionne. Je pense que vous avez pu le remarquer : cette notation pour l'algorithme n'est pas très pratique, si vous ne voyez pas où est le problème, voici une version plus propre (plus proche d'une suite d'instructions) de cet algorithme :

```
PGCD:
    Entrée : a >= b
    Si b>a alors renvoyer PGCD(b,a)
    Sinon Si b=0 renvoyer a
    Sinon renvoyer pgcd(a % b, b)
```

Remarque: Cet algorithme s'appelle lui-même, si vous l'avez déjà vu en NSI vous pouvez continuer à lire cette section, sinon je vous conseille de passer à l'introduction à la récursivité.

Maintenant que vous avez vu les deux versions, je suppose que si je vous demandais de me coder en python l'algorithme du PGCD, vous préféreriez avoir la deuxième version plutôt que la première comme indication : elle est plus claire. Cette définition est **informatiquement correcte mais elle n'a aucun lien avec votre ordinateur**, en fait je peux écrire le même genre de programme pour mon réveil et ça restera de l'informatique :

```
Reveil:
    Entrée : cours1
    Si cours1 != SI alors se_lever()
    Sinon dormir()
```

Pour en revenir à l'algorithme d'Euclide, la suite d'instructions définie est donc bien un **algorithme**, en revanche, le code suivant en est une implémentation en OCaml (**n'apprenez pas le OCaml pour le moment** ! Si vous souhaitez vous initier au OCaml, ce document contient dans les dernières pages un lien vers une super série de vidéos).

Code — Euclide en OCaml

```
let rec pgcd a b =
  if b>a then pgcd b a
  else begin
        if b=0 then a
        else pgcd (a mod b) b
  end;;
```

Exercice (1-1) — Maximum — 1⋆

1. Écrire un algorithme qui prend en entrée un ensemble fini de valeurs (un objet similaire à une liste python) et qui permet d'en obtenir le plus grand élément.
2. L'implémenter dans le langage de votre choix.

Exercice (1-2) — Croissance — 1⋆

1. Écrire un algorithme qui prend en entrée une suite finie de nombres (un objet similaire à une liste python) et qui vérifie si elle est de nature croissante.
2. Modifier votre algorithme pour qu'il permette de dire si la suite est croissante, décroissante, constante ou qu'elle ne possède aucune de ces propriétés.

Exercice (1-3) / Euclide borné / 2⋆

1. Écrire un algorithme (séquence d'instructions) qui prend en entrée trois entiers `a`, `b` et `t` et renvoie Vrai si l'algorithme d'Euclide termine en moins de `t` appels à l'algorithme (en accord avec l'algorithme récursif ci-dessus) et Faux sinon.

Remarque: Il n'est pas possible de juste réaliser des appels à l'algorithme précédemment écrit, il faut le redéfinir et modifier son comportement

ENS ULM (1-4) / Chevaux / 4⋆

Vous avez 25 chevaux et vous voulez les 3 plus rapides, mais vous n'avez le droit qu'à 7 courses d'au plus 5 chevaux : comment faire ?

Remarque: (INDICATION) Il faut des courses d'exactement 5 chevaux.

I.3 - Comparer les algorithmes

Pour comparer deux algorithmes, on peut envisager plusieurs pistes. Une méthode naïve serait de lancer les deux algorithmes sur une même machine et comparer le temps de calcul. Cette méthode pose plusieurs problèmes :

- Qui nous dit qu'un l'algorithme n'est pas plus rapide qu'un autre à cause d'une spécificité du système ?
- On ne teste ici le programme que sur une seule entrée: rien ne nous assure que le résultat serait toujours valable pour d'autres entrées.

Il y a encore d'autres problèmes qui peuvent intervenir mais les deux cités suffisent à comprendre que ce n'est pas la piste à suivre.

Avant d'analyser un algorithme (en donner la **complexité**), il faut savoir en fonction de quoi l'analyser. Par exemple, pour un algorithme de tri, l'analyser en fonction du plus grand élément passé en entrée n'a pas réellement de sens (puisqu'on suppose que la comparaison entre deux entiers est, à une constante près, la même indépendamment de la taille). En revanche, l'étudier selon la taille de la liste passée en entrée semble plus intéressant. On note désormais n la taille de l'entrée.

Maintenant que l'on sait selon quel paramètre nous voulons étudier notre algorithme, il faut savoir ce que l'on veut mesurer. Notre but est d'obtenir une **approximation asymptotique**, ce qui (dans un langage plus raisonnable) revient à expliquer à quel point la complexité croît en fonction de l'entrée. Par exemple, pour une fonction qui parcourt deux fois une liste, on aura une croissance linéaire ($2 \times n$). Ainsi, on notera $\theta(n)$, ne vous tracassez pas pour la notation θ, il s'agit d'une des **notations de Landau**, elles sont au programme de NSI mais vous seront clairement rappellés en prépa. Retenez simplement que pour exprimer que **la croissance de l'algorithme, notée C_n,est "à une constante près", une fonction $g(n)$**, on note $C_n = \theta(g(n))$.

Vous vous dîtes sûrement que cette notation n'a pas d'intérêt car écrire $2n$ est bien plus précis. Vous auriez raison s'il n'y avait pas de constantes ignorées dans ce calcul. Pour comprendre ce que je viens de dire, voici un exemple :

CODE NbJoursDeNoel

NbJoursDeNoel(cadeaux[2...n]):

```
for i=1 to n:
  Chanter “ Pour le i ème jour de Noël, j'ai reçu ”
  for j=i downto 2
    Chanter "j cadeaux[j]"
  if i>1
    Chanter "et"
  Chanter "un oréo"
```

L'algorithme prend pour entrée une liste de $n-1$ cadeaux dans un tableau. On va donc exprimer la complexité selon cette taille n. On peut se rendre compte qu'on a une complexité $\theta(n^2)$ à la main: On fait une boucle principale de 1 à n et à chaque appel on a une boucle de taille i.

On pourraît se dire qu'un calcul mathématique nous permettrait de connaître la complexité précise de la fonction. Mais pour l'obtenir, il faudrait connaître la complexité de `Chanter`, or on sait juste qu'elle est constante, ce qu'on peut noter $\theta(1)$. Il y a dès ce moment une approximation, aussi précis que l'on veuille être, cette approximation sera toujours la, il faut donc uniquement chercher un autre de grandeur car le $\theta(1)$ de `Chanter` ne dépend pas de n et n'affectera donc que par des constantes l'évolution de la complexité en fonction de la valeur n passée en entrée.

I.3.a - À QUEL ORDRE REGARDER ?

Avant toute chose, on s'intéresse aux **pires cas** des algorithmes, c'est-à-dire les entrées les plus défavorables, ce qui permet de dire “je suis sûr que pour toute entrée, la complexité ne peut pas dépasser cet ordre de grandeur”. Si on prend le meilleur cas, l'algorithme consistant à prendre une liste en entrée et attendre (boucle `while` ne faisant rien dans le cas non trié) qu'elle se trie toute seule est un algorithme de tri linéaire puisque le meilleur cas (liste triée) est de complexité linéaire (pour savoir si elle est triée). Il n'y a donc aucun intérêt à prendre le meilleur cas.

Voici deux algortihmes :

```
Algo1:
    Entrée : une matrice de taille n*n symétrique
    Pour i allant de 0 à (n-1)
        Pour j allant de i à (n-1)
            faire_truc(i,j)
```

```
Algo2:
    Entrée : une matrice de taille n*n symétrique
    Pour i allant de 0 à (n-1)
        Pour j allant de 0 à (n-1)
            faire_truc(i,j)
```

On pourraît se dire que le premier est plus optimisé car il va parcourir moins de cases, il ne va parcourir que la partie supérieure de la matrice. Or, si on fait une analyse fine, on aura une complexité (on suppose que `faire_truc` a un coût de 1) $1+2+3+\ldots+n$ et cette somme vaut $\frac{n(n+1)}{2}$ (au programme de première spé maths), on a donc quelque chose “de l'ordre de n^2”, c'est-à-dire qu'à un terme négligeable (tracer les fonctions $x \to x^2$ et $x \to x$ sur votre calculatrice ou sur géogebra) près, on a un algorithme de l'ordre de n^2. C'est le même ordre que pour l'algorithme 2 qui lui va pourtant parcourir toutes les cases.

En classe prépa (et souvent ailleurs, sauf si on est vraiment sûr de l'optimisation à une ligne près, ce qui est rare) on ne cherche qu'à avoir des complexités "à un ordre près" (ce sera défini plus précisément en maths, le but ici est juste d'avoir une idée) donc ce qui va compter va être une approximation. On ne va pas s'amuser à dire qu'une boucle qui effectue 6 opérations pour chacune des n entrées est en complexité $6n$, on va dire qu'elle est de l'ordre de n (noté $\theta(n)$, le θ aspire les constantes multiplicatives).

Depuis le début on utilise des θ car on prend des exemples simples et qu'on peut à la fois minorer et majorer par des constantes multiplicatives les termes asymptotiques qui nous intéressent. En prépa (de mon expérience personnelle), vous manipulerez beaucoup plus les O qui sont uniquement des majorations (ie "cet algorithme ne peut pas être pire que celui-ci"), on va donc utiliser des O à partir de maintenant en retenant qu'il suffit de majorer (ie écrire que $C_n \leq k \times g(n)$ avec k une constante)

C'est en réalité très pratique, il devient beaucoup plus simple de donner des complexités à l'oeil nu :

```
Additionner_matrice:
    Entrée : M une matrice de taille n*n
    resultat <- 0
    pour i allant de 0 à (n-1)
        pour j allant de 0 à (n-1)
            resultat <- résultat + M[i][j]
    renvoyer resultat
```

On a deux boucles `for` (pour) imbriqués, chacune a n éléments et la deuxième boucle exécute une opération $O(1)$ (une somme, un accès à `résultat` et une affectation), donc $O(n^2)$. Et **c'est tout** ! Si vous avez compris ça, vous savez donner la complexité de 50% des algorithmes que vous verrez durant vos années en prépa (évidemment il y aura des subtilités, notamment avec la récursivité, mais ceci est l'idée principale à avoir en tête).

Exercice (1-5) / Complexité du produit matriciel / 2⋆

1. Ecrire un algorithme qui fait le produit de deux matrices. (on supposera les conditions nécessaires remplies)
2. Donnez-en la complexité.

Exercice (1-6) / Mais qui a inventé ça ? / 1⋆

1. Soit l'algorithme suivant :

```
CoffeeSearch:
    Entrée: L une liste d'entiers de taille n, x un entier
    Pour e dans L:
        Si e == x, alors renvoyer Vrai
        Sinon faire_un_cafe()
    renvoyer Faux
```

Sachant que l'opération `faire_un_cafe` prend environ 100000 étapes, quelle est la complexité (à un ordre près) de l'algorithme `CoffeeSearch` ?

Aller plus loin (Autres notions de complexité)

Retiré après relecture.

Exercice (1-7) Complexité du tri par insertion 1⋆

1. Donner un algorithme pour le tri par insertion.
2. L'implémenter dans le langage de votre choix.
3. Donnez un ordre de grandeur de la complexité

Exercice (1-8) Carmin-sur-mer 2⋆

Voici l'arène de Carmin-sur-mer. Il y a 3 rangées de 5 poubelles devant vous. Deux d'entre elles contiennent un interrupteur, on sait qu'elles sont adjacentes. Tant qu'aucun interrupteur n'a été trouvé, ils ne changent pas de position. Dès qu'un d'entre eux est trouvé, vous avez un essai. Si vous trouvez les deux interrupteurs, vous gagnez, sinon les interrupteurs changent aléatoirement de place et il faut recommencer.

1. Proposez un algorithme naïf pour trouver une solution à l'arène (celle qu'un enfant de 8 ans ferait).
2. Proposez une optimisation à l'aide d'un tableau de poubelles "déjà vus".
3. Sachant que si vous vous ratez plus de 100 fois, vous êtes assurés d'avoir bon à la 101ème tentative, quelle est la complexité de votre premier algorithme ? De la version optimisée ?

Exercice (1-9) Inverser un tableau 1⋆

1. Soit un tableau T de taille n, proposez un algorithme permettant d'obtenir T' le tableau inverse de T (le premier élément de T' est le dernier de T etc).

Remarque: Cet exercice est simple car on manipule un tableau, ie vous pouvez faire $T[i]$ pour récupérer le i-ème élément, dans la section sur la récursivité on verra que pour une liste, c'est une autre histoire

Exercice (1-10) / Nombres premiers / 3⋆

1. Donnez un algorithme qui permet de trouver les nombres premiers $\leq n$ (si vous ne l'avez pas vu, cherchez sur internet : Crible d'Ératosthène)
2. Quelle est la complexité de cet algorithme ?
3. Est-il possible, de manière simple, de savoir si un nombre est premier ou non en complexité linéaire (c'est-à-dire en $O(n)$).

Remarque: AKS n'est pas considéré simple !

Exercice (1-11) / Sudoku, première rencontre / 2⋆

Soit une grille de Sudoku qui possède des carrés de n cases (sous forme de matrice, si vous ne savez pas comment ça marche, pas de panique, regardez la section sur les tableaux et les listes)

1. Donnez un algorithme qui étant donné une configuration (la grille est remplie avec certaines valeurs) renvoie Vrai si la partie est finie (indépendamment de si la solution est correcte ou non) et Faux sinon
2. Donnez un algorithme qui étant donné une configuration renvoie Vrai si la grille est correcte (si elle ne viole aucune règle du Sudoku) et Faux sinon
3. Donnez la complexité de chacun de vos algorithmes.

Exercice (1-12) / Bruteforcing pour les nuls / 1⋆

Vous venez de voler un PC à un inconnu dans un bus (ce n'est pas très gentil), vous courrez vous enfermer chez vous et vous l'allumez : il y a un mot de passe !

1. Si il s'agit des codes PIN de windows Hello (4 chiffres), combien d'essais vous faudrait-il pour bruteforce le mot de passe ? Ecrire l'algorithme.
2. Par manque de chance, vous avez bloqué Windows Hello : pas grave, il vous reste son mot de passe classique. Heureusement pour vous, il a laissé une note "longueur = 14". Sachant que son mot de passe n'utilise que des lettres de l'alphabet, combien de mot de passe faut-il tester ? Quelle serait la complexité d'un algorithme de bruteforce ?
3. Si vous savez que son mot de passe commence par un P, de combien pouvez-vous réduire la complexité ?

Exercice (1-13) / L'unique / 4⋆

Ecrire un algorithme qui, étant donné une liste d'entiers telle que chacun des éléments dans la liste y soit exactement 2 fois sauf un unique élément, renvoie l'unique élément n'apparaissant qu'une seule fois. *Vous ne devrez parcourir votre liste qu'une seule fois, interdit de revenir en arrière, interdit d'utiliser une mémoire externe autre que $O(1)$, cf le paragraphe Complexité Spatiale dans le chapitre récursivité si vous ne l'avez pas vu en NSI*

II - Récursivité

II.1 - Définition

Commençons par définir une fonction récursive, nous rentrerons dans les détails juste après :

Définition 2-0 / Fonction récursive

Une fonction est dite récursive si elle s'appelle elle-même dans son corps d'instructions

Par exemple, la fonction suivante est récursive :

```
def fact(n): #on suppose n>=0
    if n==0: 1
    return n * fact(n-1)
```

Elle est strictement équivalente à cette fonction :

```
def fact(n): #on suppose n>=0
    res = 1
    for i in range(1,n+1):
        res*= i
    return res
```

mais elle est plus simple à lire. La récursivité n'est évidemment pas qu'une simplification du code et a d'autres intérêts, mais pour commencer il est bien de comprendre qu'une fonction récursive peut en partie servir à avoir un code plus lisible.

Exercice (2-1) / Une première mise sous forme récursive / 1⋆

Soit la fonction suivante :

```
def mystere(n):
    res = 1
    for i in range(2,n+1,2):
        res*=i
    return res
```

1. Que fait cette fonction ?
2. L'écrire sous forme récursive
3. Soit `mystere2` la fonction où le `range` est de la forme `range(1,n+1,2)`. Comment calculer `mystere` à partir de `fact` et `mystere2` ?
4. Quelle est la complexité de vos fonctions

Exercice (2-2) / Un produit / 1⋆

Proposez une fonction qui prend en entrée deux entiers `a` et `b` et qui calcule le produit `a` $\times$ `b` sans avoir le droit d'utiliser l'opération $\times$. Le réécrire en récursif si ce n'était pas déjà fait.

II.2 - Précautions

Dans le code de `fact`, il y a le cas `if n==0` qui est mis en première ligne, on appelle ce genre de cas les **cas de base**. Ils sont primordiaux car sans eux votre programme tournerait à l'infini. Par exemple,

si on passe `n=-1` à la fonction `fact`, elle produira une erreur. Les cas de base changent d'un problème à l'autre, il faut toujours faire attention à ne pas les oublier. De même, il faut faire attention à pouvoir garantir un **variant** sur votre fonction récursive qui permet de garantir qu'elle atteindra un cas de base. Par exemple, dans la fonction `fact`, on peut garantir que `n` est strictement décroissant à chaque appel et donc atteindra 0, ie le cas de base. Voici quelques exercices :

Exercice (2-3) / Identifier un variant / 1⋆

1. Ecrire un algorithme récursif calculant la suite de Fibonacci
2. Donner les cas de base d'un algorithme calculant la suite de Fibonacci

(⋆)3. Quel est le variant associé ?

Exercice (2-4) / Un peu de proba... / 3⋆

Soit l'algorithme suivant :

```
Proba:
    Entrée: n et k deux entiers
    if n==0:
        renvoyer k
    res<- lancer_piece()
    if res == Pile:
        renvoyer proba(n-1,k+1)
    else:
        renvoyer proba(n,k+1)
```

Exemple d'utilisation : `proba(n,0)`

1. Que fait cet algorithme ?
2. Est-ce une loi de Bernoulli ? Binomiale ?
3. Quelle est la probabilité qu'un appel baisse la valeur de n ?
4. Avec quelle probabilité la valeur de n ne décroît pas en p appels ?
5. Proposez une estimation de la complexité
6. Proposez une application à cet algorithme.

II.3 - Comment fonctionne la récursivité ⋆

À chaque appel, un "bloc d'activation" est ajouté dans votre ordinateur, plus il y en a, plus ils s'**empilent**. Le terme d'empiler est important : Une pile est une structure de données où vous n'avez accès qu'à l'élément le plus au-dessus de la pile (comme une pile d'assiette) ie le dernier élément inséré. En fait c'est logique de n'avoir accès qu'au dernier appel : quand une fonction s'appelle elle-même de manière récursive, on ne pourra y accéder qu'une fois tous les appels postérieurs effectués. Ainsi quand votre ordinateur lit :

```
def f(n):
    if n==0: return
    f(n-1)
    truc(n)
```

Il fait :

```
f(5) empilé
f(4) empilé
f(3) empilé
f(2) empilé
f(1) empilé
f(0) dépilé
truc(1)
f(1) dépilé
truc(2)
f(2) dépilé
...
```

Exercice (2-5) — Blocs d'activation — 1⋆

1. Donnez l'enchaînement décrivant `fact(5)`.
2. On propose l'algorithme suivant pour la suite de fibonacci :

```
fibo:
    Entrée:  n
    Si n==0 renvoyer 0
    Sinon Si n==1 renvoyer 1
    Sinon renvoyer fibo(n-1)+fibo(n-2)
```

Donnez l'enchaînement décrivant `fibo(5)`

3. Pouvez-vous le faire pour `fibo(10)` ?
4. Codez cette fonction dans le langage de votre choix, pour quelle valeur de n le temps d'attente devient trop long ?

II.4 - Complexité

II.4.a - Temporelle

Je ne vais pas rentrer dans les détails pour la complexité des fonctions récursives, vous le verrez dans le début de votre première année, le but ici est simplement de vous faire comprendre, avec les mains, quel algorithme sera infaisable et lequel sera faisable. Si vous avez fait l'exercice 2-5, vous avez remarqué que pour une valeur de `n` assez faible, le programme devient impossible à lancer, mais pourquoi ? Si on note `C(n)` la complexité pour un appel avec la taille `n`, on peut voir la relation de récurrence suivante : `C(n) = C(n-1) + C(n-2) + 1` (le 1 est à peu près le coût de l'addition en tant qu'opération élémentaire). Ainsi, on peut voir que la fonction va calculer plein de fois les mêmes valeurs, sans rentrer dans les détails de la résolution, on trouve que la complexité vaut $C(n) = \varphi^n$ avec φ un nombre > 1, ainsi la complexité $C(n)$ va exploser, ce qu'on appelle une complexité **exponentielle**.

À l'inverse, si on considère la fonction `fact`, on a une formule de la forme `C(n) = C(n-1) +1` et la, vous connaissez ! Si on réécrit ceci $u_n = u_{n-1} + 1$ c'est une suite arithmétique, on trouve trivialement que $u_n = n$ donc $C(n) = n$, c'est une complexité linéaire donc tout à fait acceptable.

Pour aller un peu plus loin, on peut voir à la main que `fibo` va être bien plus lourde que `fact` en imaginant l'arbre des appels, on se rend compte que `fibo` va avoir, pour chaque appels, deux sous-branches (l'appel à $n-1$ et celui à $n-2$). Chaque sous-branche va elle-même en avoir deux et ainsi de suite, on appelle

ceci un arbre binaire, vous le verrez en première année, visuellement cet arbre est bien plus lourd que les appels de `fact` qui ne font qu'une simple ligne.

II.4.b - Spatiale

Une autre notion importante est la complexité spatiale. En réalité, avec nos ordinateurs modernes, elle n'a la plupart du temps peu d'importance. Cependant elle peut-être utile dans des contextes précis (faire décoller une fusée entre autres) et peut aussi nous provoquer quelques erreurs embêtants, dont les fameux StackOverflow, qui sont un dépassement de l'espace alloué aux fonctions récursives (la pile où on stocke les blocs d'activation).

Comment évaluer la complexité spatiale ? Avec des langages comme Python, ce n'est pas toujours très facile : est-ce qu'un tableau a une taille bien adaptée, ou est-ce qu'il a été sur-évalué ? sous-évalué ? Bref, le fait que Python veuille nous simplifier la vie a des contre-parties, dont celui d'avoir plus de mal à identifier la complexité spatiale. On préfère alors les **langages typés**, ça tombe bien C et OCaml le sont ! C'est-à-dire que quand vous déclarez une variable, il faut en préciser le type. Ainsi, vous ne pouvez pas déclarer un tableau d'entiers et y stocker Harry Potter volume 1 (en Python, vous pouvez).

On va travailler en C dans la suite, pas de panique vous n'avez pas à savoir le manipuler pour le moment, c'est simplement pour être précis.

Prenons un premier exemple : on veut stocker `n` entiers. La commande C correspondante est :

```
int* tableau = (int*)(malloc(sizeof(int)*n));
```

Ce n'est pas la syntaxe la plus simple, mais c'est la plus parlante. Sans rentrer dans les détails, on a `sizeof(int)` qui renvoie la taille d'un entier dans le système qu'on utilise et la multiplication par `n` qui permet d'être sûr qu'on a un `O(n)` comme complexité en espace. On se rend donc compte que le C est certes un peu plus lourd en syntaxe, mais est mille fois plus précis que les instructions Python `tab = []` et `tab.append(truc)`.

Pour les fonctions récursives, si vous n'êtes pas en récursivité terminale (je ne rentre pas dans ces détails, vous verrez ceci en cours de OCaml), vous pouvez considérer que `n` appels récursifs à une fonction qui a une complexité spatiale $O(l)$ est en complexité spatiale $O(l \times n)$. C'est logique : les blocs d'activations sont ajoutés à la pile d'exécution donc dans le pire cas, on a ajouté tous les blocs sans en retirer on a donc une pile de n blocs de hauteur de $O(l)$ donc bien du $O(l \times n)$. Ainsi, quand on demande des fonctions récursives en complexité spatiale $O(1)$, c'est que chaque appel doit être en $O(1)$ et que vous n'avez pas le droit de faire des opérations mémoires avant d'appeller la fonction (ie pas de tableau déclaré avant l'appel)

Exercice (2-6) / Rendre fibonacci linéaire⋆ / 1⋆

1. Proposez une manière de rendre l'algorithme de fibonacci en complexité linéaire (toujours en récursif !) à l'aide de la "mémoïsation" (si vous n'avez pas eu NSI, la correction de cette question est disponible juste en-dessous)

(⋆ ⋆ ⋆)2. Faîtes de meme, mais en complexité spatiale $O(1)$ (donc $O(k)$ pour k appels récursifs)

Correction de la question 1 (exemple de mémoïsation)

```python
def fibo(n):
    tableau = [-1 for i in range(n)]
    tableau[0] = 0
    tableau[1] = 1
    def aux(k):
        if tableau[k] != -1: return tableau[k]
        a1,a2 = aux(k-1),aux(k-2)
        tableau[k] = a1+a2
        return a2
    aux(n)
```

Exercice (2-7) | **Complexité spatiale** | **1⋆**

Donnez la complexité spatiale des fonctions suivantes :

1.

```python
def coucou(n):
    print("coucou")
    if n != 0 : return coucou(n-1)
```

Remarque: Si vous connaissez déjà la récursivité terminale, n'en tenez pas compte dans votre réponse

2.

```python
def useless(n):
    t = []
    while(True):
        t.append(1)
```

3.

```python
def doublon(n,tab):
    k = chercher_val_max(n) #on suppose O(1) espace
    stockage = [false for i in range(k+1)]
    for i in range(k):
        if stockage[tab[i]]: return True
        stockage[tab[i]] = True
    return False
```

(⋆)4.

```c
char* cc = (char*)(malloc(sizeof(char)*2));
cc[0] = 'c';
cc[1] = 'c';
```

(⋆) 5.

```
let init n =
    Array.make_array n False;;
```

Exercice (2-8) Un tri 2⋆

Voici un algorithme de tri:

```
Tri:
    Entrée: T tableau d'entiers de taille n
    Si n==1: renvoyer ()
    Sinon:
        Tri(T[0:n/2],n/2)
        Tri(T[n/2:n],n/2)
        Fusionner(0,n,n/2)
```

En admettant que `Fusionner` est de complexité à peu près n, donnez une relation de récurrence décrivant la complexité de `Tri`. On admettra qu'elle se résoud en $n\log(n)$.

Exercice (2-9) Un beau sapin 1⋆

1. Décrire un algorithme récursif qui affiche un sapin de Noël, ie quelque chose de cette forme :

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

(⋆)2. Essayez d'en afficher 2 côte à côte

Exercice (2-10) Palindrome 1⋆

On suppose que l'on possède un type `parchemin` tel que on ait accès à trois informations : la première lettre du parchemin (`p.premiere`), la dernière lettre du parchemin (`p.derniere`) et le parchemin correspondant au mot entre les deux. Si le parchemin ne contient qu'une seule lettre, `p.derniere` prend une valeur spéciale appellée `NULL`.

Par exemple, abc est codé a:(b:NULL):c

1. Faites un dessin montrant comment les mots `coucou` et `bobob` sont encodés par ce parchemin.
2. Ecrire un algorithme qui prend en entrée un parchemin et renvoie `True` s'il code un palindrome et `False` sinon

Exercice (2-11) Somme des chiffres 2⋆

1. Ecrire un programme qui prend en entrée un entier $n \in \mathbb{N}$ et renvoie la somme de ses chiffres, par exemple `somme(1234)` $= 1+2+3+4 = 10$.
2. Ecrire un programme qui prend en entrée un entier n et renvoie la somme des chiffres des nombres ≤ 10000 dont la somme des chiffres est inférieure ou égale à n

Exercice (2-12) / Compte à rebours / 2⋆

Ecrire un programme qui, étant donné un entier $n \in \mathbb{N}$, affiche `0 1 ... n (n-1) ... 0 1 .....` une infinité de fois

Exercice (2-13) / Maximum d'une matrice / 2⋆

Ecrire un programme qui, étant donné une matrice de dimension $(n, p) \in \mathbb{N}^2$, renvoie la liste des plus grands éléments sur chaque ligne. (de manière récursive)

Exercice (2-14) / Permutations / 3⋆

(Tombé à ULM mais franchement c'était du vol) Ecrire un programme qui, étant donné un entier $n \in \mathbb{N}$, renvoie toutes les permutations de $[\![1; n]\!]$.

Remarque: On procédera récursivement. On peut le voir de deux manières, une méthode "simple" et une jolie méthode avec un arbre d'arité n

Exercice (2-15) / Recherche dichotomique / 3⋆

On se donne un tableau trié par ordre croissant, ie pour tout $i < j$, `tab[i]` $\leq$ `tab[j]`.

1. Donnez un algorithme naïf (en complexité linéaire) pour trouver un élément x dans un tel tableau
2. Proposez un algorithme qui exploite le côté croissant
3. Complexité ?

Exercice (2-16) / Jeu d'élimination / 2⋆

On vous donne une **liste** qui contient $[\![1; n]\!]$

On applique l'algorithme suivant :

- On retire le premier élément à gauche puis tous les éléments qui sont sur des indices accessibles en faisant des pas de 2 depuis l'élément retiré (par exemple si on a `1 2 3 4`, on retire `1 3` et il reste `2 4`).
- On retire le dernier élément (donc le plus à droite) et on fait de même (avec le même exemple il resterait `1 3`)
- On répète en alternant tant qu'il ne reste pas qu'un seul élément

1. Donnez la réponse pour $n = 9$
2. Codez un algorithme qui prend un entier n et renvoie le dernier élément restant.

Exercice (2-17) / Puissance / 2⋆

Ecrire un algorithme récursif qui renvoie x^n pour $x \in \mathbb{R}$ et $n \in \mathbb{Z}$.

Exercice (2-18) / ATOI / 3⋆

En C, il y a une fonction très pratique qui prend en entrée une chaîne de caractère et renvoie le nombre associé, par exemple `atoi("125") = 125`.

Implémentez `atoi` dans le langage de votre choix.

II.5 - Note de fin

Actuellement vous pouvez encore vous demander "mais pourquoi on utilise la récursivité alors qu'un algorithme itératif suffit", la réponse sera immédiate quand vous aurez lu la partie sur les raisonnements inductifs. En attendant, essayez avant de passer à la section suivante d'avoir bien compris comment "penser récursif" : cas de base, relation de récurrence, variant, programmation.

III - Tableaux ou listes ?

Depuis le début de ce document, je varie entre liste et tableau sans vraiment faire de différence : c'est fini. Un tableau et une liste sont deux structures totalement différentes en informatique.

III.1 - Tableau

Définition 3-0 / Tableau

Un tableau est une structure de donnée qui peut stocker un nombre fini, précisé au préalable, d'éléments d'un type (souvent précisé au préalable aussi, mais ça dépend de l'implémentation)

Ainsi, on peut voir un tableau comme une structure à deux paramètres : `t.n` sa taille (`len` en Python) et `t.content` son contenu. Par exemple, `t[i]` signifie la `i`ème case de `t.content`. Le fait que la taille soit précisée au préalable peut être perturbant au début, surtout quand on vient de Python qui permet de faire `t.append(elt)` indépendamment d'une limite de taille; en fait c'est parce qu'en python ce ne sont pas vraiment des tableaux mais plutôt un mélange. À chaque fois que vous dépassez la limite de taille du tableau, sans vous le dire, Python augmente sa taille (elle croît selon la suite de Fibonacci) pour éviter un dépassement mémoire. L'avantage de cet objet est que quand on connaît par avance la taille de ce que l'on va manipuler, le tableau permet de récupérer un élément en complexité optimale ($O(1)$). En revanche, sa taille sera limitée, et si on veut l'améliorer, il faudra en payer le prix ($O(n)$ à chaque fois qu'on augmente la taille).

Exercice (3-1) / Tableau ? / 1⋆

Dire dans chacun des cas si un tableau est adapté ou non.

1. Stocker n fixé entiers en mémoire et y accéder rapidement
2. Stocker les connexions à un site web sur une période d'1 heure sachant qu'il y a au plus 10 connexions par minute.
3. Stocker les connexions à un site web sur une période de 5 minutes

(⋆)4. Implémenter une mémoire d'ordinateur

5. Faire des mesures de température toutes les 5ms pendant 10 secondes dans le cadre d'une expérience physique

Exercice (3-2) / Tableaux dynamiques / 1⋆

Un tableau dynamique est un tableau qui met à jour sa taille automatiquement quand il dépasse sa capacité. Voici quelques stratégies pour augmenter la taille d'un tableau à chaque fois qu'on la dépasse, sans justification et uniquement par votre intuition, dire les quelles sont intéressantes et pourquoi :

1. Dès qu'on dépasse n, on créé un nouveau tableau de taille $n+1$
2. Dès qu'on dépasse n, on double la capacité pour avoir $2n$
3. Dès qu'on dépasse n, on on prend le terme suivant dans la suite de Fibonacci.

III.2 - Listes

Une liste est une structure de donnée dans laquelle on a accès qu'au premier élément et à la suite. C'est-à-dire que pour représenter `1,2,3` sous forme de liste, on peut le voir comme ceci : `1:(2:(3))`. L'avantage principale est qu'il n'y a pas de taille pré-définie pour une liste, on peut à tout moment définir une nouvelle liste `4:l` par exemple avec `l = 1:(2:(3))`. Cependant, il y a un inconvénient, essayez de faire ce mini-exercice avant de lire la suite :

Exercice (3-3) / Inconvénient d'une liste / 1⋆

Selon vous, qu'elle va être la propriété utile d'un tableau qui ne sera plus vraie pour une liste ?

Cherchez l'exercice précédent avant de lire ceci : Une liste ne permet plus d'accéder à un élément en $O(1)$, puisqu'on a accès qu'au premier élément et à la suite, si on veut accéder au dernier élément, on doit tous les regarder 1 à 1 et attendre de tomber sur celui qui n'a rien après lui, puisque la complexité étudiée est celle dans le pire cas, on a bien du $O(n)$ pour la recherche.

Mais alors pourquoi utiliser une liste ? Elle permet non seulement de manipuler des données dont on a aucune idée de la taille, mais hormis cet aspect, dans certaines applications on n'a pas besoin de chercher un élément précis et on veut tous les parcourir, dans ce cas la liste comme le tableau sont en $O(n)$ et l'avantage de la liste est qu'elle est naturellement pensée pour être récursive, en effet en voici la définition :

Définition 3-0 / Liste

Une liste est soit la liste vide, soit un élément suivi d'une liste.

Ainsi on peut facilement donner des algorithmes sur les listes :

Exercice (3-4) / Plus grand élément d'une liste / 1⋆

Donnez un algorithme pour trouver le plus grand élément d'une liste : pensez récursif !

Exercice (3-5) — Liste ou tableau ? — 1⋆

Pour chacune des fonctions suivantes, dire si elle correspond à une liste ou tableau, justifier.

1. `append`
2. `pop`
3. `truc[i]`

Exercice (3-6) — Implémentation d'une fonctionnalité Pyth — 1⋆

Essayez de faire `tableau[::-1]` en python.

1. Quelle est cette opération ?

2 (⋆). Proposez une implémentation avec des listes.
3. Quelle est sa complexité ?
4. Faire de même avec des tableaux.

Exercice (3-7) — Concaténation — 1⋆

Proposez un algorithme qui prend en entrée deux listes et les concatènent, ie renvoie la liste des éléments de la première suivant des éléments de la deuxième.

Exercice (3-8) — Choix de structure — 1⋆

Supposons que vous vouliez stocker des données triées par ordre croissant et que vous souhaitiez avoir une structure qui permette d'optimiser certains algorithmes grâce à cette propriété, quelle structure allez-vous choisir : liste ou tableau ?

III.3 - Pourquoi le OCaml ?

Si vous avez déjà regardé un petit peu le programme, vous savez qu'il contient deux langages : C et OCaml. Le C est un choix logique, mais pour le OCaml cela peut vous sembler étrange. L'avantage du OCaml est qu'il fait partie des langages dans lequel il est simple d'écrire en récursif (il est pensé pour). Ainsi (et ce n'est qu'un exemple parmis de nombreux autres), on peut facilement manipuler des types qui sont définis inductivement. Cette facilité provient de l'outil `match` majoritairement, il permet de raisonner **par cas**, voici un exemple simple (je ne vous demande pas de comprendre la syntaxe, mais plutôt l'idée derrière):

```
let rec parcours l =
    match l with
    | [] -> print_string "Il ne reste plus rien\n"
    | [e] -> print_string "Il ne reste qu'un élément\n"
    | e::t -> (
        print_string "Je viens de trouver l'élément: " ^ e ^ "\n";
        parcours t
    );;
```

Si on met de côté la syntaxe, on remarque qu'on peut distinguer différents cas en précisant "la forme attendue", on a par exemple `[e]` qui signifie une liste d'un seul élément.

Remarque: On aurait pu retirer le cas `[e]` car la liste vide reste une liste donc ce cas est englobé dans `e::t`, ici je l'ai juste mis pour bien comprendre les possibilités

Implémenter ces algorithmes sera possible en C, mais beaucoup plus encombrant car C ne possède pas ce match (une méthode souvent utilisée consiste à ajouter un entier qui va indiquer le type, par exemple liste vide codée par 0, une autre par 1)

III.4 - Note de fin

Je ne pense pas que ce chapitre soit un chapitre où les exercices soient utiles. Le but est uniquement de bien comprendre la différence et quand utiliser l'un / l'autre. Vérifiez donc bien (en dressant un petit tableau comparatif par exemple) que vous avez compris cette différence.

IV - Représentation des réels

IV.1 - Représenter un entier

Pour représenter un entier dans un ordinateur, on utilise la base 2, ie une écriture **en binaire**. Puisque nos ordinateurs permetttent de stocker tout un tas de 0 et de 1, autant s'en servir !

On admet pour cela pour un théorème que vous verrez en début de première année (en maths):

Théorème 4-1 (Représentation en base k)

Pour $k \geq 1$, pour tout $n \in \mathbb{N}$, il existe une unique manière d'écrire n en base k, c'est-à-dire d'écrire $n = \sum_{i=0}^{p} \alpha_i k^i$ avec $\forall i, \alpha_i \in [\![0; k-1]\!]$.

Si on applique ce résultat pour $k = 2$, on obtient que tout entier peut s'écrire de manière unique comme une somme de puissance de 2. Par exemple, $5 = 2^2 + 2^0$ et l'unicité permet d'assurer qu'il n'y a pas d'autres écritures.

Exercice (4-1) / Une première propriété / 1⋆

Comment, à partir de son écriture binaire, dire si un entier est pair ou impair ?

Ainsi, on représente un entier dans un ordinateur comme cette somme. Par exemple, $31 = 1 + 2 + 4 + 8 + 16 = 2^0 + 2^1 + 2^2 + 2^3 + 2^4$, donc on écrit 11111 pour représenter 31. Attention, on écrit dans l'ordre "inverse", c'est-à-dire que le 1 le plus à gauche correspond à 1×16 et non 1×2^0. Pour un autre exemple, $14 = 8 + 4 + 2 = 1110$.

Aller plus loin (Entier maximum sur n bits)

Si on veut obtenir le plus grand nombre possible, il faut que tous les bits soient à 1. Dans ce cas on a $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$.

Exercice (4-2) | Autre justification | 1⋆

Proposez un autre argument que "pour maximiser il faut que les bits soient à 1" pour justifier qu'on ne peut pas faire mieux que $2^{n+1} - 1$.

Remarque: (INDICATION) utiliser l'unicité

Exercice (4-3) | Écriture binaire | 1⋆

Donnez l'écriture binaire des nombres suivants :

1. 72
2. 89
3. 1
4. 0
5. 987

À la main, on peut faire des petites valeurs sans trop de souci, cependant, vous allez devoir écrire un algorithme pour celui-ci :

(⋆ ⋆)6. 2946654722

IV.2 - Sommer des entiers

Il s'agit du même algorithme que pour l'addition "à mains nues", on utilise une retenue.

Par exemple :

Exercice (4-4) | Somme en binaire | 1⋆

Calculez 11001 + 00011 à la main, comme en CE2.

Et voilà, vous savez sommer des nombres en binaires !

IV.3 - Soustraire des entiers

Pour faire $n - p$ en binaire, on fait $n + (-p)$. On procède ainsi car calculer $-p$ est simple si on prend la convention que le premier bit (celui de poids fort) est le bit de signe. Ainsi, si il vaut 0, on a un nombre positif, si il vaut 1, c'est un négatif.

Remarque: On peut voir le nombre binaire $\gamma 110$ comme étant $(-1)^{\gamma}110$ par exemple

Pour obtenir $-p$ voici la procédure :

On inverse tous les bits et on ajoute 1 au nombre.

Exercice (4-5) / Pourquoi ? / 1⋆

Pourquoi ajoute-t-on 1?

Exercice (4-6) / Calcul binaire négatif / 1⋆

On se place dans une convention où le bit de poids fort est le bit de signe. Calculez les nombres suivants :

1. $10011 + 01010$
2. $1000 + 0001$
3. $1111 - 1111$
4. $1111 - 1110$

(⋆)5. $1001 - 111000$ *L'étoile est par rapport aux nombres de bits différents*

IV.4 - Représenter des réels

Pour représenter un réel, on va l'écrire en écriture ingénieur, c'est-à-dire $(-1)^{\gamma}1, truc \times 10^{exposant}$. Le "truc" correspond à ce qu'on appelle la **mantisse**. Il faut donc trouver un moyen d'encoder cette forme. Pour cela on a la **norme IEEE-754**, on encode en binaire les différentes parties en les écrivant à la suite. Ainsi, le premier bit (de poids fort) est celui de signe, puis on a k bits pour l'exposant, le reste est réservé à la mantisse.

Par exemple, 1001111 avec 3 bits de d'exposant peut se réécrire 1 001 111. **Il y a une subtilité sur l'exposant**, 001 n'est pas le code de l'exposant, c'est celui de l'exposant additionné à une valeur qui permet de ne coder que des entiers positifs, cette valeur est $2^{k-1} - 1$. Ainsi, 001 représente 1 en base 10 et ici $k = 3$ donc le décalage est 1, donc on a encodé $1 - 1 = 0$. Ainsi, l'exposant est nul. Puis on a le bit de signe qui est sur 1, donc on a un négatif. Enfin la mantisse est 111 donc $0.5 + 0.25 + 0.125 = 0.875$, le réel encodé est donc $-1,875$.

Exercice (4-7) / Représentation des réels / 1⋆

Donner les réels correspondant aux écritures suivantes :

1. 11001001 avec $k = 3$
2. 00000001 avec $k = 4$
3. 101010101 avec $k = 3$

Les 3 exercices suivants sont à traiter dans l'ordre, ils vont vous apprendre à générer un nombre pseudo-aléatoire avec des opérations binaires.

Exercice (4-8) / Left et Right shift / 1⋆

On se donne les opérateurs de Left Shift (<<) et Right Shift (>>) qui permettent de "décaler" un nombre binaire. Par exemple, `5<<2 = 20` car `5 = 101` et décaler de 2 revient à ajouter 2 zéros, ie `10100` donc `4+16 = 20`.

Le Right Shift s'utilise de la même manière et sert à réduire le nombre binaire en "oubliant" son bit de poids faible.

1. Expliquez pourquoi `5>>1 = 2`

(⋆)2. Soit $n \in \mathbb{N}$, proposez un algorithme qui prend en entrée un entier $x \in \mathbb{N}$ et renvoie $x \bmod 2^n$ en utilisant les bit-shifts.

Exercice (4-9) / Xor / 1⋆

On définit sur des nombres binaires l'opération de XOR comme étant (bit par bit) 0 si les deux bits sont identiques et 1 sinon.

Par exemple, `11011 ^ 01101 = 10110`.

1. Calculez le résultat de `1010 ^ 0110`.
2. Écrivez une fonction qui prend en entrée deux nombres entiers positifs `a` et `b`, et renvoie **la distance de Hamming de ces deux nombres**, à savoir le nombre de bits différents entre `a` et `b`. On pensera évidemment à utiliser le XOR…

(⋆ ⋆)3. Écrivez un algorithme qui **échange le contenu de deux variables `a` et `b` (des entiers) sans utiliser de variable intermédiaire**.

(⋆)4. Écrivez un algorithme qui prend entrée un entier $n \in \mathbb{N}$ et renvoie la partié de ce nombre **en utilisant le XOR**.

Exercice (4-10) — Le XORShift — 2⋆

C'est un exercice qui peut être vu comme une application de fin de partie avec les deux précédents exercices. Il ne faut pas avoir traité intégralement les exercices précédents mais seulement avoir compris les concepts.

1. Remémorez-vous tout ce que vous savez sur les générateurs pseudo-aléatoires.

En python, on peut utiliser >> et << pour les shifts, ainsi que ^ pour le XOR.

On propose l'algorithme suivant:

Code — XORShift 32 bits

```
def xorshift(seed):
  x = seed
  x = x ^ (x << 13)
  x = x ^ (x >> 17)
  x = x ^ (x << 5)
  return x
```

2. Codez-le en python et appelez-le un grand nombre de fois en l'appelant à chaque fois avec le résultat de l'appel précédent.
3. En analysant le titre du code proposé, trouvez la période de ce générateur (après combien d'appels sommes-nous certains de retomber sur une valeur déjà générée) ? Est-elle atteinte pour toute graine ?
4. On admet qu'en remplaçant `13,17,5` par `13,7,17` on arrive à avoir un générateur sur 64 bits. Que devient la période trouvée précédemment ? Est-elle atteinte pour toute graine ?

Exercice (4-11) — Une introduction aux circuits booléens — 1⋆

Une porte XOR est un opérateur qui prend en entrée deux bits et renvoie 1 si ils sont différents, 0 sinon. Une porte NOT inverse son entrée (1 donne 0, 0 donne 1)

1. Expliquez comment obtenir une porte OR avec ces deux portes (OR vaut 1 si au moins une des deux entrées est vraie)
2. Expliquez comment obtenir une porte AND avec des deux portes.
3. On suppose avoir une porte qui décompose un nombre sur n bits en ses n bits (on a une entrée par bit), expliquez comment faire un XOR entre des nombres encodés sur n bits grâce à cette porte.

V - Raisonner inductivement ⋆

C'est l'un des chapitres les plus importants de tout ce document. Lisez-le plusieurs fois et essayez de comprendre un maximum de notions.

V.1 - Introduction

En informatique, vous aurez souvent besoin de décrire des **structures de données**, que ce soit des listes, des arbres, des UnionFind, etc. La plupart de ces structures sont **construites inductivement**. C'est-à-dire

qu'on a des cas de base, (la liste vide pour les listes par exemple) et des cas qui vont faire le lien entre eux. Par exemple on peut définir une liste de la sorte :
- Soit la liste est vide
- Soit la liste est un élément suivie d'une liste `e::l`

Ainsi, on peut dire que `[1;2;3]` est `1::(2::(3::(Vide)))`. Le gros avantage de cette définition va être sur les preuves.

Exercice (5-1) / Identifier la structure / 1⋆

Reprenez la définition d'une liste ci-dessus, identifiez le cas de base et le constructeur.

Remarque: CORRECTION DE L'EXERCICE : Le constructeur est `::` qui sert à concaténer, ie ajouter un élément à une liste. Le cas de base est la liste vide. On comprend donc qu'on ajoute des éléments un à un à une liste vide, ce qui est intuitif.

Exercice (5-2) / Les arbres / 2⋆

La définition d'un arbre binaire est disponible en **IV-2** (*si vous voulez la lire, ne lisez rien d'autre que la définition, sautez le théorème 4-1 et sa démonstration pour le moment*)

1. Identifiez le ou les cas de base d'un arbre binaire. Identifiez le ou les connecteurs d'un arbre binaire.

(⋆ ⋆)2. Essayez de donner un type équivalent pour des arbres d'arité `n` (l'arité d'un arbre est le nombre maximal de fils d'un noeud). Identifiez les connecteurs et les cas de base.

Théorème 5-1 (Raisonnement inductif)

Si une propriété est vraie pour les cas de base et qu'elle est préservée par l'application d'un constructeur, alors elle est vraie pour tout élément du type considéré.

Démonstration

Remarque: La démonstration est fondamentale pour le raisonnement inductif

Soit une propriété P vraie pour les cas de base et pour les autres constructeurs d'un type τ. On va **montrer par récurrence finie sur la taille de la construction de tout élément de type τ que la propriété est vraie dessus**.

On pose $H_{n\in\mathbb{N}^*}$: "P est vraie pour tous les éléments de type τ qui sont construits en moins de n constructeurs".

- H_1 est vraie car P est vraie pour les cas de base.
- Supposons H_n et montrons H_{n+1} : On note $(c_1, ..., c_{n+1})$ la suite finie de constructeurs menant à $t \in \tau$. Par hypothèse de récurrence, l'objet t' construit en suivant les constructeurs $(c_1, ..., c_n)$ est bien de type τ et respecte la propriété P. Or, c_{n+1} est un constructeur et la propriété est maintenue par passage à un constructeur par hypothèse, donc appliquer c_{n+1} à t' donne un objet qui vérifie bien τ. Cet objet est par définition t, donc t vérifie P.

D'où H_n vraie pour tout $n \in \mathbb{N}^*$ par principe de récurrence.

Ainsi on peut voir que savoir que la propriété tient pour les cas de base et que composer par un constructeur la préserve permet d'affirmer qu'elle sera vraie pour tout objet du type considéré.

Exercice (5-3) / Encore et encore / 1⋆

Refaites la démonstration du théorème **4-1** pour voir si vous l'avez bien comprise.

V.2 - Les arbres

Maintenant que vous avez une vague idée des raisonnements inductifs, voyons un cas concret (que vous reverrez en MP2I, mais le voir avant pour en comprendre les idées peut être bénéfique je pense) : **les arbres**

Définition 5-0 / Arbre binaire

Un arbre binaire est **défini inductivement** *(j'avais dit que c'était important comme terme)* de la sorte :

- L'arbre vide est un arbre
- Un arbre à un seul noeud est un arbre, qu'on nomme une **feuille**
- Un noeud avec 1 ou 2 fils qui sont eux-mêmes des arbres binaires est un arbre.

Voici un exemple d'arbre pour vous faire une idée :

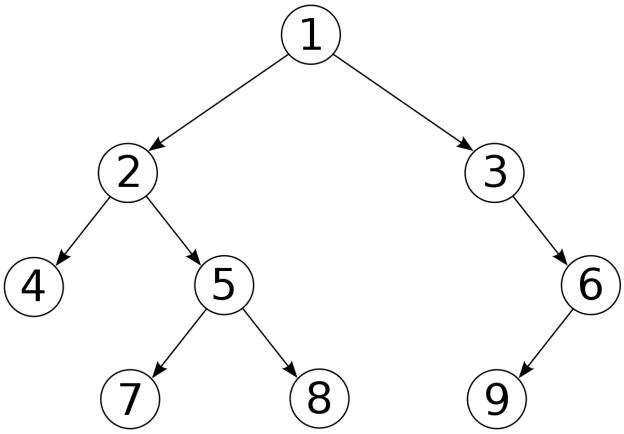

Figure 1: Exemple d'arbre (source: Wikipédia)

Aller plus loin (Le OCaml, ce sauveur⋆ ⋆)

Plus haut, j'avais dit que le OCaml était très pratique pour les types définis par induction, voyons voir pourquoi. On définit la hauteur d'un arbre binaire de la sorte :

- L'arbre vide est de hauteur -1 (*Pour permettre d'avoir la même hauteur pour un arbre qu'on le représente comme ayant des feuilles sans fils ou des feuilles qui ont pour fils l'arbre vide*)
- Un noeud x qui a pour fils a_1 et a_2 a pour hauteur $1 + \max(a_1, a_2)$.

À première vue, cette définition est compliquée à coder en C ou en Python, en revanche en OCaml, par le mot-clé `match`, on a le code magique suivant :

Code / **hauteur**

```
let rec hauteur a =
    match a with
    | Vide -> -1
    | Feuille -> 0
    | Noeud(a1,a2) -> 1 + max (hauteur a1) (hauteur a2);;
```

On voit alors la puissance du OCaml qui permet de raisonner inductivement directement dans son code.

Remarque: Pourquoi cette partie est ⋆ ⋆ ? Elle est très utile mais demande d'être capable d'avoir une intuition sur ce que fait un code sans connaître le langage, ce n'est pas une compétence simple et ce n'est pas un attendu de la CPGE puisque vous connaîtrez le OCaml, elle est donc juste ici pour les curieux ou ceux qui ont déjà des petites notions de OCaml. Pour ne pas effrayer les autres, j'ai donc mis ⋆ ⋆.

V.3 - Faire de l'inductif quand le langage n'est pas fait pour

Voyons une astuce très simple pour raisonner de manière inductive quand le langage de programmation utilisé n'a pas été pensé pour. Pour cela, on va partir du python et on va chercher à représenter un arbre.

L'idée est d'ajouter une variable qui va permettre de savoir dans quel "type" on se trouve. Ainsi, on prendra le type suivant :

Code / **type arbre**

```
class Arbre:
    def __init__(self):
        self.type = 0 #0 = Vide, 1 = Feuille, 2 = Noeud
        self.f1 = None
        self.f2 = None
```

Ainsi, on peut facilement faire un "match" en Python, par exemple, voici une fonction qui donne la hauteur en Python *définition de la hauteur d'un arbre binaire disponible dans la section Aller Plus Loin ci-dessus*:

Code — **Hauteur d'un arbre en Python**

```
def hauteur(a):
    if a == None : return -1 # cas à gérer car on a mis les fils à None par défaut
    else if a.type == 0 : return -1
    else if a.type == 1 : return 0
    else : return 1 + hauteur(a.f1) + hauteur(a.f2)
```

Remarque: Certains exercices de cette section peuvent-être délicats (vraiment), c'est normal si vous bloquez dessus : vous les reverrez probablement en cours et le fait de les avoir cherché vous donnera un énorme avantage au niveau de l'intuition sur ce genre d'exercices. Le but n'est encore une fois pas de chercher, mais de comprendre.

Exercice (5-4) — **Noeuds d'un arbre en fonction de sa haute** — 2⋆

(⋆ ⋆)1. Trouvez un encadrement du nombre de noeud d'un arbre selon sa hauteur.
2. Montrez cet encadrement par un raisonnement inductif.

Remarque: Cet exercice étant fondamental, en voici la correction.

Démonstration

1. Obtenons une minoration et une majoration séparément :

- Si on a une hauteur h, on a h "couches" de noeuds. Le cas où il y a le moins de noeuds est celui où les couches sont le moins remplies. Pour cela, on considère le cas où il n'y a qu'un seul noeud par couche (ce genre d'arbre s'appelle arbre **en peigne**). On a alors $n = h + 1$ (essayez de bien comprendre le +1 via un dessin).
- De même, le pire cas est celui où chaque noeud a 2 fils, dans ce cas, chaque couche de hauteur k a 2^h noeuds. On obtient donc $\sum_{k=0}^{h} 2^k = 2^{h+1} - 1$. Cette somme est une somme géométrique, qui est au programme terminale et vous en aurez souvent besoin.

2. **Question capitale pour vérifier si le raisonnement inductif est compris.**

- Si l'arbre est vide, on a sa hauteur qui vaut -1 et son nombre de noeud qui vaut 0, donc $0 \leq 0 \leq 0$.
- Si l'arbre est une feuille, il a 1 noeud et est de hauteur 0. Donc $1 \leq 1 \leq 1$.

On a les cas de base qui sont vrais. Soit un arbre $A = (g, d)$ (on note g le fils gauche et d le fils droit) tel que g et d vérifient l'encadrement proposé.

Alors, $h_A = 1 + \max(h_g, h_d)$ et $n_A = 1 + n_g + n_d$. Donc, en sommant les inégalités :

$$h_1 + 1 + h_2 + 1 + 1 \leq 1 + n_1 + n_2 \leq 2^{h_1+1} + 2^{h_2+1} - 1 \leq 2 \times 2^h - 1$$

Or, $(h_1 + 1) + (h_2 + 1) \geq h$ car $h = h_1 + 1$ ou $h = h_2 + 1$ et l'autre valeur est ≥ 0. Donc : $h + 1 \leq h_1 + h_2 + 1 \leq 1 + n_g + n_d \leq 2^{h+1} - 1$

Exercice (5-5) / Parchemin, le retour / 1⋆

On considère le type parchemin défini dans l'exercice Palindrome du chapitre 2.

Montrez que n'importe quel mot peut être encodé par ce type.

Exercice (5-6) / Array to List / 1⋆

1. Écrire une fonction qui prend en entrée un tableau et renvoie la liste correspondante.
2. Écrire une fonction qui prend en entrée une liste et renvoie le tableau correspondant.

Exercice (5-7) / Parcours d'un arbre / 1⋆

On définit le parcours **préfixe** d'un arbre de la sorte :

- Quand on arrive à un noeud qui a pour enfant (g, d), on fait une action sur le noeud (on le parcourt, ça peut être l'afficher par exemple), puis on rappelle le parcours sur g et après sur d.

1. Dessinez un arbre de hauteur 2, donnez-en le parcours préfixe. *De manière générale, c'est important en prépa de toujours prendre des exemples simples et d'appliquer les algorithmes qu'on nous donne, même si ce n'est pas demandé.*
2. Montrez que tous les noeuds de l'arbre sont ainsi parcourues.
3. Est-ce que parcourir le noeud avant ses fils, entre ses fils, ou après ses fils change quelque chose à la propriété de la question 2?

Exercice (5-8) / Arbre binaire de recherche / 3⋆

On définit un arbre binaire étiqueté comme un arbre binaire où les noeuds ont une valeur (un entier par exemple).

Un arbre binaire est dit de recherche s'il a la propriété suivante : Pour tous noeuds $n = (x, g, d)$ *(valeur x, fils gauche g et fils droit d)*, on a que si y est une valeur dans g, alors $y \leq x$ et si y est une valeur dans d, alors $y > x$.

1. Est-ce grave si on modifie $y > x$ par $y \leq x$?
2. Où se trouve le maximum d'un arbre binaire de recherche ? Le minimum ?
3. Est-ce que, étant donné un arbre binaire quelconque, on peut permuter ses valeurs de manière à avoir un arbre binaire de recherche ?
4. Pourquoi, selon vous, appelle-t-on ces arbres "de recherche" ?
5. Proposez un algorithme de recherche dans ce type d'arbre.

(⋆ ⋆ ⋆)6. Donnez la complexité de votre algorithme de la question 5

7. Est-ce qu'inverser $y \leq x$ et $y > x$ est gênant ?
8. Quelle forme voudrions-nous que l'arbre ait pour que la recherche soit efficace ?

Exercice (5-9) / Tri par tas / 1⋆

On appelle **tas max** un arbre binaire tel que tout élément est inférieur à son père *(on peut définir de même les tas min)* et qui est complet sauf éventuellement sur la dernière ligne, et cette dernière ligne est remplie de gauche à droite. Un arbre binaire est dit complet si chaque hauteur possède 2^h noeuds.

La définition d'un tas permet de le voir comme un tableau, en effet, on met le premier élément à la racine, les éléments 2 et 3 sont ses fils, les éléments 4 et 5 les fils du deuxième, les éléments 6 et 7 les fils du 3, ainsi de suite (on parle de **parcours en largeur** d'un arbre).

1. Donnez une relation qui donne pour un noeud i l'indice de ses fils dans le tableau et l'indice de son père.

2.a) Dans un tax max, où se trouve le maximum ? Dans un tas min, où se trouve le minimum ?

(⋆ ⋆) 2.b) Comment élimineriez-vous la racine d'un tas ? Remarque: (INDICATION) il faudrait l'inverser avec un certain élément puis faire quelque chose sur cet élément désormais à la racine

(⋆)3. Supposons qu'on ait une fonction `heapify` qui fait un tas à partir d'une liste d'entiers, proposez un algorithme, à l'aide de la question 2, qui permet de trier la liste.

(⋆ ⋆ ⋆)4. Codez la fonction `heapify`.

Exercice (5-10) / Artihmétique / 3⋆

On définit le type `Calcul` de cette façon :

```
Calcul est un de ces éléments:
    - Entier(n)
    - Addition(Calcul c1, Calcul c2)
    - Produit(Calcul c2, Calcul c2)
    - Moins(Calcul)
```

1. A-t-on besoin de parenthèses ?
2. Codez de cette manière $1 + (2 * 3) * 4 - 7$
3. Si vous deviez représenter ce calcul sous forme d'arbre, comment étiqueriez-vous les noeuds ? Les feuilles ?
4. Montrez que tout calcul admet un arbre de calcul.

(⋆ ⋆ ⋆)5. Montrez l'**unicité** ce cet arbre de calcul, à permutation des arguments d'une opération près *(ie* $1 + 2 = 2 + 1$*)*

Exercice (5-11) / Combinatoire des arbres / 4⋆

Tombé plusieurs fois à l'écrit de l'ENS, en début de sujet

Combien existe-t'-il d'arbres binaires à n noeuds ?

Exercice (5-12) / Interpréteur / 4⋆

Écrire un programme qui prend en entrée une chaîne de caractères représentant un calcul valide (par exemple `"(1+2)*3 - (4 / 5)"`) et l'évaluer.

Remarque: On pensera à d'abord faire l'exercice disponible dans la section Récursivité qui vous fait coder la fonction `atoi` (qui prend en entrée une chaîne de caractères contenant uniquement des entiers et renvoie le nombre associé)

Exercice (5-13) / Fusion croissante / 3⋆

1. Écrire un algorithme qui prend en entrée 2 listes triées par ordre croissant et qui renvoie une liste triée correspondant à la concaténation de ces deux listes.
2. En donner la complexité (si elle n'est pas linéaire, vous pouvez essayer d'améliorer votre algorithme)
3. Écrire un algorithme qui prend en entrée k listes triées par ordre croissant et qui renvoie une liste triée correspondant à la concaténation de ces k listes.

($\star\star\star$)4. Quelle en est la complexité ? (on peut trouver en $O(n\log(k))$ avec n le nombre total d'éléments si on utilise une file de priorité)

VI - Raisonnement dynamique

VI.1 - Introduction

C'est au programme de NSI. mais ce n'est visiblement pas ce que retiennent le mieux les élèves, il est donc important de refaire un point dessus, ce n'est pas compliqué mais il faut avoir compris l'objectif derrière. Commençons par poser la définition d'un raisonnement dynamique :

Définition 6-0 / Raisonnement dynamique

On **raisonne dynamiquement** quand on utilise des sous-solutions optimales pour générer des solutions optimales à des problèmes.

Bon, bon, bon... Cette définition est peut-être un peu floue, non ? Pas de panique, je vais maintenant donner l'intuition.

Le principe d'un raisonnement dynamique est le suivant :
- On a un gros problème à résoudre, un problème compliqué.
- On trouve une manière de le **découper** en différents intervalles tels que, si on souhaite résoudre le problème sur un intervalle plus gros, on ait juste à l'avoir résolu sur des intervalles plus petits.
- On calcule récursivement les valeurs dont on a besoin, puis on s'en sert pour résoudre le gros problème.

Et c'est tout !

VI.2 - Le sac à dos

Voyons un exemple d'application à travers le problème dynamique le plus célèbre : **le sac à dos**.

On a un sac à dos qui permet de porter un poids maximal P_{max}. On a devant nous n objets, de valeurs $v_1, ..., v_n$ et de poids $p_1, ..., p_n$. On cherche à prendre des objets tels qu'on ne dépasse pas le poids P_{max} et on cherche en même temps à maximiser la somme des valeurs des objets pris.

Exercice (6-1) | Premier découpage | 2⋆

Essayez de trouver vous-même comment découper ce problème.

On va dynamiser ce problème en le découpant en intervalles de sous-problèmes. On pose $S_{i,p}$ le problème du sac-à-dos où on ne considère que les i premiers objets et un poids maximal p.

Quand on dynamise un problème, on doit s'assurer d'avoir des cases triviales, c'est l'équivalent de l'initialisation d'une récurrence ou d'un raisonnement inductif. Ici, si on a un poids maximal de 0 (sous réserve que chaque objet ait un poids non nul, ce qu'on supposera ici), on a un cas trivial et la valeur maximale est 0. De même, si on considère un seul objet, on peut très facilement résoudre la ligne correspondante (0 si $p \leq p_1$ et v_1 sinon).

Une fois que le découpage a été effectué et que nous avons trouvé les cases triviales, il faut réfléchir à une **relation permettant de remplir les autres cases**. Il faut s'assurer que ce soit un ordre remplissable, ie qu'on ne tombe pas sur un raisonnement qui se mord la queue (pour avoir (i, p) on a besoin de (i', p') qui a lui-même besoin de (i, p)). Ici, on peut en trouver un.

Exercice (6-2) | Relation | 1⋆

Essayez de trouver par vous-même une relation permettant de remplir le tableau (et par conséquent un ordre de remplissage).

Voyons quelle relation nous pouvons trouver :

On a deux possibilités pour chaque objet : soit on le prend, soit on ne le prend pas. Si on le prend, on se trouve dans un poids p_{new} tel que $0 \leq p_{new} = p_{act} - p_i < p_i$. Si on ne le prend pas, on est simplement dans le même cas qu'à l'objet précédent. Ainsi :

$p_{i,p} = max\Big(v_i + p_{i-1,p-p_i}, p_{i-1,p}\Big)$ si $p - p_i \geq 0$ et $p_{i,p} = p_{i-1,p}$ sinon.

On a une relation de récurrence, qui permet effectivement un remplissage puique pour remplir une case (i, p) on n'a besoin que de cases strictement avant.

Ainsi, ce problème est résolu dynamiquement ! Remarque: Si ça vous semble trop compliqué, pas de panique, vous trouverez dans la section exercice des exercices beaucoup plus abordables sur la programmation dynamique.

VI.3 - Plus long facteur commun

Voici un deuxième exemple, beaucoup plus compliqué, réservez-le en deuxième lecture !

Étant donné deux mots a et b, on appelle **plus long facteur commun** de a et b le plus long mot w tel que w apparaisse dans a et apparaisse dans b (pas forcément à la suite, mais avec des indices strictement croissants) . Par exemple, si on prend `ABRICOTS` et `ABRIBUS`,le plus grand facteur sera `ABRIS` **avec un S**. Voyons comment résoudre ce problème grâce à un raisonnement dynamique :

On note n la taille de a et p la taille de b. On va découper notre problème selon le préfixe de u et v que l'on regarde (un préfixe est un mot de la forme $u_1 ... u_i$). On a alors un tableau de cette forme :

TODO

Maintenant, il apparaît quelque chose de classique en programmation dynamique : **les cas de base**.

Ici, si on regarde le moment où on a un mot vide (noté ε), on est sûr que le plus grand facteur commun sera de taille 0 (puisque ε n'a pour facteur que ε). Ainsi, la première ligne et la première colonne peuvent déjà être remplies de 0 :

TODO

Désormais, réfléchissons à une formule pour, connaissant certains cas, donner (i, j). Le but est de trouver **un ordre de remplissage**. Pour cela on va chercher à donner une formule de récurrence, ie à écrire $pfc(u, v)_{i,j}$ *(le plus long facteur commun entre u et v en considérant les i premières lettres de u et les j premières de v)* en fonction de certains $pfc(u, v)_{i',j'}$.

Exercice (6-3) / Formule de récurrence / 2⋆

Pouvez-vous essayer de trouver la formule avant de lire la suite ?

L'idée est la suivante : Quand on veut le plus grand facteur entre u et v avec les i et j premières lettres de u et v, il y a deux cas :

- Si $u_i = v_j$, on peut tenter de trouver un facteur qui tient compte de cette égalité : $1 + pfc(u, v)_{i-1,j-1}$
- Sinon, on renvoie simplement $\max\big(pfc(u, v)_{i,j-1}, pfc(u, v)_{i-1,j}\big)$

Enfin, on fait attention dans le premier cas à tout de même proposer l'autre solution (on peut possiblement proposer des cas pathologiques où c'est plus optimale de ne pas compter une lettre commune). Ainsi, on peut remplir le tableau ligne par ligne, ou colonne par colonne, et on aura bien que quand on calcule (i, j), on aura déjà calculé toutes les valeurs dont on a besoin (grace aux cas de base).

VI.4 - Exercices

Exercice (6-4) / Monter un escalier peut être mathématiqu / 1⋆

Vous avez devant vous un escalier de n marches. (On peut supposer $n \leq 10000$ car de toute façon, vous n'aurez pas le courage de monter plus de 10 000 marches).

À chaque marche, vous pouvez soit en monter 2 d'un coup, soit une seule. Combien de manières différentes avez-vous de monter l'escalier ?

Par exemple, pour $n = 2$ on attend une réponse de 2 : 1 marche puis 1 marche ou 2 marches.

Quelle est la complexité de votre algorithme ?

Exercice (6-5) / Escalier à coûts / 2⋆

On se place dans le même contexte que l'exercice précédent, sauf que chaque marche a un coût. Par exemple, au lieu d'avoir $n = 4$, on peut prendre `[2,1,10,1]` et dans ce cas, on préfèrera faire `[2,1,1]` (1 marche puis 2 marches) plutôt que de commencer par prendre les deux marches.

Ecrire un programme qui prend en entrée un tableau correspondant aux prix des marches et qui renvoie le coût minimal.

Exercice (6-6) / Triangle de Pascal / 1⋆

Si vous ne savez pas ce qu'est un triangle de Pascal, je vous invite à regarder sur Google ou dans votre cours de Maths de terminale.

Ecrire un algorithme qui prend en entrée un entier n et renvoie un tableau correspondant aux n premiers étages du triangle de Pascal. Par exemple, pour $n = 3$, on renverra `[[1],[1,1],[1,2,1]]`.

Exercice (6-7) / Théorie des jeux par la programmation dy / 3⋆

On considère deux joueurs qui s'affrontent dans le jeu suivant :

Il y a en face d'eux un tableau de `n` éléments. À chaque tour, le joueur i peut choisir de prendre le premier ou le dernier élément du tableau et ajouter le score de cet élément à son score. Le jeu se termine quand il n'y a plus de nombre dans le tableau.

Le joueur 1 (qui commence à jouer) gagne si son score est supérieur ou égal à celui du joueur 2. Sinon, il perd.

Écrire une fonction `peut_gagner(tab)` qui prend en entrée un tableau de jeu et renvoie `True` si le joueur 1 a un moyen de gagner et `False` sinon.

Aller plus loin (Intuition sur la théorie des jeux)

Ici, on demande à ce que le joueur 1 ait "au moins un" moyen de gagner. En fait, c'est parce que **indépendamment** de ce que fait son adversaire, il pourra toujours jouer **au moins un coup** qui lui permet d'être sûr de gagner. Ainsi, quand vous ferez de la théorie des jeux en deuxième année, la condition de victoire pour un joueur est la suivante :

Pour tout choix du joueur 2, il existe une série de coups du joueur 1 qui lui permet de gagner.

On ne demande en particulier pas à ce que tous les coups du joueur 1 lui permettent de gagner.

Exercice (6-8) / ((()))((())) / 3⋆

Écrire une fonction qui prend un entier n et renvoie le nombre de manières que l'on a de disposer n parenthèses (donc $2n$ si on compte ouvrantes et fermantes) en respectant un parenthésage correct (quand on ferme une parenthèse, il y en avait au moins une qui était ouverte)

Exercice (6-9) / Distance d'édition (de Levenshtein) / 4⋆

On appelle distance d'édition entre deux mots la distance minimale d'une série d'opérations qui permettent de passer de u à v. Les opérations sont les suivantes (applicables uniquement à u, v reste fixe) :

- Ajouter une lettre dans u pour un coût de 1
- Supprimer une lettre de u pour un coût de 1
- Remplacer une lettre de u par une autre pour un coût de 1.

Par exemple la distance d'édition de `MP2I` à `MPI` est 1 en supprimant `2` dans `MP2I`. Cependant, on aurait aussi pu s'amuser à remplacer le `I` par un `2` et supprimer le `2` initialement présent, on aurait eu une distance de 2 : on fera attention à bien renvoyer la distance minimale.

Étant donné deux mots u et v, donnez la distance d'édition entre u et v à l'aide d'un algorithme dynamique.

Exercice (6-10) / Piège à eau / 4⋆

On vous donne un $n \in \mathbb{N}$ représentant des niveaux d'élévations du sol. Par exemple, si on prend le tableau `[2;1;2]`, on a un mur de taille 2, un mur de taille 1 puis un mur de taille 2 (dans l'ordre, posé horizontalement, ils sont tous de largeur 1)

Étant donné un entier n et un tableau correspondant, combien de case d'eau peut-on emprisonner ? (Imaginez qu'il pleut, combien de cases peuvent contenir de l'eau en ayant un mur à gauche et à droite)

VII - Retour sur trace ⋆

Remarque: Cette section est en cours d'écriture / relecture, elle n'est pas terminée.

VII.1 - Forcebrute

Jusqu'à présent on considérait des problèmes dans lesquels on pouvait trouver la réponse avec quelques parcours sans pouvoir se tromper, par exemple pour calculer la hauteur d'un arbre il faut parcourir l'arbre mais on est sûrs qu'un parcours va suffire. Si désormais je vous donne un PC avec un mot-de-passe et que je vous demande de le dévérouiller, vous ne pouvez pas en un parcours (en un mot de passe) être sûr de trouver le bon (ou alors vous êtes un oracle), il parait donc compliquer de donner un algorithme "intelligent" pour le faire. C'est justement le but de la force brute:

Définition 7-0 / Bruteforce

Un algorithme bruteforce est un algorithme qui, étant donné un problème et un ensemble possible de solutions, essaye toutes les solutions jusqu'à en trouver une ou toutes dans le domaine voulu.

Par exemple, si vous savez que le mot de passe est de taille 12, vous allez générer tous les mots de passe possibles de 12 caractères jusqu'à trouver le bon.

Exercice (7-1) **Bruteforce** **1⋆**

1. Écrire un algorithme python qui prend en entrée un mot de taille 8 et qui essaye tous les mots de taille 8 jusqu'à trouver celui passé en entrée.
2. Écrire un algorithme python qui prend en entrée un mot de taille ≤ 8 et qui essaye tous les mots nécessaires jusqu'à trouver celui passé en entrée (on n'utilisera pas `len` sur l'entrée).

VII.2 - Retour sur trace

Pour certains problèmes, tester **toutes** les solutions est une perte de temps, par exemple si on prend une grille de sudoku déjà remplie et qu'il nous reste les cases $c_1, ..., c_k$ à remplir, l'algorithme bruteforce va tester toutes les valeurs de $[\![1; 9]\!]$ dans chaque c_i (ça fait beaucoup de possibilités). Pourtant, si on a déjà rempli $c_1, ..., c_{p<k}$ et que la solution actuelle a deux lignes (ou colonnes, ou carrés) qui ont la même valeur, toutes les solutions $c_1, ..., c_p, ..., c_k$ seront fausses. On a donc envie de **s'arrêter dès qu'on a fait une erreur**.

Définition 7-0 **Retour sur trace**

On considère la solution vide (qui est valide si l'entrée l'est, si elle ne l'est pas le problème n'a pas de solution). Ensuite, pour chaque position on essaye toutes les valeurs possibles et dès qu'on arrive à une solution partielle fausse (fausse selon une fonction qu'on se donne en entrée qui vérifie qu'une solution partielle reste une solution partielle si on modifie c pour la valeur v) on **passe à la valeur suivante pour la case actuelle si possible, et sinon on renvoie faux**.

Prenons un exemple concret: On va écrire un algorithme qui crée des couples de personnes selon des contraintes (soit A et B veulent absolument être ensemble, soit ils s'en moquent, soit ils ne veulent pas du tout être ensemble).

Entrée: `(Alice veut être avec Bob), (Bob veut être avec Alice), (Charlie ne veut pas être avec Bob), (Charlie se moque de Eve), (Eve se moque de Charlie)`

Algorithme à la main:

```
- Alice
  - Coupleé avec Alice
    - Impossible car on ne peut pas lier une personne à elle-même
  - Couplée avec Bob
    - Respecte les contraintes, on continue
    - Bob est déjà couplé avec Alice donc on génère pour la personne suivante
      - Charlie est couplé avec Alice
        - Impossible car Alice est déjà couplé
      - Charlie est couplé avec Bob
        - Impossible car Bob est déjà couplé
      - Charlie est couplé avec Charlie
        - Impossible car on ne peut pas lié une personne à elle-même
      - Charlie est couplé avec Eve
        - Ok
        - Eve est déjà couplé donc c'est bon
```

Dans cet exemple on a réussi, montrons un exemple qui se passe moins bien, avec 2 personnes: Entrée: `(Alice veut être avec Bob),(Bob ne veut pas être Alice)` Algorithme à la main:

```
- Alice couplée avec Alice
  - Impossible car on ne peut pas lier une personne à elle-même
  - Alice couplée avec Bob
    - Impossible car Bob ne veut pas d'Alice
  - Il n'y a plus de possibilités, Alice ne peut pas être couplé
- On n'a plus personne à modifier: c'est perdu.
```

Donnons un dernier exemple, qui, cette fois-ci réussi mais après un essai infructueux. Vous **noterez bien l'effacement des choix à chaque erreur** Entrée: `(A moque B), (B moque A), (C veut A), (D se moque de tout le monde)` Algorithme à la main:

```
- A couplée avec A
  - Impossible
- A couplée avec B
  - B déjà couplé
  - C couplé avec A
    - Impossible car A déjà couplé
  - C couplé avec B
    - Impossible car B déjà couplé
  - C couplé avec C
    - Impossible
  - C couplé avec D
    - Impossible car C veut A
  - Plus de solution possible, renvoie faux
- A n'est plus couplé à B. A couplé à C
  - B couplé avec A
    - Impossible car A déjà couplé
  - B couplé avec B
    - Impossible
  - B couplé avec C
    - Impossible car C déjà couplé
  - B couplé avec D
     - C déjà couplé
     - D déjà couplé
     - Solution trouvée
```

Il faut faire attention à quelques points:

- Il faut bien faire reculer la solution partielle de 1 quand on a une erreur. Dans notre exemple des couples, si A est couplée à B et qu'on veut coupler A à C, il faut penser à découpler B de à A.
- Quand on renvoie faux ce n'est pas la fin, sauf si c'est le premier choix qui renvoie faux. De manière générale, `faux` = retour en arrière de 1.
- La fonction de test de solutions partielles est souvent le point le plus compliqué.

VII.3 - Exercices de fin de partie

Les exercices sont durs et cette notion sera intégralement revue en MP2I donc je vous mets qu'un seul exercice, à traiter seulement si le cours vous en dit.

Exercice (7-2) / Sudoku / 3⋆

1. Reprendre l'exercice "Sudoku, première rencontre"
2. On va vouloir résoudre le sudoku par retour sur trace avec pour fonction de vérification la fonction qui vous dit si la grille viole une règle du sudoku. L'idée est de se donner un ordre arbitraire sur les cases et de combler les trous 1 à 1 avec toutes les valeurs possibles (de 0 à 9). Coder cet algorithme en python.

VIII - Introduction aux graphes

Les graphes font partie des points-clés du programme de MP2I et MPI, vous reprendrez de 0 en cours, mais il peut être avantageux d'avoir déjà quelques notions en tête pour gagner du temps.

VIII.1 - Qu'est-ce qu'un graphe

Définition 8-0 / Graphe non-orienté

Un graphe non-orienté est la donnée d'un ensemble S de sommets et d'un ensemble $A \subset P(S \times S)$ d'arêtes.

C'est-à-dire qu'un graphe, c'est des sommets (1, 2,... par exemple) et des relations entre les sommets qu'on appelle arêtes. On peut passer d'un sommet i à un sommet j sur un graphe si $\{i, j\} \in A$ ($P(S \times S)$ désigne les parties de $S \times S$ ie un ensemble de couples de sommets).

Aller plus loin (Notation ensembliste)

Les plus attentifs auront remarqué qu'on note une arête $\{i, j\}$ et non (i, j). C'est car on est sur un graphe non-orienté, ie on peut aller de i à j par une arête si et seulement si on peut aller de j à i en une arête. Un ensemble n'a pas de notion d'ordre, par exemple $\{1, 2\} = \{2, 1\}$. Par contre, quand on parle de graphe orienté, on utilise bien la notation (i, j).

Pour avoir les idées fixes, donnons un premier exemple de graphe. Modélisons le groupe d'amis d'Alice par un graphe tel que deux personnes soient reliées si elles se suivent mutuellement sur Instagram.

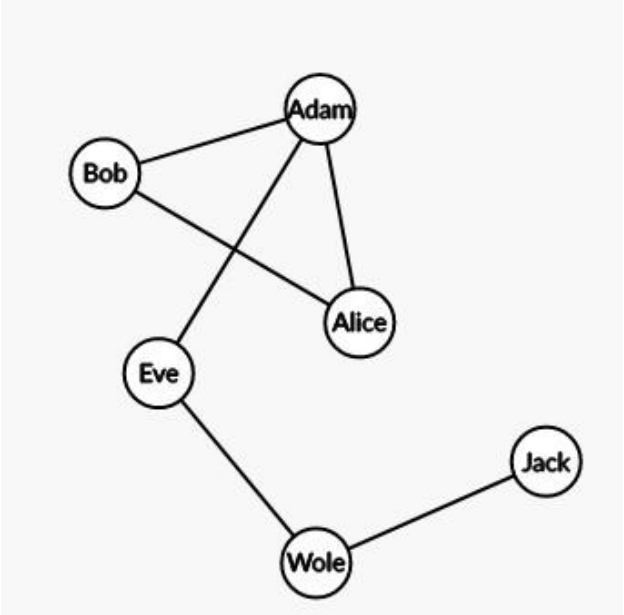

Figure 2: Exemple de graphe

On peut alors lire que `Eve` est amie avec `Wole` et `Adam` mais qu'elle n'est pas directement amie avec `Alice`.

VIII.2 - Notion de chemin

Définition 8-0 / Chemin dans un graphe non-orienté

On appelle chemin une suite finie de sommets $(s_1, ..., s_n)$ telle que $\forall i \in [\![1; n-1]\!], \{s_i, s_{i+1}\} \in A$. La taille d'un chemin est son nombre d'arêtes, donc $n-1$ selon cette définition.

Par exemple, si on considère le graphe de la figure 1, on a le chemin `Jack,Wole,Eve,Adam`, mais `Eve,Alice,Adam` n'est pas un chemin car il n'y a pas d'arête de `Eve` à `Alice`.

Il existe différents types de chemins pouvant nous intéresser.

Définition 8-0 / Chemin élémentaire

Un chemin est dit **élémentaire** s'il ne passe pas deux fois par le même sommet.

On a alors notre première démonstration importante sur les graphes :

Théorème 8-2 (Lemme de König)

Si il existe, dans un graphe $G = (S, A)$ un chemin de a à b, alors il existe un chemin élémentaire de a à b.

Démonstration

L'idée est simple : si on passe par un sommet x deux fois, on peut retirer une boucle au chemin :

Si on a un chemin de a à b qui passe deux fois par x, on a un chemin de la forme $(s_1, ..., s_i, x, s_{i+2}, ..., s_j, x, s_{j+2}..., b)$. Par définition d'un chemin, il y a une arête de x à $j+2$, on peut donc couper pour avoir un chemin de cette forme : $(s_1, ..., s_i, x, s_{j+2}, ..., b)$. On peut ainsi couper toutes les boucles et on obtient bien un chemin simple.

Exercice (8-1) | Chemin simple | 1⋆

Un chemin est dit **simple** s'il ne passe pas deux fois par une même arête. Peut-on établir un résultat similaire au Lemme de König pour les chemins simples ?

VIII.3 - Composante connexe

Définition 8-0 | Composante connexe

On dit qu'un ensemble C de sommets forme une composante connexe si $\forall (i,j) \in C^2$, il existe un chemin de i à j. (il en existe donc un de j à i en suivant le même chemin "à l'envers")

Par exemple, dans le graphe des amis d'Alice, il n'y a qu'une seule composante connexe qui est simplement tous ses amis.

Voici un deuxième exemple :

Exercice (8-2) | Composante connexe | 1⋆

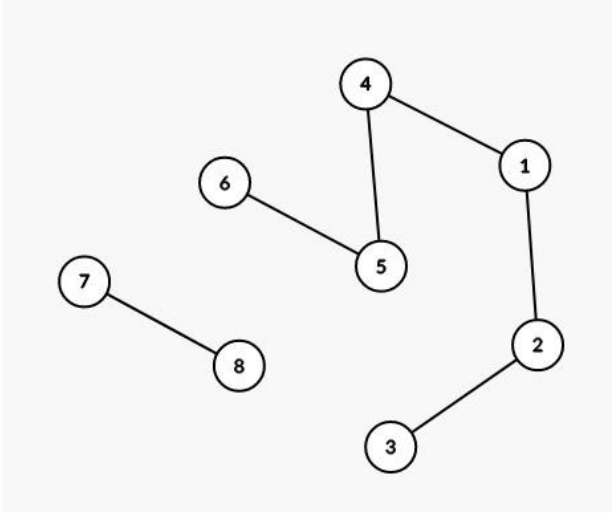

Figure 3: Composante connexe

Identifiez les composantes connexes du graphe de la figure 2.

VIII.4 - Cycle

Définition 8-0 — Cycle

Un cycle est un **chemin simple** $(s_1, ..., s_n)$ tel que $s_1 = s_n$.

Exercice (8-3) — Est un cycle ? — 1⋆

Est-ce que, dans un graphe non-orienté, pour toute arête $\{i, j\} \in A$, (i, j, i) est un cycle ?

Exercice (8-4) — Est un cycle ? — 1⋆

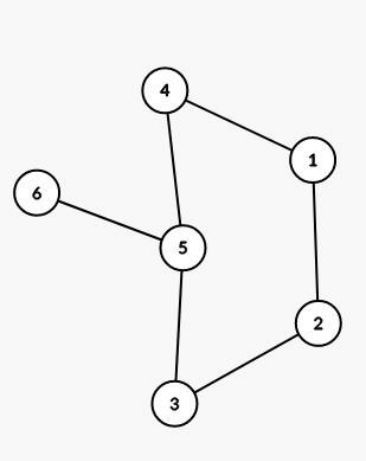

Figure 4: Graphe de l'exercice

Quel est le plus grand cycle de ce graphe ?

VIII.5 - Représenter un graphe

Il y a deux manières "usuelles" de représenter un graphe :

- Par matrice d'adjacence
- Par liste d'adjacence

Dans le premier cas, on représente les arêtes par un tableau 2-dimension tel que `tab[i][j]` soit à `True` si et seulement si l'arête reliant `i` à `j` est dans le graphe (et donc à `False` sinon). Dans le second, on a une liste par sommets et la liste du sommet `i` contient `j` si et seulement si l'arête reliant `i` à `j` existe.

Exercice (8-5) — Représentation d'un graphe — 1⋆

1. Représentez le graphe de la figure 3 sous forme de matrice d'adjacence

(⋆)2. Que peut-on dire de la matrice ainsi obtenue (quelle propriété a-t-elle) ? Est-ce que sur cet exemple ou est-ce propre aux graphes non-orientés ?

3. Représentez le graphe des amis d'Alice sous forme de liste d'adjacence.

VIII.6 - Parcours d'un graphe

On va maintenant passer à un algorithme qui est à la base de 80% des exercices de graphe : le parcours en profondeur.

L'idée est de parcourir les sommets en regardant le voisinage à chaque fois. On part d'un sommet `i`, qu'on marque comme étant "visité" puis pour chacun de ses voisins, si il n'est pas déjà visité, on appelle le parcours dessus.

Exercice (8-6) — Pourquoi s'embêter ? — 1⋆

Pourquoi ajoute-t-on cette idée de marquer les sommets visités ?

Voici un code du parcours en profondeur :

Code — Parcours en profondeur

```
visite = [False for i in range(g.taille)]

def auxiliaire(g,i):
  if visite[i]: return #déjà visité
  visite[i] = True
  for v in g.adjacence[i]:
    auxiliaire(g,v)

def parcours_prof(g):
  for i in range(g.taille):
    auxiliaire(g,i)
```

Exercice (8-7) — Autour du parcours en profondeur — 1⋆

1. Executez le parcours en profondeur sur le graphe des amis d'Alice.
2. Executez le parcours en profondeur sur le graphe de la figure 2.

(⋆)3. Quel lien peut-on faire en parcours en profondeur et composante connexe ?

(⋆ ⋆ ⋆)4. Prouvez-le.

Exercice (8-8) — Réflexion sur la représentation — 1⋆

1. Quelle représentation est utilisée dans le code proposé ?
2. Réécrivez le parcours en profondeur avec une autre représentation.

(⋆ ⋆)3. Quels sont les avantages et les désavantages de chacune des représentations en termes d'espace et de temps ?

VIII.7 - Graphes orientés

Il existe aussi des graphes qui sont **orientés**, c'est-à-dire qu'on ne prend plus les notations ensemblistes pour les arêtes. Voici la définition :

Définition 8-0 / Graphe orienté

Un graphe orienté est la donnée d'un ensemble S de sommets et d'un ensemble de couples, de sommets que l'on note A.

Remarque: Ainsi, un graphe orienté peut avoir une arête d'un sommet à lui-même, ce n'est pas interdit.

Voici un exemple de graphe orienté, on met une arête de i à j si i suit j sur Instagram. Puisqu'il se peut que i suive j sans que j suive i, un graphe orienté est plus adapté :

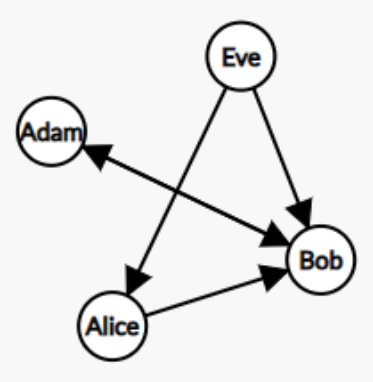

Figure 5: Graphe orienté

Remarque: On remarque qu'on a une arête dans les deux sens entre Adam et Bob.

Puisque vous repartirez de zéro sur les graphes orientés en prépa, je vous propose de voir la suite du cours comme un exercice (ce qui est d'ailleurs le cas, tout est sous forme d'exercice).

Exercice (8-9) / Circuit / 1⋆

Dans un graphe orienté, on ne parle plus de cycles mais de circuits, la définition reste sinon presque la même. "Presque" à un détail près, qui est la question de cet exercice :

Est-ce que dans un graphe orienté, un circuit à 2 sommets est effectivement un circuit ? (on a vu que pour un cycle il en fallait au moins 3)

(Faîtes un dessin)

Exercice (8-10) / Parcours en profondeur / 1⋆

Écrire un parcours en profondeur pour les graphes orientés.

Exercice (8-11) / Détection de cycle / 3⋆

1. Proposez, à partir du parcours en profondeur, un algorithme détectant les cycles dans un graphe non-orienté. *On fera attention à respecter la règle interdisant les cycles de 2 sommets*
2. Proposez, à partir du parcours en profondeur, un algorithme détectant les cycles dans un graphe orienté. *On fera attention à bien avoir traité l'exercice 6-9 avant.*

Exercice (8-12) / Composante fortement connexe / 2⋆

Dans un graphe non-orienté, la définition de composante connexe était “simple” car avoir un chemin de i à j implique d'en avoir un de j à i, ici c'est différent.

1. Pourquoi est-ce différent ? Proposez un exemple.

On définit alors une composante **fortement connexe** comme un ensemble de sommets **maximal au sens de l'inclusion** (ie on a oublié personne) tel que pour tout couple de sommets dedans, on est un chemin reliant le premier au deuxième et un reliant le deuxième au premier.

(⋆) 2. Proposez un algorithme naïf pour donner les composantes fortement connexes.

(⋆)3. Proposez un algorithme pour trouver les composantes fortement connexes sans limite de complexité

4. (⋆ ⋆ ⋆ ⋆) Pouvez-vous trouver en $O(n\log(n))$?

Remarque: Si vous bloquez sur la question 4, c'est tout à fait normal, le but est que vous cherchiez sans trouver, pour bien comprendre les raisonnements erronés que vous pourriez avoir sur les graphes. Pour les plus curieux, vous pouvez chercher *l'Algorithme de Kosaraju* sur Google, mais il sera vu en début de deuxième année, donc pas trop de panique.

Exercice (8-13) / Arbre de parcours en profondeur / 1⋆

Soit G un graphe non-orienté, on dit que c'est un **arbre s'il est acyclique et connexe.**

1. Représenter les graphes de quelques parcours en profondeurs (mettre une arête entre i et j si et seulement si i découvre j). Quelle propriété à ce graphe ? On admet que c'est un résultat correct (ou alors vous pouvez essayer de le prouver)

On appelle **racine** d'une composante connexe le sommet de la composante connexe visitée par un parcours en profondeur qui a pour père un sommet qui n'est pas dans la composante connexe (le père est le sommet qui l'a découvert).

2. Est-ce que la racine d'une composante connexe dépend du sommet duquel on commence le parcours ?

3. Prouvez l'unicité de la racine pour un sommet de départ fixé.

VIII.8 - Exercices

ENS ULM (8-14) / Autour de la connexité / 1⋆

On se donne dans cet exercice un graphe G non-orienté. On définit $c(G)$ comme le coefficient correspondant au nombre minimum de sommets à retirer de G pour obtenir un graphe non-connexe.

1. Que dire si G n'est pas connexe ?

Dans la suite, on supposera G connexe.

2. Que vaut $c(G)$ si G est le graphe complet ? (tous les sommets sont reliés entre eux)
3. Que vaut $c(G)$ si G est un anneau auquel on a ajouté une arête ? (composé d'un seul grand cycle en forme de cercle et d'une arête qui relie deux sommets sur le cercle)
4. Donner un graphe qui a $c(G) = 1$. Si vous avez pris de l'avance sur le programme (je n'invite pas à le faire, au contraire) : quelle grande famille de graphe vérifie cette propriété ?

Exercice (8-15) / Graphes biparties / 1⋆

Un graphe non-orienté est dit bipartie si il existe deux ensembles A et B de sommets tels que toute arête e du graphe connecte un sommet de A et un sommet de B, i.e. il n'y a jamais d'arêtes au sein d'une même composante.

1. Est-ce que tout graphe peut se mettre sous une forme bipartie ?
2. Dessinez un graphe bipartie à 5 noeuds. Pouvez-vous en proposer un à n sommets pour $n \in \mathbb{N}$? (*ne cherchez pas trop loin*)

($\star\star\star$)3. Écrire un algorithme qui prend en entrée un graphe et renvoie Vrai s'il est bipartie et Faux sinon.

J. Erickson (8-16) / Au moins deux solitaires / 2⋆

Prouvez que tout graphe connexe acyclique à $n \geq 2$ sommets a au moins 2 sommets de degré 1 (interdit de parler d'arbre ou de feuille)

IX - Travailler avec des mots

IX.1 - Définitions

On se donne un alphabet, noté Σ usuellement. Par exemple, en binaire on a $\Sigma = \{0; 1\}$, sur notre ordinateur on a $\Sigma =$ la table ascii, et pour mon chien on a $\Sigma = \{W, O, U, F\}$. Si vous voulez une définition "formelle":

Définition 9-0 / Alphabet

Un alphabet Σ est un ensemble **fini** de symboles (appellés lettres)

On définie ensuite les mots sur un alphabet :

Définition 9-0 / Mot

Un **mot** sur l'alphabet Σ est une suite finie de lettres.

Par exemple, $(a, b, r, a, c, a, d, a, b, r, a)$ est un mot sur l'alphabet Σ des lettres de la langue française. Cependant, ce n'est pas une notation très pratique donc on notera ce mot `abracadabra`.

IX.2 - Programmation

En Python, vous n'avez pas de distinction entre lettre et mot, vous pouvez par exemple écrire en Python :

```
a = 'a'
b = "b"
mot = "mot"
mot2 = 'mot2'
```

Il n'y a pas de distinction entre `''` et `""`. Heureusement pour vous, **en OCaml et en C il y a une distinction** !

Pour coder une lettre d'un alphabet Σ, on utilise `''`, et on appelle les objets de ce type des **caractères**.

Pour coder un ensemble de lettres, un **mot** (qu'on appelle le type `string`), on utilise `""`.

IX.3 - Recherche

Le problème le plus classique quand on parle de mots est sans doute le problème de recherche : est-ce qu'un texte `t` contient au moins une itération du mot `w` (qui peut aussi être un texte) ?

L'idée naïve est de tester à chaque indice (un indice est la position d'un "curseur" qui lit le texte, on commence en 0) si le texte qui commence à l'indice est celui qui nous intéresse suivi d'autre chose.

Par exemple, si on cherche `GAIA` dans `TAGAGAIA` :

```
TAGAGAIA
GAIA
```

On a `T` $\neq$ `G`, donc on décale d'un indice :

```
TAGAGAIA
 GAIA
```

On a `G` $\neq$ `A`. On continue et on finit par trouver le motif `GAIA` à la fin du texte.

C'est l'algorithme le plus naïf et il suffit pour des petits textes.

Exercice (9-1) / Quelques applications / 1⋆

1. Codez cet algorithme en Python
2. Donnez sa complexité en fonction de paramètres importants.
3. Modifiez-le pour qu'il compte le nombre d'itérations du motif
4. Donnez le nombre d'itérations du motif `Harry` dans le tome 1 d'Harry Potter.

On va désormais chercher à faire mieux : on va utiliser l'algorithme de Boyer-Moore-Horsepool.

IX.4 - Boyer-Moore-Horsepool

Cet algorithme sert à optimiser le décalage : on ne va plus décaler 1 à 1 à chaque fois, mais on va essayer de faire mieux.

La première idée non intuitive de cet algorithme est de **lire de droite à gauche et non de gauche à droite**. Vous pourrez réfléchir à la raison de ce choix en exercice de la section.

On va **pré-calculer** un tableau unidimensionnel où on stockera les différents décalages que l'on obtient. Pour cela, on se sert de la lecture de droite à gauche, si on veut chercher `ababb` dans `abbaababbab` :

```
abbaababbab
ababb
```

La première erreur est au premier test (`b` $\neq$ `a`), on cherche alors le dernier `a` du mot et on vient le coller sur le `a` qui a fait l'erreur. En cherchant le dernier, on s'assure de ne rater aucune possibilité (toutes les autres auraient échouées, au moins une fois sur ce caractère). Ainsi on cherche de la sorte :

```
abbaababbab
  ababb
```

Le décalage doit donc permettre de faire correspondre le caractère qui a fait échouer la tentative précédente et la dernière itération dans le motif à chercher de ce caractère. Pour obtenir ce décalage, on va stocker **le plus grand indice i tel que `motif[i] = c` pour tout caractère `c`**.

Exercice (9-2) / Pré-calcul / 1⋆

Donnez un code qui prend en entrée un motif et renvoie la table à pré-calculer.

Remarque: On pourra utiliser les 256 premiers éléments de la table ASCII pour le tableau et utiliser -1 comme valeur indiquant qu'il n'y a aucune itération.

Maintenant, il faudrait savoir utiliser ce tableau pré-calculé. On se place dans une recherche quelconque. On est actuellement à l'indice j dans le motif et on a commencé la recherche sur un indice i. On arrive à deux caractères différents, x pour le motif et y pour le texte. Par exemple, si on est dans la situation suivante :

```
GGATATTAGCCAGCA
    ATGGG
       ^
```

On a $i = 4$, $j = 3$ (on est à la 5ème lettre du texte et on est à la 4ème du motif), $x = G$ et $y = T$. On a alors 3 cas à distinguer :

- Si $table[y] = -1$, on n'a pas de y dans le motif, donc on ne peut pas réussir notre recherche, ainsi on décale après l'indice actuel, qui est $i + j$. Donc on se place en $i + j + 1$.
- Si $table[y] < j$, la dernière occurrence de y dans le motif est avant la lettre considérée, on veut alors la faire correspondre, pour cela on relance une recherche en $i + j - table[y]$
- Si $table[y] > j$ on ne peut pas décaler (ça reviendrait à revenir en erreur), donc on fait simplement $+1$ pour être sûr de ne rater aucune possibilité.

Exercice (9-3) / **Justification** / **1⋆**

1. Que dire si $table[y] = j$?
2. Justifiez, par un dessin et un exemple, les 3 cas.

Exercice (9-4) / **Avec les mains** / **1⋆**

Dans chacun des cas, recherchez à la main en précisant les cas considérés :

1. `Harry` dans `Hermione dit à Harry de rentrer dans la voiture`
2. `Python` dans `Java << Python`
3. `OCAML` dans `OOCCAAMMLOCAML`
4. `Bary` dans `Barycentre de Bary.`

Exercice (9-5) / **Reconnaissance de motif Boyer-Moore-Ho** / **1⋆**

Implémentez l'algorithme en Python.

X - Déboguer un programme

Vous avez désormais vu la totalité des points de cours importants pour aborder sereinement votre MP2I et pouvoir commencer à écrire du code en TP/TD/DM, cependant **vous aurez des erreurs**, beaucoup d'erreurs. C'est **normal** et je pense que c'est en les corrigeant que vous apprendrez le plus. Ce chapitre se décompose en deux parties, la deuxième étant pour les MP2I qui viennent de rentrer et qui découvrent l'enfer du Seg Fault. Les exercices de fin de partie sont traitables par les terminales tant qu'il n'y a pas de C ou de OCaml dedans.

X.1 - Les erreurs de logique

Le cas le plus simple Avant d'attaquer le débuguage, il y a certains cas dans lesquels vous n'avez presque rien à faire: quand votre IDE (ou compilateur, ou interpréteur) le fait pour vous. Par exemple si on donne `a = "a" + 2` à Python, on obtient `TypeError: unsupported operand type(s) for +: 'int' and 'str'` et on peut facilement comprendre notre erreur. C'est le cas le plus gentil et on peut ainsi facilement modifier notre code, cependant ce genre d'erreur n'arrivera très vite plus et vous aurez à la place des **erreurs de logique** (nom non-officiel) à savoir des erreurs dans le principe même de votre code, dans l'implémentation de votre algorithme, dans les arguments d'un appel de fonctions, etc.

X.1.a - Introduction à Coin Coin

Les autres 90% du temps (statistique personnelle) les erreurs ne se jouent qu'à un +1, qu'à un −1, qu'à un $\leq$ qui devait être un $\geq$ ou qu'à un i qui était en fait un j. Le reste du temps, il faut reprendre précisément le **cheminement de votre code** pour trouver l'endroit qui le fait buguer. Le meilleur moyen pour le trouver est d'expliquer votre code dans votre tête, de la manière la plus précise possible. C'est un peu comme la maïeutique de Socrate sauf qu'ici c'est vous-même qui vous interrogez. Vous pouvez évidemment expliquer votre code à un ami, mais il ne servira que de figuration et il perdra juste son temps, le mieux est d'investir dans cet outil révolutionnaire :

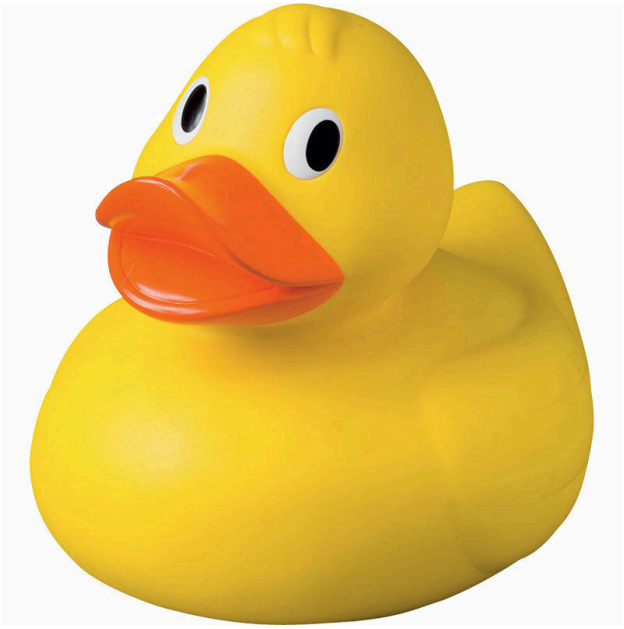

Figure 6: Le meilleur logiciel de débuguage

Dès que vous avez un problème, parlez-en à Coin Coin (Coin pour les intimes) et en lui expliquant votre problème **très précisément**, il vous soufflera la solution. Cependant, pour que ça marche il faut parler correctement à Coin Coin. Voici donc les règles à suivre quand vous parlez à Coin Coin :

1. Ligne par ligne tu expliqueras

   Il n'est pas question de laisser passer une ligne "triviale" sous-prétexte qu'elle est évidente ou qu'elle ne sert à rien, Coin Coin a besoin de **tous les détails** pour vous aider.

2. Les liens tu expliciteras

   Coin Coin a une mémoire très limitée, ainsi vous devez toujours lui **expliciter dans quel contexte la ligne est exécutée**. Par exemple la ligne `for i in range(100):` n'aura pas le même contexte d'exécution selon qu'elle est la première boucle `for` ou quelle est imbriquée dans d'autres (on pourraît avoir une erreur avec les indices qui se recouvrent). Ne soyez **jamais vague dans les liens**, car sinon en lisant trop vite vous passerez à côté de l'erreur

1. Les bornes tu donneras

   Coin Coin est également très mauvais en maths, ainsi vous devrez lui préciser chacun de vos bornes, que ce soit pour les boucles **ou pour les fonctions récursives** (ainsi vous n'aurez plus d'erreurs de + ou - 1)

4. Le résultat attendu tu compareras

   Coin Coin adore les histoires imagées, ainsi quand vous lui parlez d'une fonction, vous lui donnerez un exemple d'exécution pour des valeurs recouvrant toutes les possibilités, afin de tester tous les cas de votre code à la main

5. **Rien** tu ne supposeras

Enfin, vous ne devrez supposer aucun résultat sur des fonctions de vos anciennes questions traitées : l'erreur peut venir de n'importe où. Il se peut que vous ayez oublié un cas dans vos anciennes questions.

Discuter avec Coin Coin de cette manière permet en général de trouver votre problème. Même si dans certains cas vous **pouvez court-circuiter le processus**.

X.1.b - Court-ciruiter au bon endroit

Quand votre code est long, ou que vous n'avez envie de tester qu'une partie précise, vous pouvez **déterminer où est votre problème** sans raconter toute l'histoire à Coin Coin. Pour trouver ce qui fait planter votre programme vous pouvez utiliser des **print** pour trouver la bonne ligne à étudier (ou la bonne fonction). L'idée est la suivante : Quand votre programme plante, il aura exécuté toutes les lignes avant celle qui a mis fin à l'exécution (du moins en Python) et donc le dernier print affiché sera le dernier avant le bug. Vous pouvez **procéder par bloc** pour le trouver. Par exemple si on considère ce code :

```
 def f1():
  exec_1()
  exec_2()
  exec_3()

def f2():
  exec_4()
  exec_5()
  exec_6()

i = input("Entrez un choix")
if i=="0": f1()
else: f2()
```

Il va falloir mettre un `print` après le `i =`, ce qui permet d'être sûr que la fonction `input` marche bien (ici c'est immédiat mais si c'est votre propre fonction, il vaut mieux tester), s'il s'affiche on en met un entre chaque exécution de `f1` et un entre chaque exécution de `f2` et on voit lesquelles font buguer directement dans le terminal en mettant une première fois `0` comme en entrée et une deuxième fois autre chose. Il ne vous reste plus qu'à faire pareil dans les exécutions qui font buguer, ou à parler à Coin Coin selon le contexte.

X.2 - Segmentation Fault

Votre pire ennemi en prépa, de loin. En OCaml les erreurs sont assez précises en général, vous avez toujours au moins la ligne qui cause problème. En C c'est assez rare d'avoir la ligne, et il y a pire : quand le compileur ne marche pas et vous dit juste **Segmentation Fault**. Cette erreur apparaît quand vous accédez à une partie de la mémoire qui ne vous est pas allouée, ainsi elle est compliquée à corriger car elle peut avoir des centaines de raisons d'apparaître (malloc avec les mauvaises bornes, accès à une zone mémoire que vous avez free, ...). Vous pouvez alors, et c'est ce qui va vous faire apprendre le plus (car vous rentrerez plus en détail dans le fonctionnement de `malloc`, des pointeurs etc en général) parler à Coin Coin pour trouver votre erreur (attention aux prints, `C` ne les affichera pas forcément en fonction des optimisations du compilateur).

Cependant, vous n'aurez peut-être pas le temps de faire ce travail à chaque fois (encore plus si vous avez bien assimilé les différentes notions). Il y a un outil qui existe : **gdb**. Une fois vos fichiers compilés,

vous pouvez lancer `gdb votre_fichier.extension_des_exec` (l'extension dépendant de l'OS) et vous obtiendrez un terminal dont les lignes commencent par `(gdb)`. Une fois dedans, l'instruction `run` permet d'avancer dans le programme jusqu'à l'erreur, vous aurez alors le **nom de la fonction qui vous a fait planter** et mieux, vous aurez accès à la mémoire actuelle du programme pour pouvoir voir l'état dans lequel vous êtes. (je vous laisse regarder la documentation de `gdb` à ce sujet, elle l'expliquera mieux que moi, en particulier vous pourrez y voir comment placer des **breakpoints**)

X.3 - Exercices de fin de partie

Exercice (10-1) / Débuguage de boucles / 1⋆

```
tab = [[0 for i in range(10)] for j in range(20)]
for i in range(10):
  for j in range(20):
    print(tab[i][j])
```

1. Débuguez le code avec la technique de Coin Coin (explicitez bien les liens)

Exercice (10-2) / Débuguage de Fibonacci / 1⋆

```
def fibo(n):
  return fibo(n-1)+fibo(n-2)
```

1. Parlez à Coin Coin de votre exécution de `fibo(3)`, rendez-vous compte alors de l'erreur.

Exercice (10-3) / Tri par pile / 2⋆

```python
class Pile:
  def __init__(self):
      self.elements = []

  def est_vide(self):
      return len(self.elements) == 0

  def empiler(self, element):
      self.elements.append(element)

  def depiler(self):
      if not self.est_vide():
          return self.elements.pop()
      else:
          raise IndexError("La pile est vide")

  def sommet(self):
      if not self.est_vide():
          return self.elements[-1]
      else:
          raise IndexError("La pile est vide")

  def taille(self):
      return len(self.elements)

def tri_par_pile(liste):
  pile = Pile()
  for element in liste:
      pile.empiler(element)
  liste_triee = []
  while not pile.est_vide():
      element = pile.depiler()
      if not liste_triee or element > liste_triee[-1]:
          liste_triee.append(element)
      else:
          while liste_triee and element < liste_triee[-1]:
              pile.empiler(liste_triee.pop())
          liste_triee.append(element)

  return liste_triee
```

1. Débuguer ce programme en parlant à Coin Coin sur des exemples d'exécution

Exercice (10-4) / Liste doublement chaînée / 2⋆

```c
struct node{
 int val;
 struct node* left;
 struct node* right;
};
typedef struct node node;
typedef node* liste;

liste creer_liste_vide(){ return NULL; }

liste insert(liste l, int x){

    liste new = malloc(sizeof(node));
    new->val = x;
    new->left = l;
    if(l == NULL){
        return new;
    }
    l->right = new;
    new->right = l->right;
    return new;
}

void free_liste(liste l){
    if(l->left != NULL){
        return free_liste(l->left);
    }

    if(l==NULL){
        return;
    }
    free_liste(l->left);
    free_liste(l->right);
    free(l);
    l = NULL;
}
```

1. Débuguer cette structure (il y a peut-être une chose qui ne marche pas, ou deux, ou trois, ou dix)

Exercice (10-5) / Les pointeurs / 1⋆

```c
int main()
{
    int* ptr;
    int* nptr = NULL;
    printf("%d %d", *ptr, *nptr);
    return 0;
}
```

```c
int main() {
    int arr[5];
    for (int i = 0; i <= 5; ++i) {
        arr[i] = i;
    }
    for (int i = 0; i <= 5; ++i) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

```c
int main() {
    int *ptr;
    int value = 42;
    if (value == 42) {
        printf("%d", *ptr);  // Utilisation d'un pointeur non alloué
conditionnellement
    }
    return 0;
}
```

1. Corrigez les SegFault.

Des exercices plus compliqués sont à venir.

XI - Exercices sans thèmes précis

Exercice (11-1) / Borne inférieure des tris à comparaison / 4⋆

Montrez qu'un tri à comparaison a besoin de au moins $n \log(n)$ opérations.

Exercice (11-2) / Inspiré du Duc de Densmore / 4⋆

Au cours d'une journée, un nombre N de personnes se sont rendues à la mairie. Chaque personne, exceptée une, y est allée exactement une fois.

Personne ne se souvient quand il y est allé, mais tout le monde se souvient de qui il a croisé.

Comment déterminer la personne qui est allée 2 fois à la mairie ?

Remarque: À traiter une fois le chapitre sur les graphes lu.

Indice 1 : Raisonner sur un exemple

Indice 2 : Essayer de représenter sur une frise le moment où les personnes sont à la mairie.

Exercice (11-3) / Le meilleur pâtissier / 3⋆

Un concours national a lieu pour déterminer le meilleur pâtissier. Chaque personne a une note dans $[\![0; 100]\!]$.

Problème, le concours a eu lieu dans M lieux différents, chaque lieu possède une liste de candidat-note non triée. Par exemple si le candidat 1 a 80 points et le candidat 2 en a 0 (il ne faut pas le juger), on a la liste `"(1-80)::(2-0)"` (il faut donc formatter les chaînes de caractères) Chaque lieu a accueilli N candidats.

1. Comment obtenir la liste globale des candidats classés par ordre décroissant des notes (le meilleur est donc le premier) ?

(⋆ ⋆ ⋆)2. Comment obtenir la liste globale des notes classées par ordre décroissant ? On attend une complexité $O(M \times N)$.

Exercice (11-4) / Sélection aléatoire / 4⋆

Étant donné un fil de données que vous ne pouvez parcourir qu'une seule fois, comment obtenir un élément de manière uniforme ?

Remarque: Indice : $\frac{1}{2} \times ... \times \frac{N-1}{N} = \frac{1}{N}$

Exercice (11-5) / k-ème minimum / 4⋆

Étant donné une liste ou un tableau de N valeurs, comment déterminer le k-ème minimum ?

On pourra fournir une solution en $O(kN)$ pour débuter, mais il existe (et on invite à chercher) une solution en $O(N \times log(N))$.

Exercice (11-6) / exo oxe eox oex ...⋆ ⋆ / 1⋆

Étant donné un mot (une chaîne de caractères), afficher toutes les permutations de ce mot (attention, pour le mot `bob`, `bob` doit être affiché deux fois).

Remarque: (Indication) Commencer par générer les permutations de $[\![1; n]\!]$

XII - Corrections

XII.1 - Chapitre 1

1-1: Garder en mémoire le maximum qui vaut intialement le premier élément. A chaque élément du tableau, le comparer au maximum en mémoire.

1-2:

1. Toujours garder en mémoire l'élément précédent (au début le premier élément) et pour chaque élément, vérifier qu'il est $\geq$ à celui en mémoire, puis le mettre en mémoire.
2. Faire de même avec $\leq$ et $=$ et vérifier l'un des 3

1-3: Lui ajouter un paramètre qu'on décroît de 1 à chaque fois, si il est nul alors on renvoie faux. Si l'algorithme termine avant on renvoie vrai

1-6: $O(|L|)$! Le 100000 est une constante

1-12:

1. 10^4 2. $(26 * 2)^{14}$ 3. On peut diviser par $26 * 2$

1-13: On fait un XOR. Ou si vous ne connaissez pas le XOR, on met 1 en mémoire, puis pour tout élément, si l'élément divise le nombre en mémoire on divise, sinon on multiplie par cet élément.

XII.2 - Chapitre 2

2-1:

1. Elle calcule le produit des nombres pairs entre 2 et n
2.
```
def fact_pair(n):

    if n==0: return 1
    if n%2==1: return fact_pair(n-1)
    return n * fact(n-2)
```
3. On peut faire `fact(n)/mystere2(n)`
4. $O(n)$

2-3:

1.
```
def fib(n):
if n<=1: return n
return fib(n-1)+fib(n-2)
```
2. Ce sont les cas $n \leq 1$. On le voit car il n'y a plus d'appel récursif.
3. n décroît strictement à chaque appel, ce qui garantit la terminaison.

2-6:

1. On créé un tableau de taille `n` et pour chaque `fib(k)`, si la case k a déjà été remplie on renvoie sa valeur, sinon on calcule `fib(k)` avec `fib(k-1)` et `fib(k-2)` et on ajoute la valeur dans le tableau. Comme ça on calcule une seule fois chaque valeur de Fibonacci et on a une complexité linéaire.
2. L'idée est de faire `fib` qui renvoie le couple (u_{n-1}, u_n). Je vous laisse chercher avec cette indication.

2-7:

1. $O(n)$
2. Ça ne termine pas, on ne parle pas de complexité.
3. $O(\max(tab))$

4. $O(1)$
5. $O(n)$

2-12:

```
def rebours_aux2(n):
  print(n)
  if n!=1: rebours_aux2(n-1)

def rebours_aux1(k,n):
  print(k)
  if k==n:
    rebours_aux2(n)
  else:
    rebours_aux1(k+1)

def infinite(n):
  while True:
    rebours_aux1(0,n)
```

2-14: Obtenir par appel récursif les permutations de $[\![1; n-1]\!]$ puis ajouter n à toutes les positions dans toutes les permutations.

XII.3 - Chapitre 3

3-1:

1. Les tableaux sont optimaux car accès $O(1)$ et on connaît déjà la taille au début.
2. De même car on peut borner la taille et que c'est pas trop grand, donc un tableau est suffisant.
3. Les deux sont possibles, si c'est un très gros site c'est un peu compliqué de borner donc préférer des listes (mais en pratique ce sera une base de donnée)
4. Je sais pas pourquoi j'ai mis cette question, surtout que les deux sont utiles.
5. On peut borner la taille et c'est acceptable donc tableau.

3-3: Accès $O(1)$

3-4:

```
def maxi(a,b):
  if a>=b:
    return a
  return b
def max(l):
  if un_seul_element(l): return seul_element(l)
  if vide(l): ERREUR
  return maxi(element_act(l),max(suite(l)))
```

XII.4 - Chapitre 4

XII.5 - Chapitre 5

5-1: Cas de base: Liste vide. Constructeur: ::

5-2: Cas de base: Arbre vide, Feuille. Constructeur: Noeud

5-4:
1. $h+1 \leq n \leq 2^{h+1}-1$. Pour cela considérer le pire cas et le meilleur cas (que 1 fils / que 2 fils)
2. Ça sera fait en prépa.

5-11: J'ai dérapé, ne pas traiter.

XII.6 - Chapitre 6

6-1: On fait deux cas: Celui dans lequel on prend l'objet et celui dans lequel on ne le prend pas. Et on fait un appel récursif dessus (toujours en considérant l'objet k, pour permettre de le prendre plusieurs fois. De toute façon vu qu'on a une limite de poids on ne va jamais tourner à l'infini tant que l'objet n'a pas de poids 0 et s'il a un poids 0 il n'y a pas de solutions car la solution c'est ∞)

XII.7 - Chapitre 7

XII.8 - Chapitre 8

8-1: Si un chemin de u à v passe deux fois par une même arête, on peut simplement retirer tout ce qui fait entrer les deux passages dans l'arête et on aura toujours un chemin de u à v

8-2: `7,8` et `6,5,4,1,2,3`

8-3: Non ! Un cycle est un chemin **simple** et la l'arête $\{i, j\}$ est utilisée deux fois.

8-4: $(5, 4, 1, 2, 3)$ (ou tout cycle équivalent)

8-5: Todo, elle est importante celle-là

8-9: Oui ! Car l'arête (i, j) est cette-fois différente de celle (j, i) (différence ensemble / couple, au programme de terminale et première)

XII.9 - Chapitre 9

XII.10 - Chapitre 10

10-1: L'erreur à débuguer est le fait que ce soit `tab[j][i]`, je laisse Coin-Coin vous expliquer pourquoi.
10-2: L'erreur à débuguer est l'oubli de cas de base je laisse Coin-Coin vous expliquer pourquoi.

XIII - Crédits

Auteur : Clément ROUVROY (https://www.cr-dev.io/)

Merci à Wyrdix, Grégoire, et aux terminales qui ont proposé des exercices supplémentaires pour le polycopié et des retouches sur le cours.

Merci aux terminales qui ont relu le polycopié et permis de mieux calibrer les exercices / le nombre d'étoiles.

Merci à tous ceux qui m'enverront des exercices à mettre dans les versions prochaines.