Colles d'informatique

Sujets

MPII

Table des matières

I 3 mai 2022

I.1 Adam Glin

I.1.a Codage de Huffman

un texte est codé avec 8 caractères : $\{0, 1, 2, 3, 4, 5, 6, 7\}$ avec les fréquences $f(0) = 0,2$, $f(1) = 0,04$, $f(2) = 0,15$, $f(3) = 0,1$, $f(4) = 0,3$, $f(5) = 0,08$, $f(6) = 0,06$ et $f(7) = 0,07$.

Exercice I.1.1 - Un codage de Huffman
Déterminer un code Huffman associé.

I.1.b Fibonacci (ultra-)rapide

$(F_n)_{n\in\mathbb{N}}$ est la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$.

Exercice I.1.2 - Fibonacci 1
Pourquoi le calcul récursif naïf n'est-il pas recommandé ?
Décrire un algorithme de calcul des F_n de complexité linéaire.

Exercice I.1.3 - Fibonacci 2
Prouver qu'on a $F_{2n} = 2 \cdot F_n \cdot F_{n+1} - F_n^2$ et $F_{2n+1} = F_n^2 + F_{n+1}^2$.
En déduire un algorithme, qu'on espère plus rapide, de calcul des F_n.
Quelle est sa complexité ?

I.2 Clément Petit

I.2.a Haricots rouges et verts

Une boîte contient des haricots rouges et des haricots verts. On répète la routine :

- on retire 2 haricots,
- s'ils tous deux rouges, on les jette et on replace un haricot vert,
- s'il y a un haricot vert, on le jette et on remet l'autre dans la boîte (rouge ou vert).

Comme on retire un haricot à chaque étape, on finit par aboutir à un seul haricot.

Exercice I.2.1 - Haricots rouges et verts
Peut-on prévoir la couleur du dernier haricot en fonction de la configuration initiale ?

I.2.b Suite modale

On considère un tableau d'entiers tel que :

- Tous ses éléments sont uniques (pas de doublon).
- Ces éléments sont d'abord croissants, puis décroissants (un seul changement de monotonie).

Ceci assure l'existence et, surtout, l'unicité d'un élément maximal.

Exercice I.2.2 - Maximum d'une suite modale
Décrire une fonction qui reçoit un tableau du type ci-dessus et renvoie la valeur de l'élément maximal avec un algorithme de complexité logarithmique.

Indication : On pourra couper en 3.

I.3 Quentin Watelle

I.3.a Décalage

Exercice I.3.1 - Retournement partiel
Décrire un algorithme qui permet de retourner la portion d'un tableau entre les indices i et j.
Quelle est sa complexité ?

```
retourner 2 6 [|3; 2; 11; 8; 5; 12; 1; 14; 6|]
[|3; 2; 1; 12; 5; 8; 11; 14; 6|]
```

Exercice I.3.2 - Décalage
En déduire un algorithme qui décale de k éléments vers la droite de manière circulaire les éléments d'un tableau sans utiliser d'autre tableau.
Quelle est la complexité ?

```
decaler 3 [|3; 2; 11; 8; 5; 12; 1; 14; 6|]
[|1; 14; 6; 3; 2; 11; 8; 5; 12|]
```

I.3.b Égalité de Kraft

Un arbre binaire sur E est soit une feuille (vide) soit un nœud qui contient un élément de E, deux fils, droit et gauche, qui sont des arbres binaires sur E.
On note N_k le nombre de nœuds à la profondeur k et F_k le nombre de feuilles à la profondeur k d'un arbre binaire .

Exercice I.3.3 - Décalage
Montrer que $N_{k+1} + F_{k+1} = 2N_k$ et $N_0 + F_0 = 1$.

Si la hauteur de l'arbre est h, prouver l'égalité de Kraft $\sum_{k=0}^{h+1} \frac{F_k}{2^k} = 1$.

I.4 Nathan Coquerel

I.4.a Produit

Exercice I.4.1 - produit par additions et décalages
Prouver que le code suivant effectue bien le produit de deux entiers positifs
Combien effectue-t-il d'additions ?

```
produit(a, b)
    c <- 0
    tantque b > 0
        c <- c + a*(b mod 2)
        a <- 2*a
        b <- b div 2
    fintantque
    renvoyer c
```

I.4.b Plus longue sous-suite croissante

Un sous-suite d'une suite de nombre $(a_0, a_1, \dots, a_{n-1})$ est une suite de la forme $(a_{k_0}, a_{k_1}, \dots, a_{k_p-1})$ avec $0 \leqslant k_0 < k_1 < \cdots k_{p-1} < n$.

Exercice I.4.2 - Longueur maximale d'une sous-suite croissante
Décrire un algorithme qui calcule la longueur de la plus longue sous-suite **croissante** d'une suite dont les éléments sont supposés distincts.
L'objectif est de parvenir à une complexité quadratique en la longueur n de la suite.

Exercice I.4.3 - Sous-suite croissante maximale
Peut-on modifier l'algorithme pour qu'il renvoie une sous-suite croissante de longueur maximale ?

On veut améliorer la complexité.
Pour cela on maintient un tableau m de longueur n tel que m_k est le plus petit élément possible qui soit le dernier élément d'une sous-suite croissante de longueur k.
Par exemple, pour la suite $a = (4, 8, 7, 2, 9, 1, 3)$ on calcule successivement les valeurs de m pour les sous-suites de $(a_0, a_1, \dots, a_{i-1})$ et on obtient

i	m_0	m_1	m_2	m_3	m_4	m_5	m_6
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	4	8	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	4	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
4	2	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
5	2	7	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
6	1	7	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
7	1	3	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Exercice I.4.4 - Autre solution
Décrire l'algorithme utilisé.
Après avoir prouvé que la suite m est croissante, montrer qu'on peut obtenir une complexité meilleure qu'un $\mathcal{O}(n^2)$.
Comment obtenir la sous-suite croissante maximale ?

I.5 Samuel Duquesnoy

I.5.a Tri par A.B.R.

Exercice I.5.1 - A.B.R. et tris
Après avoir rappelé la fonction d'ajout d'un élément dans un arbre binaire de recherche, décrire comment on peut utiliser un arbre binaire de recherche pour trier un tableau.
Quelle complexité peut-on espérer ?

Exercice I.5.2 - A.B.R. et tas
Peut-on utiliser un arbre binaire de recherche pour implémenter un tas ?

I.5.b Flottants sur 8 bits

On souhaite coder les nombres à l'aide d'un octet seulement, ce n'est pas très réaliste mais illustre les phénomènes. On note $b_0, b_1 \dots, b_7$ les 8 bits, $b_i \in \{0,1\}$.
La mantisse est $m = 4 \cdot b_1 + 2 \cdot .b_2 + b_3 \in \{0,1,\dots,7\}$, l'exposant est $e = 8 \cdot b_4 + 4 \cdot b_5 + 2 \cdot .b_6 + b_7 \in \{0,1,\dots,15\}$, le nombre représenté est alors $x = (-1)^{b_0} \cdot \left(1 + \frac{m}{8}\right) \cdot 2^{e-8}$.

Exercice I.5.3 - Flottants 8 bits

- Que représente $(0,1,1,0,0,1,1,0)$?
- Quel est le plus petit réel positif représenté, a ?
- Quel est le suivant, b ?
- Quel est le grand réel représenté ?
- Quel est le précédent ?
- Quel est petit entier positif non représenté ?
- Comment est approché $0,1$?
- Comment est approché π ?

On voit que, pour les petites valeurs positives, on a un problème : il y a un écart entre 0 et a bien plus gros qu'entre a et b.

Exercice I.5.4 - Prolongement vers 0
Proposer une amélioration.

I.6 Gustave Gomila

I.6.a Découpage

Exercice I.6.1 - Split
Décrire un algorithme qui découpe un texte selon un caractère.

Par exemple `decouper("abahfrtrhuy", "h"` done la suite "aba", "frtr", "uy".

I.6.b Longueur de cheminement

Un arbre binaire sur E est soit une feuille (vide) soit un nœud qui contient un élément de E, deux fils, droit et gauche, qui sont des arbres binaires sur E.
La longueur de cheminement d'un arbre binaireest la somme des profondeurs de tous ses nœuds.

Exercice I.6.2 - Calcul de LC
Déterminer une fonction qui calcule la longueur de cheminement d'un arbre.

Exercice I.6.3 - Encadrement de LC
Si LC est la longueur de cheminement d'un arbre de taille n,
prouver $\sum_{k=1}^{n} \lfloor \log_2(k) \rfloor \leqslant LC \leqslant \frac{n(n-1)}{2}$.

Une arbre de hauteur h est quasi-complet si toutes ses feuilles sont à une profondeur h ou $h+1$.

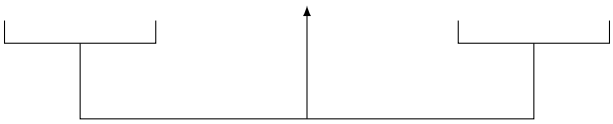
Exercice I.6.4 - Caractérisation des arbres quasi-complets
Prouver qu'un arbre de taille n est quasi-complet si et seulement si sa longueur de cheminement est minimale parmi les arbre de taille n et que cette longueur de cheminement vaut $LC = (n+1)h - 2^{h+1} + 2$ avec $h = \lfloor \log_2(n) \rfloor$.

II 17 mai 2022

II.1 Nicolas Jourdain

II.1.a Pièce trop lourde

Vous disposez d'une balance à plateaux qui permet de comparer deux masses avec une triple réponse, même poids, plus lourd à droite, plus lourd à gauche.



On dispose de n pièces dont une est fausse : elle a une masse strictement supérieure aux autres

Exercice II.1.1 - Tester
Si $n = 3$, peut-on déterminer la pièce fausse en une pesée ?

Exercice II.1.2 - Tester
Quelle est la valeur maximale de n permettant de déterminer la pièce fausse en p pesées ?

Exercice II.1.3 - Tester
Proposer une méthode pour $n = 3^p$.
Peut-on le faire avec une méthode qui ne dépend pas des résultats antérieurs.

II.1.b Parenthèses

Un texte contient des parenthèses qui doivent être bien associées.
"(af)u(ds(lo))" est bien parenthésée, "(af)u(d(s(lo))" ne l'est pas.

Exercice II.1.4 - Test de parenthésage
Décrire un algorithme qui teste si une chaîne est bien parenthésée.

Exercice II.1.5 - Association de parentèses
Décrire un algorithme qui renvoie, pour une chaîne bien parenthésée, la suite des coupes des positions des parenthèses associée.

Indication : on peut utiliser une pile.

Pour "(af)u(ds(lo))" la fonction doit renvoyer $(0, 3), (5, 12), (8, 11)$.
On considère maintenant deux paires de délimiteurs (,) et [,]. Chaque délimiteur ouvrant doit être fermé, et des délimiteurs différents ne peuvent se croiser : "(ae[j)r]" est invalide, "(t[df]dr(o))" est valide.

Exercice II.1.6 - Généralisations
Généraliser les exercices précédents dans ce cas.

II.2 Thibaut Midavaine

II.2.a Sacs de billes

Vous disposez de 1000 petites billes d'acier, chacune pesant 1 gramme.
On peut donc peser tout poids de 0 à 1000 grammes en équilibrant une balance avec le bon nombre de billes, qu'il faut ensuite compter. Mais ce n'est pas pratique ...
On se propose de placer toutes les billes dans des sacs étiquetés, par exemple

- 10 sacs de 1 bille
- 9 sacs de 10 billes
- 9 sacs de 100 billes

en gardant la possibilité d'équilibrer toute masse entre 0 et 1000 grammes avec un certain nombre de sacs.

Exercice II.2.1 - Une meilleure solution
Proposer une autre distribution avec moins de sacs

Exercice II.2.2 - Borne inférieure
Prouver qu'il faut au moins 10 sacs

Exercice II.2.3 - Solution optimale
Donner une solution avec 10 sacs.

II.2.b Reconstruction d'un arbre à partir de ses parcours

Un arbre binaire sur E est soit une feuille (vide) soit un nœud qui contient un élément de E, deux fils, droit et gauche, qui sont des arbres binaires sur E.

Exercice II.2.4 - Reconstruction
Montrer que si on connaît le parcours préfixe et le parcours infixe d'un arbre homogène sous la forme de deux listes alors on peut reconstituer l'arbre.
Décrire une fonction qui renvoie l'arbre binaire à partir des deux listes correspondant à ses parcours préfixe et infixe.

II.3 John Wang

II.3.a Égalité de Kraft

Un arbre binaire sur E est soit une feuille (vide) soit un nœud qui contient un élément de E, deux fils, droit et gauche, qui sont des arbres binaires sur E.
On note N_k le nombre de nœuds à la profondeur k et F_k le nombre de feuilles à la profondeur k d'un arbre binaire .

Exercice II.3.1 - Décalage
Montrer que $N_{k+1} + F_{k+1} = 2N_k$ et $N_0 + F_0 = 1$.

Si la hauteur de l'arbre est h, prouver l'égalité de Kraft $\sum_{k=0}^{h+1} \frac{F_k}{2^k} = 1$.

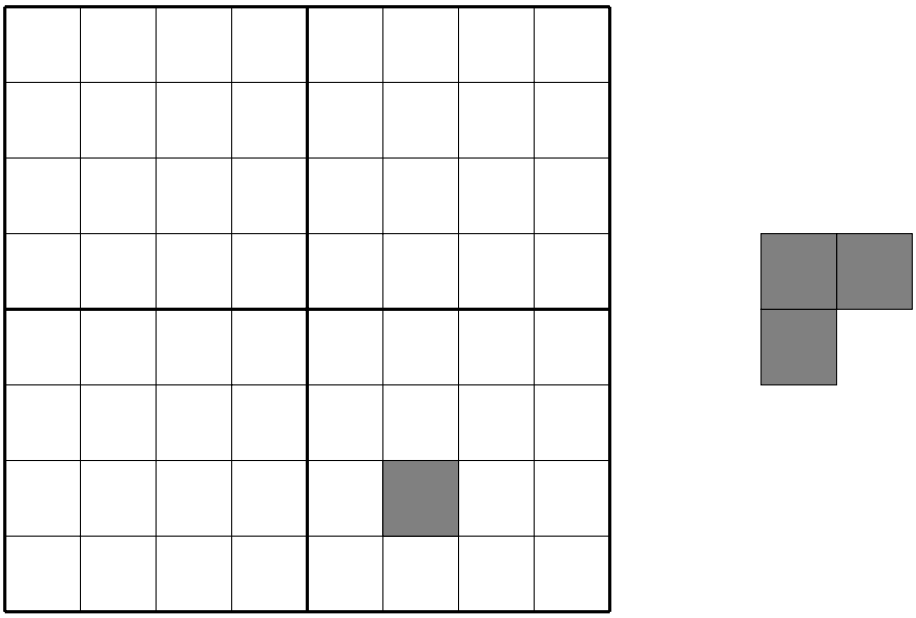
II.3.b Trigonométrie

Exercice II.3.2 - Itération de l'angle
Décrire une fonction `trigo(cosx, sinx, n)` qui retourne le couple $cos(nx), sin(nx)$ à partir des valeurs de $\cos(x)$ (`cosx`), $\sin(x)$ (`sinx`), et de n.
On commencera par un algorithme simple, linéaire, puis on donnera un algorithme logarithmique.

II.4 Loann Grosset

II.4.a Recouvrement par des triminos

On dispose d'un plateau de $2^n \times 2^n$ cases dans lequel on marque une case et de pièces en L formées de 3 cases.



Exercice II.4.1 - Algorithme
Prouver qu'on peut recouvrir le plateau en dehors de la case marquée par des triminos.

II.4.b Drapeau hollandais

Exercice II.4.2 - Le cas classique
Un tableau ne contient que les valeurs $a < b < c$. Décrire une fonction qui transforme le tableau en plaçant les termes dans l'ordre.
On obtient 3 bandes, comme le drapeau néerlandais. Le concepteur de cet algorithme est Edsger Dijkstra, qui est hollandais.

Exercice II.4.3 - Quicksort
En vous inspirant de cette fonction, décrire une fonction de séparation et une fonction de tri dans le style du quicksort en séparant la partie à trier entre deux bornes selon les termes strictement inférieurs, égaux et strictement supérieurs au pivot.
Dans quels cas cet algorithme est-il avantageux ?

II.5 Sothaline Huot

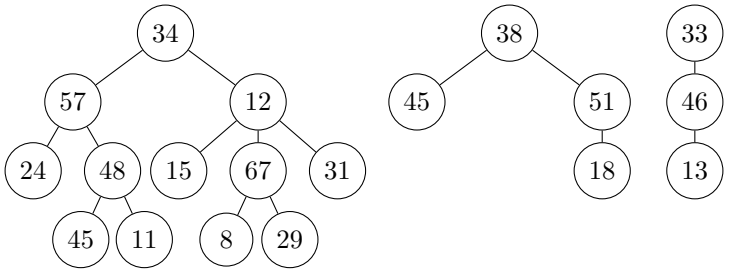
II.5.a Décomposition en facteurs premiers

On rappelle que tout entier positif n peut s'écrire sous la forme d'un produit de nombres premiers : $n = p_1 \cdot p_2 \cdot \cdots \cdot p_m$. La décomposition est unique si on impose la croissance $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_m$. On pourra utiliser que, dans ce cas, p_1 est le plus petit diviseur supérieur ou égal à 2 de n et que $p_2 \cdot \cdots \cdot p_m$ est la décomposition croissante en facteurs premiers de $\frac{n}{p_1}$.

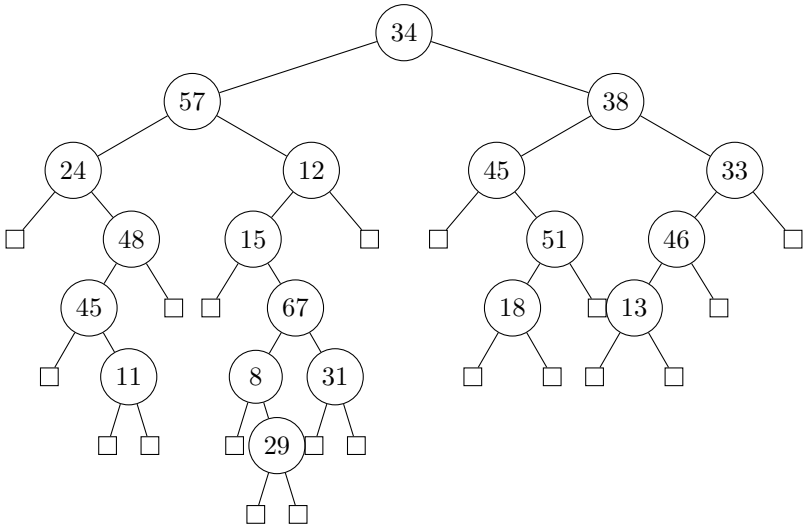
Exercice II.5.1 - Algorithme
Décrire une fonction `decomposition n` qui calcule la suite croissante des diviseurs premiers de n avec répétition et par ordre croissant.

II.5.b Arbres généraux et arbres binaires

Une arbre généralde type α est défini récursivement comme la donnée d'un élément de type α et d'une liste (éventuellement vide) d'arbres de type α, une **forêt**.
Autrement dit une forêt est une liste d'arbres et un arbre est un couple élément, forêt.
On peut transformer une forêt en arbre binaire en associant à une liste d'arbre le nœud dont la racine est la valeur du premier arbre dans la liste, le fils droit est associé à queue de la liste et le fils gauche est associé à la forêt du premier arbre de la liste.



L'exemple ci-dessus devient



Exercice II.5.2 - Taille
Décrire les fonctions `foret2bin` et `bin2foret` qui convertissent une forêt en un arbre binaire homogène (`tree`) et réciproquement.

II.6 Elio Vasseur

II.6.a Minimum local

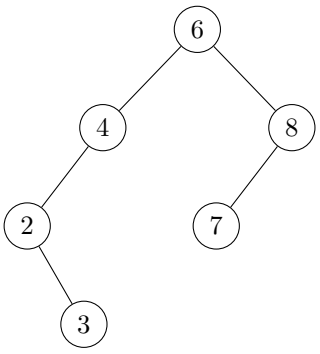
On cherche un minimum local d'un tableau d'entiers de taille n, $t = [|t_0; t_1; t_2; \ldots; t_{n-2}; t_{n-1}|]$ c'est-à-dire une valeur t_i avec i compris entre 0 et $n-1$ telle que t_i est inférieur ou égal à ses voisins ; on remarquera que t_i admet en général 2 voisins mais n'en a qu'un seul si $i = 0$ ou $i = n-1$ ou même aucun dans le cas d'un tableau de taille 1.

Exercice II.6.1 - Séparation
k est un indice tel que $0 < k < n-1$ et t_k n'est pas un minimum local. Prouver que
si $t_k > t_{k-1}$ alors un minimum local de $[|t_0; \ldots; t_{k-1}|]$ est un minimum local du tableau,
si $t_k > t_{k+1}$ alors un minimum local de $[|t_{k+1}; \ldots; t_{n-1}|]$ est un minimum local du tableau.

Exercice II.6.2 - Algorithme rapide
En déduire une fonction `minimum_local` utilisant la méthode diviser-pour-régner qui calcule un minimum local d'un tableau avec une complexité logarithmique (on ne demande pas de justifier la complexité).

II.6.b Nombre de constructions

Exercice II.6.3 - Un exemple
Déterminer tous les ordres possibles d'insertion de clés à partir de l'arbre vide qui donnent l'arbre suivant par adjonction aux feuilles



Exercice II.6.4 - Cas général
Plus généralement montrer que le nombre de permutations des clés d'un arbre T, de taille n, qui engendrent l'arbre par adjonctions successives aux feuilles est $n!$ divisé par le produit des tailles de tous les sous–arbres (y compris T) de T.

III 24 mai 2022

III.1 Théo Applincourt

III.1.a Tours de Hanoï non récursif

On veut déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en utilisant une tour intermédiaire tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand ou sur un emplacement vide.

Exercice III.1.1 - Récursivité
Décrire un algorithme récursif de résolution

Exercice III.1.2 - Complexité
Combien de déplacements sont-ils effectués ?
Montrer que cebombre est le minimum possible.

Exercice III.1.3 - Itération
Décrire un algorithme de résolution utilisant une boucle

III.1.b Une réduction

On dispose d'une fonction `possible(t, k,s)` qui renvoie un booléen signifiant qu'il existe k termes dans le tableau dont la somme est S. On suppose que le tableau ne contient que des entiers positifs et que S est positif

Exercice III.1.4 - Calcul de la somme
Si le résultat de `possible(t, k,s)` est `true`, montrer qu'on obtenir les k termes du tableau dont la somme est S en ne faisant appel qu'à la fonction `possible` et à l'extraction de sous-tableaux.
On pourra remplacer le tableau par une liste chaînée si cela simplifie le raisonnement.

III.2 Dédé Deme

III.2.a Plus long chemin dans un arbre

Dans un arbre binaire il existe un unique chemin simple entre deux nœuds.
On recherche le plus long chemin simple entre deux nœuds.

Exercice III.2.1 - Propriété
Montrer, par un exemple, que ce plus long chemin ne passe pas toujours par la racine.

Exercice III.2.2 - Calcul
Décrire un algorithme qui renvoie le plus long chemin dans un arbre binaire.

III.2.b Tranche maximale

On veut chercher la somme maximale de la forme $S_{a,b} = \sum_{i=a}^{b} a_i$ dans un tableau de nombres (positifs et négatifs) $t = [a_0; a_1; \ldots; a_{n-1}]$ de taille n.

Exercice III.2.3 - Algorithme simple
Donner un algorithme simple de complexité quadratique en n.

Exercice III.2.4 - Diviser pour régner
Décrire un algorithme qui utilise la méthode diviser-pour-régner.
Quelle est sa complexité ?

Exercice III.2.5 - Liste chaînée
Décrire un algorithme récursif de complexité linéaire
Il doit pouvoir s'appliquer à une liste chaînée sans la convertir en tableau.

III.3 Sarah El Fadali

III.3.a Graphes orientés sans circuits

Exercice III.3.1 - Lemme
Prouver que, dans un graphe orienté sans circuit, il existe un sommet de degré sortant nul.

Définition 1 : Ordre topologique

n ordre topologique d'un graphe de taille n est une bijection σ de l'ensemble des sommets vers $\{0, 1, \ldots, n-1\}$ telle que, pour toute arête (i, j) on a $\sigma(i) < \sigma(j)$.

Exercice III.3.2 - Caractérisation
Prouver qu'un graphe orienté est sans circuit si et seulement si il admet un ordre topologique.

III.3.b Somme maximale avec contraintes

On se donne un tableau de nombres positifs $t = [a_0; a_1; \ldots; a_{n-1}]$ de taille n.
On cherche, dans ce tableau d'entiers positifs k éléments **non consécutifs** dont la somme est maximale. Un première solution gloutonne peut être de répéter k fois les deux instructions

- déterminer le maximum du tableau,
- le retirer du tableau ainsi que ses voisins de droite et de gauche.

Exercice III.3.3
Donner un exemple pour lequel cette stratégie ne donne pas un résultat optimal.

On note $g(t, p, k)$ la somme maximale que l'on peut obtenir en choisissant k termes du tableau `t` parmi les p premiers termes de ce tableau, et en respectant, bien sur, la contrainte d'indices séparés de 2 au moins.

Exercice III.3.4
Exprimer une relation de récurrence (sur p) pour les $g(t, p, k)$

Exercice III.3.5
En déduire un algorithme qui donne le résultat optimal.
On pourra commencer par un algorithme récursif puis on l'améliorera avec la programmation dynamique.

III.4 Victor Muzard

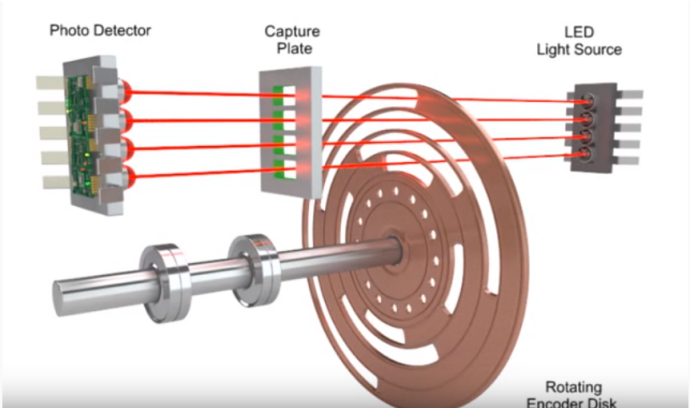
III.4.a Cycle d'ordre 4

Exercice III.4.1
Décrire un algorithme qui détermine si un graphe non orienté $G = (S, A)$ contient un cycle simple d'ordre 4. La complexité doit être un $\mathcal{O}(|S|^3)$.

III.4.b Code de Gray

On veut énumérer, pour chaque p, les entiers de 0 à $2^p - 1$ avec la propriété supplémentaire que deux entiers consécutifs (ainsi que le premier et le dernier) ont des décompositions binaires qui ne diffèrent qu'en un seul bit. Par exemple, pour $p = 3$, le tableau `[|0; 1; 3; 7; 5; 4; 6; 2]` convient parce que les décompositions binaires (sur 3 bits) sont `[0;0;0]`, `[1;0;0]`, `[1;1;0]`, `[1;1;1]`, `[1;0;1]`, `[0;0;1]`, `[0;1;1]`, `[0;1;0]`
La suite des valeurs est appelée une suite de Gray.

Exercice III.4.2
Décrire une méthode permettant de passer de p à $p + 1$.
En déduire un algorithme `gray(p)`



III.5 Martin Ovaere

III.5.a Décomposition de Fibonacci

$(F_n)_{n\in\mathbb{N}}$ est la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$.
Une décomposition de Fibonacci d'un entier $n \geqslant 1$ est une suite $(F_{k_2}, F_{k_2}, \ldots, F_{k_r})$ de nombres de Fibonacci telle que

$$\begin{cases} k_1 \geqslant 2 \\ k_{i+1} \geqslant k_i + 2 \text{ pour } 1 \leqslant i < r \\ n = F_{k_1} + F_{k_2} + \cdots + F_{k_r} \end{cases}$$

Exercice III.5.1 - Décomposition de Fibonacci
Prouver que tout entier n admet une décomposition de Fibonacci unique.
Décrire un algorithme qui renvoie la décomposition d'un entier n sous la forme d'une suite strictement croissante des indices k_i.

III.5.b Caractérisations des arbres complets

Exercice III.5.2 - Nombres de nœuds
Prouver qu'un arbre de hauteur h est complet si et seulement si, pour tout $k \in \{0, 1, \ldots, h\}$, le nombre, N_k, de nœuds de profondeur k est $N_k = 2^k$.

Exercice III.5.3 - Taille
En déduire qu'un arbre de hauteur h est complet si et seulement si sa taille n vérifie $n = 2^{h+1} - 1$.

Exercice III.5.4 - Équilibre strict
Prouver qu'un arbre est complet si et seulement si, pour tout nœud de l'arbre, les deux fils ont la même hauteur.

III.6 Abir Saoud

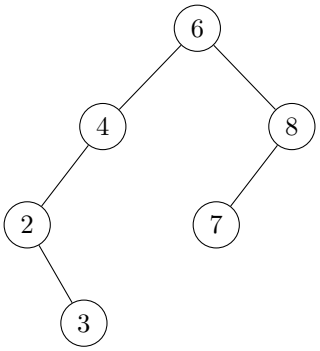
III.6.a Triangle dans un graphe

Dans un graphe non orienté $G = (S, A)$, un triangle est un cycle du graphe à 3 arêtes

Exercice III.6.1
Décrire un algorithme qui détermine si un graphe contient un triangle.
La complexité doit être un $\mathcal{O}(|S|^3)$.

III.6.b Nombre de constructions

Exercice III.6.2 - Un exemple
Déterminer tous les ordres possibles d'insertion de clés à partir de l'arbre vide qui donnent l'arbre suivant par adjonction aux feuilles



Exercice III.6.3 - Cas général
Plus généralement montrer que le nombre de permutations des clés d'un arbre T, de taille n, qui engendrent l'arbre par adjonctions successives aux feuilles est $n!$ divisé par le produit des tailles de tous les sous–arbres (y compris T) de T.

IV 31 mai 2022

IV.1 Quentin Behague

IV.1.a Diamètre d'un graphe

$G = (S, A)$ est un graphe non orienté.
On considère la distance entre deux sommets comme le nombre minimal d'arêtes permettant de les joindre.

Définition 2

L'**excentricité** $e_G(s)$ d'un sommet s est la distance maximale entre s et un sommet de G.
Le **diamètre** $d(G)$ de G est l'excentricité maximale parmi les sommets de G.
Le **rayon** $\rho(g)$ de G est l'excentricité minimale parmi les sommets de G.
Un sommet est un **centre** si son excentricité est égale au rayon.

Exercice IV.1.1 - Exemples
Prouver qu'on a $\rho(G) \leqslant d(G)$; donner un cas où il y a égalité.
Prouver qu'on a $d(G) \leqslant 2 \cdot \rho(G)$; donner un cas où il y a égalité.
Prouver qu'on a $1 \leqslant d(G) \leqslant |S| - 1$; donner des cas où il y a égalité.
Donner un exemple pour lequel il y a un unique centre.
Donner un exemple où tous les sommets sont des centres ; prouver qu'alors $\rho(G) = d(G)$.
Donner un exemple de graphe de diamètre 2 pour lequel tous les sommets sont des centres.

Exercice IV.1.2 - Algorithme
Comment calculer le diamètre, le rayon, les centres ?
Estimer la complexité.

IV.1.b k-combinaisons

On cherche tous les sous ensembles à k éléments de $\{0, 1, 2, \cdots, n-1\}$ sous la forme de liste croissantes en les calculant dans l'ordre lexicographique L'idée est donc de définir la partie suivant une partie donnée.
Par exemple, pour $n = 11$ et $k = 6$,
$(0, 3, 5, 6, 8, 9)$ est suivi de $(0, 3, 5, 6, 8, 10)$, suivi de $(0, 3, 5, 6, 9, 10)$ puis de $(0, 3, 5, 7, 8, 9)$, $(0, 3, 5, 7, 8, 10)$, $(0, 3, 5, 7, 9, 10)$, $(0, 3, 5, 8, 9, 10)$, $(0, 3, 6, 7, 8, 9)$

Exercice IV.1.3 - Algorithme
Décrire un algorithme `suivant(suite, n)` qui calcul le terme suivant.

IV.2 Houssine Reda Bennani

IV.2.a Plus longue sous-suite croissante

Un sous-suite d'une suite de nombre $(a_0, a_1, \ldots, a_{n-1})$ est une suite de la forme $(a_{k_0}, a_{k_1}, \ldots, a_{k_p-1})$ avec $0 \leqslant k_0 < k_1 < \cdots k_{p-1} < n$.

Exercice IV.2.1 - Longueur maximale d'une sous-suite croissante
Décrire un algorithme qui calcule la longueur de la plus longue sous-suite **croissante** d'une suite dont les éléments sont supposés distincts.
L'objectif est de parvenir à une complexité quadratique en la longueur n de la suite.

Exercice IV.2.2 - Sous-suite croissante maximale
Peut-on modifier l'algorithme pour qu'il renvoie une sous-suite croissante de longueur maximale ?

On veut améliorer la complexité.
Pour cela on maintient un tableau m de longueur n tel que m_k est le plus petit élément possible qui soit le dernier élément d'une sous-suite croissante de longueur k.
Par exemple, pour la suite $a = (4, 8, 7, 2, 9, 1, 3)$ on calcule successivement les valeurs de m pour les sous-suites de $(a_0, a_1, \ldots, a_{i-1})$ et on obtient

i	m_0	m_1	m_2	m_3	m_4	m_5	m_6
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	4	8	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	4	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
4	2	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
5	2	7	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
6	1	7	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
7	1	3	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Exercice IV.2.3 - Autre solution
Décrire l'algorithme utilisé.
Après avoir prouvé que la suite m est croissante, montrer qu'on peut obtenir une complexité meilleure qu'un $\mathcal{O}(n^2)$.
Comment obtenir la sous-suite croissante maximale ?

IV.2.b Graphe Eulérien

Définition 3

Un chemin dans un graphe connexe non orienté est **eulérien** s'il est fermé et contient toutes les arêtes une fois et une seule.

Exercice IV.2.4 - Critère
Prouver qu'un graphe admet un chemin eulérien si et seulement tous les degrés des sommets sont pairs.

IV.3 Hugo Martin

IV.3.a Arbres quasi-complets

Un arbre binaire sur E est soit une feuille (vide) soit un nœud qui contient un élément de E, deux fils, droit et gauche, qui sont des arbres binaires sur E.
Un arbre binaire est quasi-complet si toutes les feuilles sont à la profondeur h ou $h-1$ où h est la hauteur de l'arbre, la profondeur maximale des feuilles.

Exercice IV.3.1 - Taille et hauteur
Prouver que si un arbre binaire de hauteur h est quasi-complet alors le nombre de nœuds, n, vérifie $2^{h-1} \leqslant n < 2^h$.

Exercice IV.3.2 - Test
Décrire un algorithme permettant de tester si un arbre binaire est quasi-complet

IV.3.b Pseudo-médiane

Exercice IV.3.3 -
Décrire un algorithme prenant en entrée un tableau et trois indices distincts et retournant la position de la médiane, parmi les trois éléments du tableau dont on a donné les indices.

On considère un tableau de taille 3^k.
On lui applique la procédure suivante :

- on place à la position $3 \cdot i$, à l'aide d'un échange, la médiane des valeurs du tableau aux positions $3 \cdot i$, $3 \cdot i + 1$ et $3 \cdot i + 2$ pour $0 \leqslant i < 3^{k-1}$,
- on place à la position $9 \cdot i$, à l'aide d'un échange, la médiane des valeurs du tableau aux positions $9 \cdot i$, $9 \cdot i + 3$ et $9 \cdot i + 6$ pour $0 \leqslant i < 3^{k-2}$,
- ...
- on place à la position 0, à l'aide d'un échange, la médiane des valeurs du tableau aux positions 0, 3^{k-2} et $2 \cdot 3^{k-2}$.

Exercice IV.3.4 - Algorithme
Décrire une fonction réalisant cette méthode.

Exercice IV.3.5 - Complexité
Déterminer la complexité de la fonction en nombre de comparaisons.

La pseudo-médiane est la valeur finalement affecté à l'indice 0.

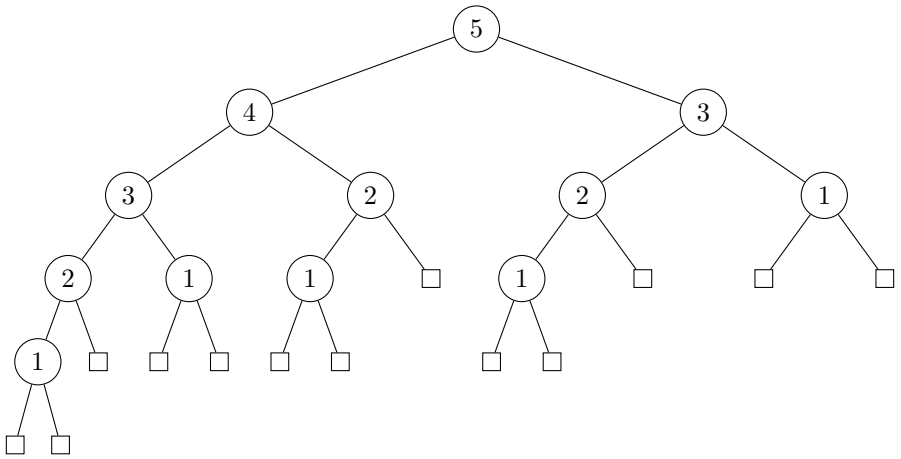
Exercice IV.3.6 - Une valeur centrale
Montrer qu'il existe au moins 2^k éléments majorés par la pseudo médiane.

IV.4 Hadrien Sievers

IV.4.a Arbres de Fibonacci

On définit les arbres de Fibonacci par :

- fibo(0) est l'arbre vide,
- fibo(1) est le nœud de racine 1 et de feuilles vides
- fibo(n+2) est le nœud de racine $n+2$ et de feuilles fibo(n+1) à gauche et fibo(n) à droite.



Exercice IV.4.1 - Construction des arbres de Fibonacci
Écrire une fonction `fibo` telle que `fibo n` renvoie fibo(n).

Exercice IV.4.2 - Hauteur des arbres de Fibonacci
Quelle sont la hauteur et la taille de fibo(n) ?
En déduire qu'un arbre de Fibonacci est équilibré.
Prouver que fibo(n) admet des feuilles à la profondeur n et à la profondeur $\lceil \frac{n}{2} \rceil$ pour $n \geqslant 1$.

IV.4.b Graphe Eulérien

Définition 4

Un chemin dans un graphe connexe non orienté est **eulérien** s'il est fermé et contient toutes les arêtes une fois et une seule.

Exercice IV.4.3 - Critère
Prouver qu'un graphe admet un chemin eulérien si et seulement tous les degrés des sommets sont pairs.

IV.5 Anaëlle Wiersh

IV.5.a Degrés égaux

Exercice IV.5.1
Prouver que, dans un graphe non orienté, il existe deux sommets ayant le même degré.

IV.5.b Produit de matrices

On considère des matrices matrices réelles $\left(B^i\right)_{1\leqslant i\leqslant n}$ avec $B^i \in \mathcal{M}_{k_{i-1},k_i}(\mathbb{R})$.

Exercice IV.5.2 - Complexité
Quelle est la complexité du calcul de $B^i \cdot B^{i+1}$?

Exercice IV.5.3 - Un exemple
On suppose, dans cette question, qu'on a $n = 3$, $k_0 = 2$, $k_1 = 4$, $k_2 = 2$ et $k_3 = 4$ donc $B^1 \in \mathcal{M}_{2,4}(\mathbb{R})$, $B^2 \in \mathcal{M}_{4,2}(\mathbb{R})$ et $B^3 \in \mathcal{M}_{2,4}(\mathbb{R})$.
Quelle est la complexité du calcul de $B^1 \cdot (B^2 \cdot B^3)$ et de $(B^1 \cdot B^2) \cdot B^3$?

On cherche à écrire un programme effectuant le parenthésage optimal sur les B^i pour un calcul efficace de leur multiplication $B^1 \times \cdots \times B^n$.
Pour chaque $1 \leqslant i \leqslant j \leqslant n$, on note $B^{i,j}$ le produit de matrices $B^i \times \cdots \times B^j$, et $p^{i,j}$ le nombre minimum de multiplications de flottants nécessaires pour calculer $B^{i,j}$.
En particulier on a $B^{i,i} = B^i$ et $p^{i,i} = 0$ et aussi $p^{i,i+1} = k_{i-1} \cdot k_i \cdot k_{i+1}$.
Les différentes manières de calculer $B^{i,j}$ doivent finir par un des produits

$$B^{i,i} \times B^{i+1,j},\ B^{i,i+1} \times B^{i+2,j},\ \ldots,\ B^{i,j-2} \times B^{j-1,j} \text{ ou } B^{i,j-1} \times B^{j,j}$$

Exercice IV.5.4 - Formule de récurrence
Déterminer une relation permettant de calculer $p^{i,j}$ en fonction des k_r avec $i - 1 \leqslant r \leqslant j$ et des valeurs des $p^{i,r}$ et des $p^{r+1,j}$ avec $i \leqslant r < j$.

Exercice IV.5.5 - algorithme
En déduire un algorithme qui prend en entrée la suite des tailles $(k_0, \ldots, k_n)$ et renvoie $p^{1,n}$.
On essaiera d'obtenir une complexité polynomiale.

V 7 juin 2022

V.1 Emilien Morfouace

V.1.a Un jeu

Un jeu solitaire consiste à disposer de deux tas non vides de jetons et, à chaque étape, à retirer 2 jetons d'un tas et en ajouter un des deux à l'autre.

Exercice V.1.1 - Fin du jeu
Montrer que le jeu s'arrête.
Quelles sont les dispositions possibles quand le jeu n'est plus possible ?

Exercice V.1.2 - Pronostic
Montrer que l'état final est indépendant de la suite des mouvements et le déterminer en fonction de l'état initial : $_0$ jetons dans le premier tas et q_0 dans le second.

V.1.b Isthme

Un isthme d'un graphe connexe non orienté est une arête dont la suppression déconnecte le graphe.

Exercice V.1.3 - Caractérisation
Soit $e = \{x, y\}$ une arête d'un graphe connexe non orienté G.
Montrer l'équivalence des trois propriétés suivantes.

- (x, y) est un isthme de G.
- (x, y) est le seul chemin simple de G reliant x et y.
- (x, y) n'est inclus dans aucun cycle simple de G.

Exercice V.1.4 - Détection
Décrire un algorithme permettant de de déterminer si une arête est un isthme.
Quelle est sa complexité ?

V.2 Nicolas Morillas

V.2.a Cycles dans un graphe Eulérien

Un graphe non orienté est **Eulérien** si tous ses sommets sont de degré pair non nul.

Exercice V.2.1 -
Décrire un algorithme qui construit un cycle dans un graphe Eulérien.
Donner sa complexité.

V.2.b Racine entière rapide

On définit la fonction racine entière, `isqrt`, par

$$\forall n \in \mathbb{N},\ k = \texttt{isqrt}(n) \iff \big(k \in \mathbb{N} \text{ et } k^2 \leqslant n < (k+1)^2\big)$$

Autrement dit $\texttt{isqrt}(n) = \lfloor \sqrt{n} \rfloor$.

Exercice V.2.2
Prouver que $\texttt{isqrt}(n) = 2\texttt{isqrt}(n \div 4)$ ou $\texttt{isqrt}(n) = 2\texttt{isqrt}(n \div 4) + 1$.

Exercice V.2.3
En déduire un algorithme qui calcule $\texttt{isqrt}(n)$ avec une complexité en $\mathcal{O}\big(\ln(n)\big)$.

V.3 Clément Rouvroy

V.3.a Mots périodique

On considère les mots sur un alphabet (fini) $\mathcal{A}$.
Par exemple $u = aababbba$ est un mot sur $\mathcal{A} = \{a, b\}$.
On peut concaténer deux mots avec la notation $u \cdot v$

Exercice V.3.1 - Propriété
Prouver que si $u \cdot v = v \cdot v$ alors il existe un mot w tel que $u = w^p$ et $v = w^q$.

V.3.b Arbres

On définit un arbre général à valeurs entières par

```
type arbre = Noud of int * arbre list;;
```

On supposera que les valeurs des nœuds sont distinctes.

Exercice V.3.2 - Profondeur
Décrire un algorithme qui permet de calculer la profondeur d'un arbre

Exercice V.3.3 - Noeud de profondeur maximale
Décrire un algorithme qui permet de calculer la valeur d'un nœud de profondeur maximale d'un arbre

Exercice V.3.4 - Chemin depuis la racine
Décrire un algorithme qui permet de calculer le chemin entre la racine et un nœud déterminé par sa valeur.

V.4 Nahel Beldamani

V.4.a Arbres binomiaux

Définition 5 : Arbres binomiaux

Soit k un entier positif ou nul. Un arbre binomial d'ordre k est défini comme suit :

- un arbre binomial d'ordre 0 est réduit à sa racine ;
- si $k > 0$, un arbre binomial d'ordre k est de la forme $\Big(r_k, [\mathcal{T}_{k-1}, \ldots, \mathcal{T}_1, \mathcal{T}_0]\Big)$ où chaque $\mathcal{T}_i$ est un arbre binomial d'ordre i.

Dans la suite, $\mathcal{B}_k$ désigne un arbre binomial d'ordre k.

Exercice V.4.1
Dessiner $\mathcal{B}_4$, avec une numérotation des nœuds de votre choix.

Exercice V.4.2
Quel est le nombre de nœuds de $\mathcal{B}_k$? Combien sont externes ?

Exercice V.4.3
Pour $k > 0$, montrer qu'on peut définir récursivement $\mathcal{B}_k$ à l'aide de deux copies de $\mathcal{B}_{k-1}$.

Exercice V.4.4
Écrire une fonction `copie : int -> arbre -> arbre` telle que `copie n arbre` renvoie une copie de l'arbre dans laquelle chaque nœud de numéro i est remplacé par un nœud de numéro $i + n$.

Exercice V.4.5
Écrire une fonction `bin : int -> arbre` telle que `bin k` renvoie l'arbre $\mathcal{B}_k$, avec une numérotation des nœuds de votre choix.

Exercice V.4.6
Quelle est la profondeur de $\mathcal{B}_k$?
Quelle est la longueur maximale d'un chemin entre deux nœuds ?

V.5 Tom Catoire

V.5.a Découpe de barres

L'entreprise Khoupe achète des barres d'acier et les coupe pour en faire des barres plus courtes qui sont ensuite vendues. Chaque coupe peut se faire où l'on veut et l'entreprise Khoupe veut connaître la meilleure façon de couper les barres.
On suppose que l'on connaît, pour $i = 1, 2 \dots$ le prix p_i que l'entreprise Khoupe facture pour une barre de longueur i m. Les longueurs sont toujours des nombres entiers.
Le tableau suivant donne un exemple de tarifs :

longueur i	1	2	3	4	5	6	7	8	9	10
prix p_i	1	5	8	9	10	17	17	20	22	35

Le problème est le suivant : étant donné une barre de longueur n et un tableau de prix p_i pour $i = 1, 2, \dots, n$, déterminer le revenu optimal r_n que l'on puisse obtenir en coupant la barre et en revendant les morceaux.
Les prix seront donnés sous la forme d'un tableau `prix`.

Exercice V.5.1
Résoudre le problème en fonction de la taille n de la barre.
Quelle est la complexité ?

V.5.b Fonction 91

On définit la fonction f par $f(n) = n - 10$ pour $n \geqslant 100$ et $f(n) = f \circ f(n + 11)$ pour $0 \leqslant n < 100$.

Exercice V.5.2
Calculer $f(n)$.

V.6 Antoine Dumont

V.6.a Arbres

G est un graphe non orienté connexe acyclique (un arbre).

Exercice V.6.1 - Chemin
Montrer qu'il existe un chemin simple (ne passant qu'au plus une fois par chaque sommet) entre deux sommets quelconques et que ce chemin est unique.

Quand on choisit un sommet pour racine, on transforme le graphe en arbre général défini par le type OCaml suivant (on suppose que les sommet du graphe sont des entiers)

```
type arbre. = Noud of int * arbre list;;
```

Exercice V.6.2 - Construction
Décrire un algorithme qui donne l'arbre en fonction du graphe et de la racine.

Le diamètre d'un graphe est la distance maximale entre deux sommets.

Exercice V.6.3 -
On note D le diamètre de G.
Montrer que D est la plus grande profondeur d'un arbre obtenu depuis G et que $\lceil \frac{D}{2} \rceil$ est la plus petite profondeur d'un arbre obtenu depuis G.

On propose l'algorithme suivant :

1. on choisit un sommet s_0 et on définit l'arbre de racine s_0,
2. on choisit un sommet s_1 de profondeur maximale, s_1,
3. la profondeur de l'arbre de sommet s_1 est le diamètre.

Exercice V.6.4 -
Prouver que cet algorithme calcule bien le diamètre.

VI 14 juin 2022

VI.1 William Hasley

VI.1.a Racine entière rapide

On définit la fonction racine entière, `isqrt`, par

$$\forall n \in \mathbb{N},\ k = \texttt{isqrt}(n) \iff \big(k \in \mathbb{N} \text{ et } k^2 \leqslant n < (k+1)^2\big)$$

Autrement dit $\texttt{isqrt}(n) = \lfloor \sqrt{n} \rfloor$.

Exercice VI.1.1
Prouver que $\texttt{isqrt}(n) = 2\texttt{isqrt}(n \div 4)$ ou $\texttt{isqrt}(n) = 2\texttt{isqrt}(n \div 4) + 1$.

Exercice VI.1.2
En déduire un algorithme qui calcule $\texttt{isqrt}(n)$ avec une complexité en $\mathcal{O}\big(\ln(n)\big)$.

VI.1.b Arbres binaires de recherche

Exercice VI.1.3 - Parcours de recherche
On suppose que tous les nombres de 1 à 800 sont les clés d'un arbre binaire de recherche. Parmi le suites suivantes lesquelles ne peuvent pas être des parcours de recherche de 417 ?

- 222, 457, 122, 217, 417
- 545, 215, 333, 401, 422, 417
- 601, 403, 555, 408, 523, 411, 466, 417
- 601, 403, 555, 408, 563, 411, 466, 417
- 901, 403, 555, 411, 466, 417

Exercice VI.1.4 - Fusion
Décrire une fonction `Fusion a b` qui joint deux arbres binaires de recherche ; on pourra utiliser, en en décrivant le principe, la fonctions de coupure d'un arbre binaire de recherche.

Exercice VI.1.5 - Successeurs et prédécesseurs
Montrer que si un nœud d'un arbre binaire de recherche a deux fils alors son prédécesseur n'a pas de fils droit et son successeur n'a pas de fils gauche.

VI.2 Clovis Johnson

VI.2.a Fonction 91

On définit la fonction f par $f(n) = n - 10$ pour $n \geqslant 100$ et $f(n) = f \circ f(n+11)$ pour $0 \leqslant n < 100$.

Exercice VI.2.1
Calculer $f(n)$.

VI.2.b Grands entiers

On veut manipuler des entiers positifs nécessitent une précision qui dépasse celle des entiers de la machine (type `int`).
On se donne pour cela une base de calcul, par exemple `base = 10000`, sa valeur importe peu et on supposera seulement qu'elle est paire, supérieure ou égale à 2 et que son double n'excède pas le plus grand entier machine.
Un entier naturel de précision arbitraire est alors représenté par la liste de ses chiffres en base `base`, les chiffres les moins significatifs étant en tête de liste.
Ainsi la liste $[1;2;3]$ représente l'entier $1 + 2 \cdot \texttt{base} + 3 \cdot \texttt{base}^2$. On définit le type `nat` suivant pour de tels entiers :

```
let base = 10000 ;; (* Par exemple *)
type nat = int list ;;
```

Dans la suite, on garantira l'invariant suivant sur le type `nat` :

- tout élément de la liste est compris entre 0 et `base - 1`, au sens large ;
- le dernier élément de la liste, lorsqu'il existe, n'est pas nul.

On notera que l'entier 0 est représenté par la liste vide.

Exercice VI.2.2 - Addition
Définir une fonction `add_nat` qui calcule la somme de deux grands entiers.

Exercice VI.2.3 - Comparaison
Définir une fonction `cmp_nat` qui prend en argument deux grands entiers n_1 et n_2 et qui renvoie un entier machine valant, -1 si $n_1 < n_2$, 1 si $n_1 > n_2$ et 0 si $n_1 = n_2$.

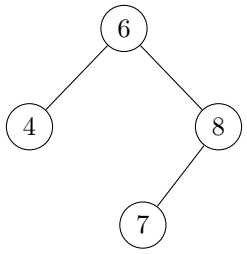
Exercice VI.2.4 - Soustraction
Définir une fonction `sous_nat` qui prend en argument deux grands entiers n_1 et n_2 et qui calcule la différence $n_1 - n_2$ en supposant $n_1 \geqslant n_2$.

Exercice VI.2.5 - Division par 2
Définir une fonction `div2_nat` qui prend en argument un grand entier n et qui calcule le quotient et le le reste dans la division euclidienne de n par 2. Le quotient est un grand entier et le reste un entier machine valant 0 ou 1.
On rappelle que la constante `base` est paire.

VI.3 Ethan Quilliot

VI.3.a Chemin d'Euler

Le **chemin d'Euler** d'un arbre T est la suite $E(T) = (v1, \dots, v_k)$ des étiquettes des nœuds de T rencontrés lors d'un parcours en profondeur de gauche à droite. La liste commence et termine ainsi par la racine, et l'on parcourt les feuilles une seule fois, les nœuds ayant un seul sous-arbre deux fois, et les nœuds ayant deux sous-arbres trois fois.
Par exemple, le chemin d'Euler de l'arbre suivant est $(6, 4, 6, 8, 7, 8, 6)$



Exercice VI.3.1 - Longueur du chemin
Prouver que le chemin d'Euler de T est de longueur $2n - 1$ si n est la taille de T.

Exercice VI.3.2 - Calcul
Décrire un algorithme permettant de calculer le chemin d'Euler.

Exercice VI.3.3 - Profondeurs
Décrire un algorithme permettant de calculer la suite des profondeurs des nœuds dans le chemin d'euler
Dans l'exemple ci-dessus on doit trouver $(0, 1, 0, 1, 2, 1, 0)$.

Exercice VI.3.4 - Plus petit ancêtre commun
Montrer pourquoi la recherche de plus petit ancêtre commun des nœuds d'étiquettes u et us e réduit à la recherche de l'indice du minimum dans la liste des profondeurs entre les indices i et j avec i (resp. j) premier indice où apparaît u (resp. v) dans le parcours d'Euler.

VI.4 Karim Benani

VI.4.a Minimums

Le minimum d'intervalle $\text{mi}(t, i, j)$ d'un tableau t est l'indice du minimum des valeurs de t entre les indices i et j. On cherche à calculer tous les $\text{mi}(t, i, j)$.
n est la longueur du tableau

Exercice VI.4.1 - Calcul basique
Quelle est la complexité en n d'un calcul naïf ?

Exercice VI.4.2 - Programmation dynamique
Déterminer $\text{mi}(t, i, j+1)$ en fonction de $\text{mi}(t, i, j)$.
En déduire une méthode de calculs de tous les $\text{mi}(t, i, j)$ de complexité quadratique.
Quelle est la complexité spatiale. ?

Exercice VI.4.3 - Réduction de la complexité spatiale
Établir une relation entre $\text{mi}(t, i, j)$ et $\text{mi}(t, x, x + 2^k - 1)$ pour $k = \lfloor \log_2(j - i) \rfloor$ et deux valeurs de x bien choisies.

VI.4.b Un jeu

Un jeu solitaire consiste à disposer de deux tas non vides de jetons et, à chaque étape, à retirer 2 jetons d'un tas et en ajouter un des deux à l'autre.

Exercice VI.4.4 - Fin du jeu
Montrer que le jeu s'arrête.
Quelles sont les dispositions possibles quand le jeu n'est plus possible ?

Exercice VI.4.5 - Pronostic
Montrer que l'état final est indépendant de la suite des mouvements et le déterminer en fonction de l'état initial : $_0$ jetons dans le premier tas et q_0 dans le second.

VI.5 Antoine Quint

VI.5.a Produit de matrices

On s'intéresse au produit de deux matrices carrées de taille n.

L'algorithme classique suit la définition mathématique : $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$.

Les matrices sont représentées par des tableaux doubles de flottants.

Exercice VI.5.1 - Produit classique
Décrire une fonction qui calcule ainsi le produit de deux matrices.
Quelle est sa complexité en nombre d'opérations ?

En particulier le produit de deux matrices 2×2 demande 8 multiplications :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} \begin{pmatrix} a'+bc' & ab'+bd' \\ ca'+dc' & cb'+dd' \end{pmatrix}$$

Straßen, en 1969, a proposé une méthode pour calculer ce produit avec 7 multiplications.
On calcule les 7 produits (formules P) :

- $p_1 = (a+d).(a'+d')$
- $p_2 = (a-c).(a'+b')$
- $p_3 = (b-d).(c'+d')$
- $p_4 = a.(b'-d')$
- $p_5 = (a+b).d'$
- $p_6 = (c+d).a'$
- $p_7 = d.(a'-c')$

On les combine avec des additions ou des soustractions (formules C) :

- $aa' + bc' = p_1 + p_3 - p_5 - p_7$
- $ab' + bd' = p_4 + p_5$
- $ca' + dc' = p_6 - p_7$
- $cb' + dd' = p_1 - p_2 + p_4 - p_6$

On suppose écrites des fonctions `somme` et `diff` qui calculent la somme et la différence de matrices carrées de même taille ainsi qu'une fonction `decoupe` qui calcule les 4 blocs carrés d'une matrice carrée d'ordre pair et une fonctions `assemblage` qui reconstitue une matrice à partir de 4 blocs.

Exercice VI.5.2 - Produit rapide de matrices
Décrire une fonction de produit de matrices qui utilise la méthode de Straßen.
On calculera le produit de blocs avec la méthode de Strassen pour ne faire que que 7 produits, ceux-ci étant calculés récursivement.

Exercice VI.5.3 - Complexité
Calculer $m(r)$, le nombre de multiplications, et $a(r)$, le nombre d'additions et soustractions, effectuées dans le produit de deux matrices carrées de taille 2^r.
Pour quelles valeurs de r, le calcul est-il plus efficace que la méthode classique ?

Exercice VI.5.4 - Généralisation
Pour des matrices de taille quelconque on pourra border les matrices par des 0.
Quelle est alors la compleexité ?

VI.6 Majdi Si Salah

VI.6.a Flottants sur 8 bits

On souhaite coder les nombres à l'aide d'un octet seulement, ce n'est pas très réaliste mais illustre les phénomènes. On note $b_0, b_1 \ldots, b_7$ les 8 bits, $b_i \in \{0, 1\}$.
La mantisse est $m = 4 \cdot b_1 + 2 \cdot .b_2 + b_3 \in \{0, 1, \ldots, 7\}$, l'exposant est $e = 8 \cdot b_4 + 4 \cdot b_5 + 2 \cdot .b_6 + b_7 \in \{0, 1, \ldots, 15\}$, le nombre représenté est alors $x = (-1)^{b_0} \cdot \left(1 + \frac{m}{8}\right) \cdot 2^{e-8}$.

Exercice VI.6.1 - Flottants 8 bits

- Que représente $(0, 1, 1, 0, 0, 1, 1, 0)$?
- Quel est le plus petit réel positif représenté, a ?
- Quel est le suivant, b ?
- Quel est le grand réel représenté ?
- Quel est le précédent ?
- Quel est petit entier positif non représenté ?
- Comment est approché $0,1$?
- Comment est approché π ?

On voit que, pour les petites valeurs positives, on a un problème : il y a un écart entre 0 et a bien plus gros qu'entre a et b.

Exercice VI.6.2 - Prolongement vers 0
Proposer une amélioration.

VI.6.b Découpe de barres

L'entreprise Khoupe achète des barres d'acier et les coupe pour en faire des barres plus courtes qui sont ensuite vendues. Chaque coupe peut se faire où l'on veut et l'entreprise Khoupe veut connaître la meilleure façon de couper les barres.
On suppose que l'on connaît, pour $i = 1, 2 \ldots$ le prix p_i que l'entreprise Khoupe facture pour une barre de longueur i m. Les longueurs sont toujours des nombres entiers.
Le tableau suivant donne un exemple de tarifs :

longueur i	1	2	3	4	5	6	7	8	9	10
prix p_i	1	5	8	9	10	17	17	20	22	35

Le problème est le suivant : étant donné une barre de longueur n et un tableau de prix p_i pour $i = 1, 2, \ldots, n$, déterminer le revenu optimal r_n que l'on puisse obtenir en coupant la barre et en revendant les morceaux.
Les prix seront donnés sous la forme d'un tableau `prix`.

Exercice VI.6.3
Résoudre le problème en fonction de la taille n de la barre.
Quelle est la complexité ?

Solutions

Solution de l'exercice I.1.1 - Un codage de Huffman

Solution de l'exercice I.1.2 - Fibonacci 1

Solution de l'exercice I.1.3 - Fibonacci 2

Solution de l'exercice I.2.1 - Haricots rouges et verts
Invariant : la parité du nombre de rouges.

Solution de l'exercice I.2.2 - Maximum d'une suite modale
Invariant : la parité du nombre de rouges.

Solution de l'exercice I.3.1 - Retournement partiel

Solution de l'exercice I.3.2 - Décalage
Échanger de 0 à n-k-1, échanger de n-k à n-1, échanger de 0 à n-1.
Complexité linéaire

Solution de l'exercice I.3.3 - Décalage
Chaque nœud de la profondeur k a deux fils qui sont à la profondeur $k+1$ et qui sont des nœuds ou des racines d'où $N_{k+1} + F_{k+1} = 2.N_k$.
Si l'arbre est réduit à une feuille alors $F_0 = 1$ et $N_0 = 0$, si l'arbre admet au moins un nœud alors la racine est un nœud donc $F_0 = 0$ et $N_0 = 1$.

Si la hauteur est h on a $N_{h+1} = 0$. De plus $N_k = 2N_{k-1} - N_k$ pour $1 \leqslant k \leqslant h+1$ d'où

$$\begin{aligned}\sum_{k=0}^{h+1} F_k 2^{-k} &= F_0 + \sum_{k=1}^{h+1}(2N_{k-1} - N_k)2^{-k} = F_0 + \sum_{k=1}^{h+1} N_{k-1}2^{-k+1} - \sum_{k=1}^{h+1} N_k 2^{-k} \\ &= F_0 + \sum_{k=0}^{h} N_k 2^{-k} - \sum_{k=1}^{h+1} N_k 2^{-k} = F_0 + N_0 - N_{h+1}2^{-h-1} = F_0 + N_0 = 1\end{aligned}$$

Solution de l'exercice I.4.1 - produit par additions et décalages

Solution de l'exercice I.4.2 - Longueur maximale d'une sous-suite croissante
On calcule successivement les l_i, longueur d'une sous-suite croissante maximale finissant par a_i : $li + 1 = 1 + \max\{l_j \ ; \ a_j < a_i\}$ ou 1.

Solution de l'exercice I.4.3 - Sous-suite croissante maximale
On calcule aussi $p_i = j$ valeur du maximum.

Solution de l'exercice I.4.4 - Autre solution
On cherche l'indice j tel que $m_j < a_i < m_{j+1}$ et on remplace m_{j+1} par a_i.
m_k est le dernier élément d'une SSC de longueur k, l'avant dernier est inférieur àm_k, il est le dernier élément d'une SSC de longueur $k-1$ donc supérieur à m_{k-1}. On peut donc faire une recherche par dichotomie et parvenir à une complexité en $\mathcal{O}\big(n \log_2(n)\big)$.
Quand on remplace m_{j+1} par a_i, le prédécesseur de a_i est m_j.

Solution de l'exercice I.5.1 - A.B.R. et tris

Solution de l'exercice I.5.2 - A.B.R. et tas

Solution de l'exercice I.5.3 - Flottants 8 bits

Solution de l'exercice I.5.4 - Prolongement vers 0
On peut remplacer le calcul pour $e = 0$ par $x = (-1)^{b_0} \cdot \dfrac{m}{8} \cdot 2^{-7}$

Solution de l'exercice I.6.1 - Split

Solution de l'exercice I.6.2 - Calcul de LC
La longueur de cheminement des nœuds d'un fils est celle de ces nœuds dans le fils augmentée de 1 pour tout nœud car on doit ajouter la longueur entre la racine du fils et la racine de l'arbre. On va employer une fonction auxiliaire qui renvoie la longueur de cheminement et la taille.

```
let longChem arbre =
  let rec auxLC a =
    match a with
    |Feuille _ -> 0, 0
    |Noeud(g,_,d) -> let lg, ng = auxLC g in
                     let ld, nd = auxLC d in
                     lg + ld + ng + nd, ng + nd + 1
  in fst (auxLC arbre);;
```

Solution de l'exercice I.6.3 - Encadrement de LC
On note $h_1, h_2, \ldots, h_n$ les profondeurs des n nœuds de l'arbre en parcourant ceux-ci par le parcours en largeur. Ces profondeurs forment donc une suite croissante : $h_i \leqslant h_{i+1}$ pour $1 \leqslant i \leqslant n-1$.
Il y a au moins un nœud à chaque profondeur donc l'accroissement entre deux profondeurs successives est majoré par 1 : $h_{i+1} \leqslant h_i+1$. On en déduit, par récurrence sur k, que $h_k \leqslant h_1+k-1 = k-1$.
Ainsi $LC = \sum_{k=1}^{n} h_k \leqslant \sum_{k=1}^{n} k-1 = \dfrac{n(n-1)}{2}$.
On atteint le cas maximal quand $h_k = k-1$ pour tout k, c'est-à-dire s'il y a un seul nœud par profondeur ; c'est le cas des arbres réduit à une liste.

On sait de plus qu'il y a au plus 2^k nœuds à la profondeur k.
On en déduit que h_{i+2^k} vaut au moins $k+1$ si $h_i = k$.
On a $h_1 = 0$ donc $h_2 = h_{1+1} = h_{1+2^0} \geqslant 1$ (en fait h_1 vaut 1).
Si on suppose $h_{2^p} \geqslant p$ alors

- soit $h_{2^p} = p$ donc $h_{2^{p+1}} = h_{2^p+2^p} \geqslant p+1$,
- soit $h_{2^p} \geqslant p+1$ donc $h_{2^{p+1}} \geqslant h_{2^p} \geqslant p+1$.

Dans les deux cas on a $h_{2^{p+1}} \geqslant p+1$ donc, par récurrence sur p, $h_{2^p} \geqslant p$ pour tout $p \in mathbbN$.
On a alors, pour $2^p \leqslant k < 2^{p+1}$, $h_k \geqslant h_{2^p} \geqslant p = \lfloor \log_2(k) \rfloor$ donc
$LC = \sum_{k=1}^{n} h_k \geqslant \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$. Le cas d'égalité sera vu plus tard.

Solution de l'exercice I.6.4 - Caractérisation des arbres quasi-complets
Pour un arbre quasi-complet de taille n, la hauteur est $h = \lfloor \log_2(n) \rfloor$ et il y a 2^k nœuds à la profondeur k pour $k < h$. Il reste $n - \sum_{k=0}^{h-1} 2^k = n - 2^h + 1$ nœuds à la profondeur h.
La longueur de cheminement est ainsi $LC = \sum_{k=0}^{h-1} 2^k.k + h(n-2^h+1)$.
On remarque que $\sum_{k=0}^{h-1} 2^k.k = \sum_{k=1}^{h-1} 2^k.k = P(2)$ avec $P(X) = \sum_{k=1}^{h-1} kX^k = X.Q(X)$
où $Q(X) = \sum_{k=1}^{h-1} kX^{k-1}$ est la dérivée de $\sum_{k=0}^{h-1} X^k = \dfrac{X^h-1}{X-1}$.
Ainsi $P(X) = X\dfrac{hX^{h-1}(X-1)-(X^h-1)}{(X-1)^2}$ donc
$LC = P(2) + h(n-2^h+1) = 2h2^{h-1} - 2(2^h-1) + hn - h2^h + h = (n+1)h - 2^{h+1} + 2$.

Si $h_1, h_2, \ldots, h_n$ sont les profondeurs des n nœuds de l'arbre (quasi-complet) parcouru en largeur. La complétude de l'arbre implique

$h_1 = 0$, $h_2 = h_3 = 1$, …, $h_{2^k} = h_{2^k+1} = \cdots = h_{2^{k+1}-1} = k$, …donc $h_p = \lfloor \log_2(p) \rfloor$.
On en déduit que $LC = \sum_{k=1}^{n} h_k = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$.
Ainsi les arbres quasi-complets réalisent le minimum de la longueur de cheminement.

Inversement si un arbre vérifie $LC = \sum_{k=1}^{n} h_k = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$ alors, comme on a $h_k \geqslant \lfloor \log_2(k) \rfloor$, on doit avoir $h_k = \lfloor \log_2(k) \rfloor$ pour tout k.
On en déduit que $h_k = p$ pour $2^p \leqslant k < 2^{p+1}$ et $k \leqslant n$ donc qu'il y a 2^p nœuds de profondeur p pour toute profondeur atteinte sauf la dernière. Cela caractérise les arbres quasi-complets.

Solution de l'exercice II.1.1 - Tester
On pèse les pièces 1 et 2.
S'il n'y a pas équilibre on sait où est la pièce lourde, si les plateaux sont équilibrés, 3 est la mauvaise pièce.

Solution de l'exercice II.1.2 - Tester
3 réponses possibles par pesée donc 3^p cas distincts avec p pesées : on peut discriminer théoriquement une mauvaise pièce parmi 3^p.

Solution de l'exercice II.1.3 - Tester
Par récurrence su p : on discrimine de paquet de 3^{p-1} pièce contenant la mauvaise pièce en une pesée comme dans le premier exercice.
On peut aussi écrire les numéro des pièces (entre 0 et $3^{p-1} - 1$) en base 3 : $p = \sum_{i=0}^{p-1} t_i 3ß$.
Pour chaque i entre 0 et $p-1$, on place à gauche les pièces avec $t_i = 0$, à droite les pièces avec $t_i = 2$. Si la balance penche à gauche on pose $u_i = 0$, $u_i = 1$ s'il y a équilibre et $u_i = 2$ si la balance penche à droite. La mauvaise pièce est de numéro $\sum_{i=0}^{p-1} u_i 3ß$

Solution de l'exercice II.1.4 - Test de parenthésage

Solution de l'exercice II.1.5 - Association de parentèses

Solution de l'exercice II.1.6 - Généralisations

Solution de l'exercice II.2.1 - Une meilleure solution
$1 + 2 + 2 + 5 + 10 + 10 + 20 + 50 + 100 + 100 + 200 + 500$

Solution de l'exercice II.2.2 - Borne inférieure
Si p est le nombre de sacs, on peut faire au plus 2^p sommes distinctes (2^p est le nombre d'ensembles de sacs possibles. Il faut donc $2^p \geqslant 1001$ donc $p \geqslant 10$.

Solution de l'exercice II.2.3 - Solution optimale
$1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 489$

Solution de l'exercice II.2.4 - Reconstruction
Le parcours préfixe d'un arbre est composé de la racine, du parcours préfixe du fils gauche suivis du parcours préfixe du fils droit.
Le parcours infixe d'un arbre est composé du parcours infixe du fils gauche, de la racine suivis du parcours infixe du fils droit.
On peut donc reconstruire récursivement l'arbre.

1. On sépare la liste préfixe en sa tête, la racine r et le reste `pref_reste`.
2. On sépare le parcours infixe selon r : la liste des termes placés avant r : `inf_gauche` et la liste des termes placés après r : `inf_droit`. r n'est dans aucune des deux listes. On conserve l'ordre initial dans les deux listes.
3. On sépare `pref_reste` en deux listes, les premiers termes dans `pref_gauche` de même longueur que `inf_gauche` et les autres dans `pref_droit`.

4. Le fils gauche est reconstruit à partir de `inf_gauche` et `pref_gauche`.
5. Le fils droit est reconstruit à partir de `inf_droit` et `pref_droit`.

La fonction de découpage selon une valeur renverra aussi la longueur de la première liste.

```
let rec decoupagePivot liste k =
  match liste with
  |[] -> failwith "Le pivot n'est pas dans la liste"
  |t::q when t=k -> [], q, 0
  |t::q -> let g, d, n = decoupagePivot q k in t::g, d, n+1;;
```

Solution de l'exercice II.3.1 - Décalage

Chaque nœud de la profondeur k a deux fils qui sont à la profondeur $k+1$ et qui sont des nœuds ou des racines d'où $N_{k+1} + F_{k+1} = 2.N_k$.
Si l'arbre est réduit à une feuille alors $F_0 = 1$ et $N_0 = 0$, si l'arbre admet au moins un nœud alors la racine est un nœud donc $F_0 = 0$ et $N_0 = 1$.

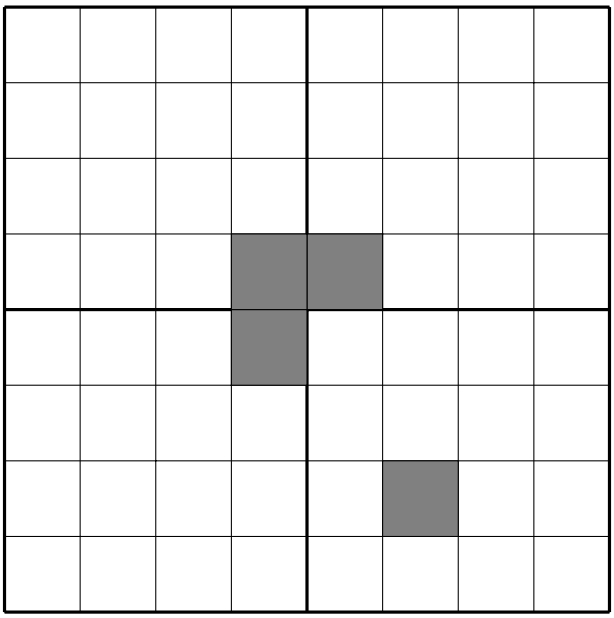
Si la hauteur est h on a $N_{h+1} = 0$. De plus $N_k = 2N_{k-1} - N_k$ pour $1 \leqslant k \leqslant h+1$ d'où

$$\begin{aligned}\sum_{k=0}^{h+1} F_k 2^{-k} &= F_0 + \sum_{k=1}^{h+1}(2N_{k-1} - N_k)2^{-k} = F_0 + \sum_{k=1}^{h+1} N_{k-1}2^{-k+1} - \sum_{k=1}^{h+1} N_k 2^{-k} \\ &= F_0 + \sum_{k=0}^{h} N_k 2^{-k} - \sum_{k=1}^{h+1} N_k 2^{-k} = F_0 + N_0 - N_{h+1}2^{-h-1} = F_0 + N_0 = 1\end{aligned}$$

Solution de l'exercice II.3.2 - Itération de l'angle

Solution de l'exercice II.4.1 - Algorithme

Par récurrence sur n.
C'est immédiat pour $n = 1$.
Sinon on se ramèneà $n-1$ en recouvrant le carré central des 3 petits plateaux ne contenant pas la case marquée par un trimino.



Solution de l'exercice II.4.2 - Le cas classique
Ici utilise deux variables, k_1 et k_2.
les éléments d'indices entre 0 et $k_1 - 1$ valent a
les éléments d'indices entre k_1 et $k_2 - 1$ valent b
les éléments d'indices entre k_2 et $i - 1$ valent c

```
let drapeau t b =
  let n = Array.length t in
  let k1 = ref 0 in
  let k2 = ref 0 in
  for i = 0 to (n-1) do
    if t.(i) < b
     then begin
          echanger t i !k2;
          echanger t !k1 !k2
          incr k1;
          incr k2 end
     else if t.(i) = b
          then (echanger t i !k2; incr k2) done;;
```

Solution de l'exercice II.4.3 - Quicksort

```
let decomposition t a b =
  let pivot = t.(b) in
  let k1 = ref a in
  let k2 = ref a in
  for i = a to b do
    if t.(i) < pivot
    then begin
      echanger t i !k2;
      echanger t !k1 !k2;
      incr k1;
      incr k2 end
    else if t.(i) = pivot
    then (echanger t i !k2; incr k2) done;
  !k1, !k2;;
```

```
let triPivot t =
  let rec tri a b =
  if  a < b
  then begin
       let k1, k2 = decomposition t a b in
       tri a (k1 - 1);
       tri k2 b end in
  tri 0 (Array.length t -1);;
```

Le tri est plus rapide pour des tableaux contenant beaucoup de valeurs multiples.

Solution de l'exercice II.5.1 - Algorithme

```
let decomposition n =
  let rec dec k p =
    if p =1
    then []
    else if p mod k =0
    then k :: (dec k (p/k))
    else dec (k+1) p in
  dec 2 n;;
```

Solution de l'exercice II.5.2 - Taille

```
let rec foret2bin f =
  match f with
  |[] -> Vide
  |N(r,fils)::ff -> Noeud(foret2bin fils, r, foret2bin ff);;

let rec bin2foret a =
  match a with
  |F -> []
  |Noeud(g,r,d) -> N(r, bin2foret g) :: (bin2foret d);;
```

Solution de l'exercice II.6.1 - Séparation
Un minimum local de $[|t_0; \ldots; t_{k-1}|]$ est un minimum local du tableau sauf s'il est atteint en t_{k-1}. Mais comme on a $t_{k-1} < t_k$, t_{k-1} reste un minimum local. De même pour l'autre cas.

Solution de l'exercice II.6.2 - Algorithme rapide
On écrit une fonction auxiliaire qui reçoit les bornes dans lesquelles chercher un minimum local. Les cas terminaux sont ceux où il reste un seul élément ou deux car on va chercher au milieu et celui-ci ne doit pas être une des bornes.

```
let minimum_local tab =
  let rec aux_ml i j =
    if j > i + 1
    then let k = (i+j)/2 in
         if tab.(k) > tab.(k-1)
         then aux_ml i (k-1)
         else if tab.(k) > tab.(k+1)
              then aux_ml (k+1) j
              else tab.(k)
    else if j = i + 1
         then min tab.(j) tab.(i)
         else tab.(i)
  in aux_ml 0 (Array.length tab -1);;
```

Solution de l'exercice II.6.3 - Un exemple
Les ordres d'insertions possibles sont :

```
6, 4, 2, 3, 8, 7
6, 4, 2, 8, 3, 7
6, 4, 2, 8, 7, 3
6, 4, 8, 2, 3, 7
6, 4, 8, 2, 7, 3
6, 4, 8, 7, 2, 3
6, 8, 4, 2, 3, 7
6, 8, 4, 2, 7, 3
6, 8, 4, 7, 2, 3
6, 8, 7, 4, 2, 3
```

Solution de l'exercice II.6.4 - Cas général
Les ordres d'insertions possibles sont : On prouve la propriété par récurrence sur la taille n.
La propriété est simple pour $n = 1$: il y a une seule manière de construire l'arbre.
On suppose que la propriété est vraie pour tous les arbres (non vides) de taille $k < n$.
`T = Noeud(g, x, s)` est un arbre de taille n avec g de taille p et d de taille q.
Les sous-arbres de T sont les sous-arbres de d, les sous-arbres de g et T.
On suppose d'abord que g et d sont non vides.
Pour engendrer T on doit commencer par x puis on doit engendrer d et g en choisissant un remplissage de chacun et l'ordre d'insertion des élément à droite et à gauche.

- Il y a $\dfrac{p!}{\prod_{t\in g}|t|!}$ façon d'engendrer g.
- Il y a $\dfrac{q!}{\prod_{t\in d}|t|!}$ façon d'engendrer d.
- Pour distribuer les insertions à droite et à gauche on doit choisir les p occurrences d'ajout à gauche parmi les $p+q$: il y a donc $\dfrac{(p+q)!}{p!q!} = \dfrac{(n-1)!}{p!q!}$ telles distributions.

On trouve donc $\dfrac{p!}{\prod_{t\in g}|t|!}\dfrac{q!}{\prod_{t\in d}|t|!}\dfrac{(n-1)!}{p!q!} = \dfrac{(n-1)!}{\prod_{t\in g}|t|!\prod_{t\in d}|t|!} = \dfrac{n!}{|T|\prod_{t\in g}|t|!\prod_{t\in d}|t|!} = \dfrac{n!}{\prod_{t\in T}|t|!}$.
La propriété est donc vraie pour T.
Si d (ou g) est vide on construit T en insérant x puis les éléments de g (ou d) : il y a le même nombre de construction et $\dfrac{(n-1)!}{\prod_{t\in g}|t|!} = \dfrac{n!}{|T|\prod_{t\in g}|t|!} = \dfrac{n!}{\prod_{t\in T}|t|!}$
La propriété est encore vraie pour T, c'est-à-dire pour tous les arbres de taille n.
On a prouvé la récurrence.

Solution de l'exercice III.1.1 - Récursivité

Solution de l'exercice III.1.2 - Complexité

Solution de l'exercice III.1.3 - Itération

Solution de l'exercice III.1.4 - Calcul de la somme

Solution de l'exercice III.2.1 - Propriété

Solution de l'exercice III.2.2 - Calcul

Solution de l'exercice III.2.3 - Algorithme simple

Solution de l'exercice III.2.4 - Diviser pour régner

Solution de l'exercice III.2.5 - Liste chaînée

Solution de l'exercice III.3.1 - Lemme
On suppose que, dans un graphe orienté, tous les sommets ont degré sortant non nul.
n est le nombre de sommets. Chaque sommet admet, au moins, un voisin.
On part d'un sommet s_0 : il admet un voisin s_1.
On construit ainsi un chemin de $n+1$ sommets en ajoutant, à chaque étape, un voisin du sommet précédent $s_0 \to s_1 \to \cdots \to s_n$.
Parmi ces $n+1$ sommets, deux aux moins sont égaux donc le chemin contient un circuit.
La contraposée du résultat prouvé est le lemme.

Solution de l'exercice III.3.2 - Caractérisation

- On suppose que G admet un ordre topologique.
  S'il admettait un circuit $s_0 \to s_1 \to \cdots \to s_{p-1} \to s_0$ alors on devrait avoir
  $\sigma(s_0) < \sigma(s_1) < \cdots < \sigma(s_{p-1}) < \sigma(s_0)$ ce qui est impossible : G est sans circuit.
- On prouve le sens direct par récurrence sur la taille, n, de G.
  Un graphe de taille 1 est sans circuit et admet un ordre topologique.
  On suppose que les graphes de taille n sans circuit admettent un ordre topologique.
  Si G est de taille $n+1$ et est sans circuit, il admet un sommet sans degré sortant d'après le lemme. On le note s et on pose $S' = S \setminus \{s\}$.
  Le graphe induit par S' est de taille n et n'admet pas de circuit.
  D'après l'hypothèse de récurrence il admet un ordre topologique σ' de S' vers $\{0, 1, \ldots, n-1\}$. On prolonge σ' sur S par $\sigma(s) = n$.
  σ est un ordre topologique sur S car les seules arêtes de G qui ne sont pas dans le graphe induit sont de la forme (t, s) avec $t \in S'$ donc $\sigma(t) = \sigma'(t) \leqslant n-1 < n = \sigma(s)$.
  On a donc un ordre topologique pour tout graphe orienté sans circuit.

Solution de l'exercice III.3.3

```
let t0 = [|2; 17; 19; 18|];;
```

Solution de l'exercice III.3.4

- $g(t, p, k)$ n'a de sens que pour $2k-1 \leqslant p < n$, n étant la longueur de t,
- $g(t, p, 0) = 0$,
- $g(t, 2k-1, k) = g(t, 2k-3, k-1) + t_{2k-1}$,
- $g(t, p, k) = \max\big(g(t, p-1, k), g(t, p-2, k-1) + t_{p-1}$ pour $p > 2k-1$.

Solution de l'exercice III.3.5

```
let somme_max2_rec t k =
    let rec aux p k =
      if k = 0  then 0, []
      else if p = 2*k - 1
           then let s, l = aux (p-2) (k-1) in s + t.(p-1), (p-1)
                :: l
           else let s1, l1 = aux (p-2) (k-1) in
                let s2, l2 = aux (p-1) k in
                if s1 + t.(p-1) > s2
                then s1 + t.(p-1), (p-1) :: l1
                else s2, l2
   in aux (Array.length t) k;;
```

```
let somme_max2_dyn t k0 =
    let n = Array.length t in
    let mem = Array.make_matrix (n+1) (k0+1) 0 in
    for k = 1 to k0 do
    if k = 1 then mem.(1).(1) <- t.(0)
    else mem.(2*k-1).(k) <- t.(2*k-2) + mem.(2*k-3).(k-1);
        for m = 2*k to n do
            mem.(m).(k) <- max mem.(m-1).(k) (t.(m-1) + mem.(m-2)
               .(k-1))
        done done;
    mem.(n).(k0);;
```

Si on veut aussi renvoyer la liste des indices :

```
let somme_max2_dyn t k0 =
    let n = Array.length t in
    let mem = Array.make_matrix (n+1) (k0+1) 0 in
    let pris = Array.make_matrix (n+1) (k0+1) false in
    for k = 1 to k0 do
    if k = 1 then mem.(1).(1) <- t.(0)
    else mem.(2*k-1).(k) <- t.(2*k-2) +  mem.(2*k-3).(k-1);
        for m = 2*k to n do
            let s1 = mem.(m-1).(k) in
            let s2 = t.(m-1) + mem.(m-2).(k-1) in
            mem.(m).(k) <- max s1 s2;
            if s2 > s1 then pris.(m).(k) <- true done done;
    let rec solution m k =
        if k = 0
        then []
        else if pris.(m).(k)
             then  (m-1)::(solution (m-2) (k-1))
             else solution (m-1) k in
  mem.(n).(k0), solution n k0;;
```

On a utilisé une matrice de booléens pour minimiser la place mémoire.

Solution de l'exercice III.4.1

Solution de l'exercice III.4.2

Solution de l'exercice III.5.1 - Décomposition de Fibonacci

Solution de l'exercice III.5.2 - Nombres de nœuds
Si $h = -1$, l'arbre est vide et il n'y a rien à prouver.
On suppose $h \geqslant 0$.
Un arbre est complet si et seulement si le nombre de feuille de profondeur k, F_k, vérifie $F_k = 0$ pour $k \leqslant h$.
Comme on a $N_{k+1} + F_{k+1} = 2N_k$ pour tout $k \geqslant 0$ on en déduit qu'un arbre est complet si et seulement si $N_{k+1} = 2N_k$ pour tout $k \leqslant h-1$. Comme on a $N_0 = 1$ c'est donc équivalent à $N_k = 2^k$ pour $k \leqslant h$.

Solution de l'exercice III.5.3 - Taille
On sait qu'on a $N_k \leqslant 2^k$ donc la taille n vérifie $n = \sum_{k=0}^{h} N_k \leqslant \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$ avec égalité si et seulement si $N_k = 2^k$ pour tout k, c'est-à-dire si et seulement si l'arbre est complet.

Solution de l'exercice III.5.4 - Équilibre strict
On va procéder par récurrence sur la hauteur de l'arbre.
$\mathcal{P}(h)$ est la propriété : un arbre de hauteur h est complet si et seulement si, pour tout nœud de l'arbre, les deux fils ont la même hauteur.
$\mathcal{P}(-1)$ est vérifiée car tous les arbres de hauteur -1 sont des feuilles donc sont complets et vérifient toute propriétés sur leurs fils car ils n'ont pas de fils.
On suppose que $\mathcal{P}(h)$ est vérifiée avec $h \geqslant -1$.
a est un arbre de hauteur $h+1$, ses fils droit et gauche sont notés d et g respectivement.

- Si a est complet alors ses feuilles, donc les feuilles de g et d, sont à la profondeur $h+2$. Ainsi les feuilles de g sont à la profondeur $h+1$ dans g donc g est complet ; de même d est complet.
  g et d sont de hauteur h au plus mais ils admettent des feuilles à la profondeur $h+1$ donc ils sont de hauteur h au moins. Ainsi g et h sont complets de hauteur h.
  On peut appliquer l'hypothèse de récurrence : tous les nœuds de g et d ont des fils de même hauteur. Comme g et d, fils de la racine de a, ont même hauteur on en déduit que tous les nœuds de a ont des fils de même hauteur.
- Si tous les nœuds de a ont des fils de même hauteur alors g et d ont même hauteur h et ont des nœuds qui ont des fils de même hauteur. D'après l'hypothèse de récurrence g et d sont complets : leurs feuilles sont à la profondeur $h+1$.
  Les feuilles de a sont les feuilles de g et d et sont à la profondeur augmentée de 1 : $h+2$. On en déduit que a est complet.

Ainsi $\mathcal{P}(h+1)$ se déduit de $\mathcal{P}(h)$ donc $\mathcal{P}(h)$ est vraie pour tout h.

Solution de l'exercice III.6.1

Solution de l'exercice III.6.2 - Un exemple
Les ordres d'insertions possibles sont :

```
6, 4, 2, 3, 8, 7
6, 4, 2, 8, 3, 7
6, 4, 2, 8, 7, 3
6, 4, 8, 2, 3, 7
6, 4, 8, 2, 7, 3
6, 4, 8, 7, 2, 3
6, 8, 4, 2, 3, 7
6, 8, 4, 2, 7, 3
6, 8, 4, 7, 2, 3
6, 8, 7, 4, 2, 3
```

Solution de l'exercice III.6.3 - Cas général
Les ordres d'insertions possibles sont : On prouve la propriété par récurrence sur la taille n.
La propriété est simple pour $n = 1$: il y a une seule manière de construire l'arbre.
On suppose que la propriété est vraie pour tous les arbres (non vides) de taille $k < n$.
`T = Noeud(g, x, s)` est un arbre de taille n avec g de taille p et d de taille q.
Les sous-arbres de T sont les sous-arbres de d, les sous-arbres de g et T.
On suppose d'abord que g et d sont non vides.
Pour engendrer T on doit commencer par x puis on doit engendrer d et g en choisissant un remplissage de chacun et l'ordre d'insertion des élément à droite et à gauche.

- Il y a $\dfrac{p!}{\prod_{t \in g} |t|!}$ façon d'engendrer g.
- Il y a $\dfrac{q!}{\prod_{t \in d} |t|!}$ façon d'engendrer d.

- Pour distribuer les insertions à droite et à gauche on doit choisir les p occurrences d'ajout à gauche parmi les $p+q$: il y a donc $\dfrac{(p+q)!}{p!q!} = \dfrac{(n-1)!}{p!q!}$ telles distributions.

On trouve donc $\dfrac{p!}{\prod_{t\in g}|t|!}\dfrac{q!}{\prod_{t\in d}|t|!}\dfrac{(n-1)!}{p!q!} = \dfrac{(n-1)!}{\prod_{t\in g}|t|!\prod_{t\in d}|t|!} = \dfrac{n!}{|T|\prod_{t\in g}|t|!\prod_{t\in d}|t|!} = \dfrac{n!}{\prod_{t\in T}|t|!}$.
La propriété est donc vraie pour T.
Si d (ou g) est vide on construit T en insérant x puis les éléments de g (ou d) : il y a le même nombre de construction et $\dfrac{(n-1)!}{\prod_{t\in g}|t|!} = \dfrac{n!}{|T|\prod_{t\in g}|t|!} = \dfrac{n!}{\prod_{t\in T}|t|!}$
La propriété est encore vraie pour T, c'est-à-dire pour tous les arbres de taille n.
On a prouvé la récurrence.

Solution de l'exercice IV.1.1 - Exemples
2 sommets ou graphe complet.
Graphe linéaire avec un nombre impair de sommets.
Graphe complet puis graphe linéaire.
Graphe linéaire avec un nombre impair de sommets.
Graphe complet

Solution de l'exercice IV.1.2 - Algorithme

Solution de l'exercice IV.1.3 - Algorithme

```
let suivant t n =
  let k = Array.length t in
  let i = ref (k-1) in
  while !i >= 0 && t.(!i) = n - k + !i do decr i done;
  if !i < 0
  then false
  else begin
    t.(!i) <- t.(i) + 1;
    for j = (i+1) to (k-1) do
      t.(j) <- lt.(j-1) + 1 done;
    true end;;
```

Pour des listes, il est plus simple de les écrire à l'envers

```
let rec suivant l n =
  match l with
  |[] -> []
  |t::q when t = (n-1)
    -> begin match suivant q (n-1) with
       |[] -> []
       |s::r -> (s+1) :: s :: r end
  |t::q -> (t+1)::q;;
```

Solution de l'exercice IV.2.1 - Longueur maximale d'une sous-suite croissante
On calcule successivement les l_i, longueur d'une sous-suite croissante maximale finissant par a_i : $li+1 = 1+\max\{l_j\ ;\ a_j < a_i\}$ ou 1.

Solution de l'exercice IV.2.2 - Sous-suite croissante maximale
On calcule aussi $p_i = j$ valeur du maximum.

Solution de l'exercice IV.2.3 - Autre solution
On cherche l'indice j tel que $m_j < a_i < m_{j+1}$ et on remplace m_{j+1} par a_i.
m_k est le dernier élément d'une SSC de longueur k, l'avant dernier est inférieur àm_k, il est le dernier élément d'une SSC de longueur $k-1$ donc supérieur à m_{k-1}. On peut donc faire une recherche par dichotomie et parvenir à une complexité en $\mathcal{O}\big(n\log_2(n)\big)$.
Quand on remplace m_{j+1} par a_i, le prédécesseur de a_i est m_j.

Solution de l'exercice IV.2.4 - Critère
Dans un cycle eulérien on associe 2 arêtes à chaque apparition d'un sommet donc le degré est pair.
On procède par récurrence sur le nombre d'arêtes.
Une arête est impossible car les 2 sommets sont de degré 1.
2 arêtes est impossible aussi car alors il y a 3 sommets dont deux de degré 1.
Pour 3 arêtes le nombre de sommets d'un graphe connexe est au plus 4 ; 4 est impossible car la somme des degrés est 6 à partager en 4 sommets, il ne peut pas y avoir 4 sommets de degré 2 au moins. Pour 3 sommets on a un triangle, le cycle eulérien est évident.
Si tous les sommets sont de degré pair on peut construire un cycle simple car on peut toujours prolonger un chemin : on finit par boucler.
Si on retire les arêtes du cycle c et les sommets qui deviendraient isolés, on parvient
soit à un graphe vide et on a un cycle eulérien
soit à un graphe connexe dont les sommets sont de degré pair ou au graphe vide. d'où l'existence d'un cycle eulérien. La connexité impose qu'au moins un des sommets de c est resté dans le graphe et on peut alors ajouter c pour contenir toutes les arêtes.

Solution de l'exercice IV.3.1 - Taille et hauteur
Il y a 2^k nœuds à la profondeur k pour $0 \leqslant k < h-1$.

Solution de l'exercice IV.3.2 - Test

```
type 'a arbre = V | N of 'a arbre * 'a * 'a arbre;;

let rec cp h arbre =
  match arbre with
  |V -> h = 0
  |N(g, _, d) -> (cp (h-1) g) && (cp (h-1) d);;

let rec ht arbre =
  match arbre with
  |V -> 0
  |N(g, _, d) -> 1 + (max (ht g) (ht d));;

let qc arbre =
  let rec aux k a =
    match a with
    |V -> k = 0
    |N(g, _, d) -> (aux (k-1) g) && (aux (k-1) d)
                   || (aux (k-1) g) && (cp (k-2) d)
                   || (cp (k-2) g) && (aux (k-1) d) in
  aux (ht arbre) arbre;;
```

Solution de l'exercice IV.3.3 -

```
let medianePlace t a b c=
  if t.(a) < t.(b)
  then if t.(b) < t.(c)
       then b
       else if t.(a) < t.(c) then c else a
  else if t.(a) < t.(c)
      then a
      else if t.(b) < t.(c) then c else b;;
```

Solution de l'exercice IV.3.4 - Algorithme

```
let pseudo_mediane t =
  let rec auxMed a b =
    if b=a+2
    then let i = mediane t a (a+1) b in echange a i t
    else let r =(b+1-a)/3 in
         begin
           auxMed a         (a + r - 1);
           auxMed (a + r)   (a + 2*r - 1);
           auxMed (a + 2*r) b;
           let i = mediane t a (a + r) (a + 2*r) in
           echange a i t end in
  auxMed 0 (Array.length t -1);
```

Solution de l'exercice IV.3.5 - Complexité
Linéaire

Solution de l'exercice IV.3.6 - Une valeur centrale
Par récurrence

- Pour un tableau de taille $3^0 = 1$ la pseudo-médiane est l'unique valeur :
  la propriété est vraie.
- On suppose la propriété vraie pour les tableaux de taille 3^k.
  Si on cherche la pseudo médiane d'un tableau de taille 3^{k+1} alors on le divise en 3 tableaux de taille 3^k. Pour chacun d'eux la pseudo médiane est minorée par au moins 2^k éléments.
  La pseudo médiane de l'ensemble est la médiane des 3 pseudo médianes des sous-tableaux, elle est minorée par 2 d'entre elles donc par au moins $2^k + 2^k = 2^{k+1}$ éléments : la propriété vraie pour les tableaux de taille 3^{k+1}.

Solution de l'exercice IV.4.1 - Construction des arbres de Fibonacci

```
let rec fibo n =
  match n with
  |0 -> Vide
  |1 -> Noeud(Vide,1,Vide)
  |n _> Noeud(fibo (n-1), n, fibo (n-2));;
```

Solution de l'exercice IV.4.2 - Hauteur des arbres de Fibonacci

- Si h_n est la hauteur de fibo(n) on a $h_{n+2} = 1 + \max(h_{n+1}, h_n)$ avec $h_0 = -1$ et $h_1 = 0$.
  On en déduit, par récurrence sur n que $h_n = n - 1$. fibo(n) admet donc une feuille à la profondeur n.
  Ainsi les hauteurs des fils différent de 1 pour tout nœud, sauf ceux qui sont des fibo(1) dont les deux fils ont la même hauteur -1 ; un arbre de Fibonacci est donc équilibré.

- Si n_p est la taille de fibo(p) on a $n_{p+2} = 1 + n_{p+1} + n_p$ avec $n_0 = 0$ et $n_1 = 1$.
  Si on pose $N_p = n_p + 1$ on a $N_{p+2} = N_{p+1} + N_p$ avec $N_0 = 1$ et $N_1 = 2$; on retrouve la récurrence de la suite de Fibonacci avec $N_0 = F_1$ et $N_1 = F_2$ donc $N_p = F_{p+1}$ et $n_p = F_{p+1} - 1$.
- Si F_n a une feuille à la profondeur $k = \lceil \frac{n}{2} \rceil$ alors F_{n+2}, qui admet F_n comme fils droit, a une feuille à la profondeur $k + 1 = \lceil \frac{n}{2} \rceil + 1 = \lceil \frac{n+2}{2} \rceil$.
  On en déduit le résultat souhaité par récurrence sur n car fibo(1) a une feuille à la profondeur $1 = \lceil \frac{1}{2} \rceil$ et fibo(2) a une feuille à la profondeur $1 = \lceil \frac{2}{2} \rceil$.

Solution de l'exercice IV.4.3 - Critère
Dans un cycle eulérien on associe 2 arêtes à chaque apparition d'un sommet donc le degré est pair.
On procède par récurrence sur le nombre d'arêtes.
Une arête est impossible car les 2 sommets sont de degré 1.
2 arêtes est impossible aussi car alors il y a 3 sommets dont deux de degré 1.
Pour 3 arêtes le nombre de sommets d'un graphe connexe est au plus 4 ; 4 est impossible car la somme des degrés est 6 à partager en 4 sommets, il ne peut pas y avoir 4 sommets de degré 2 au moins. Pour 3 sommets on a un triangle, le cycle eulérien est évident.
Si tous les sommets sont de degré pair on peut construire un cycle simple car on peut toujours prolonger un chemin : on finit par boucler.
Si on retire les arêtes du cycle c et les sommets qui deviendraient isolés, on parvient
soit à un graphe vide et on a un cycle eulérien
soit à un graphe connexe dont les sommets sont de degré pair ou au graphe vide. d'où l'existence d'un cycle eulérien. La connexité impose qu'au moins un des sommets de c est resté dans le graphe et on peut alors ajouter c pour contenir toutes les arêtes.

Solution de l'exercice IV.5.1
Par l'absurde : si ce n'est pas le cas alors, comme le degré est compris entre 0 et $n-1$, tous les degrés sont atteints. Il y a donc un sommet de degré 0 donc sans voisin et un sommet de degré $n-1$ donc voisin de tous. Il y a contradiction.

Solution de l'exercice IV.5.2 - Complexité
$k_{i-1} \cdot k_i \cdot k_{i+1}$.

Solution de l'exercice IV.5.3 - Un exemple
On commence par calculer $B^2 \times B^3$ qui demande $4 \cdot 2 \cdot 4 = 32$ multiplications et donne une matrice de $B^{2,3} \in \mathcal{M}_{4,4}(\mathbb{R})$ puis $B^1 \times B^{2,3}$ qui demande $2 \cdot 4 \cdot 4 = 32$ multiplications d'où un total de 64 multiplications.
$(B^1 \times B^2) \times B^3$ demande $2 \cdot 4 \cdot 2 + 2 \cdot 2 \cdot 4 = 32$ multiplications.

Solution de l'exercice IV.5.4 - Formule de récurrence
Les calculs de $B^{i,r}$ et $B^{r+1,j}$ demandent respectivement $p^{i,r}$ et $p^{r+1,j}$ multiplications et sont de taille $k_{i-1} \times k_r$ et $k_r \times k_j$. Leur produit demandera $k_{i-1} \cdot k_r \cdot k_j$ multiplications en plus.
On a donc $p^{i,j} = \text{Min}\big(p^{i,r} + p^{r+1,j} + k_{i-1} \cdot k_r \cdot k_j \ ; \ i \leqslant r \leqslant j-1\big)$.

Solution de l'exercice IV.5.5 - algorithme

```
let min_mul k =
  let n = Array.length k in
  let rec pm i j =
    if i = j
    then 0
    else begin
      let p0 = ref ((pm i (j-1)) + k.(i-1)*k.(j-1)*k.(j)) in
      for r = i to (j-2) do
        let p = (pm i r)+(pm (r+1) j)+k.(i-1)*k.(r)*k.(j) in
        p0 := min !p0 p done;
      !p0 end in
  pm 1 (n-1);;
```

La complexité de `pm i j` ne dépend que $h = j - i$, on le note $C(h)$.
On a $C(0) = 0$, $C(1) = 4$ (une affectation, une addition et 2 multiplications) et, pour $h \geqslant 2$
$C(h) = C(h-1) + 3 + \sum_{k=0}^{h-2} \big(C(k) + C(h-k-1) + 6\big) = 6h - 3 + 2\sum_{k=1}^{h-2} C(k) + C(h-1)$.
On a donc $C(h+1) - C(h) = C(h) + C(h-1) + 6$ pour $h \geqslant 2$,
donc $C(h) = A \cdot (1+\sqrt{2})^h + B \cdot (1-\sqrt{2})^h + 3$ pour $h \geqslant 2$ avec $A > 0$ et $1 + \sqrt{2} > 2$.
La complexité est exponentielle.

```
let min_mul k =
  let n = Array.length k in
  let p = Array.make_matrix n n 0 in
  for h = 1 to (n-2) do
    for i = 1 to (n-1-h) do
      let j = i + h in
      p.(i).(j) <- p.(i).(j-1) + k.(i-1)*k.(j-1)*k.(j);
      for r = i to (j-2) do
      let pp = p.(i).(r)+p.(r+1).(j)+k.(i-1)*k.(r)*k.(j) in
        p.(i).(j) <- min p.(i).(j) pp done done done;
  p.(1).(n-1);;
```

La complexité est cubique.

Solution de l'exercice V.1.1 - Fin du jeu
On enlève un jeton à chaque tour.
(1, 0), (0, 1) ou (1, 1)

Solution de l'exercice V.1.2 - Pronostic
$p - q \mod 3$ est un invariant.

Solution de l'exercice V.1.3 - Caractérisation
On note G' le graphe obtenu en ôtant l'arête (x, y) : $G' = (S, A \setminus \{(x, y)\})$.

- 1) ⇒ 2) Si (x, y) est un isthme et s'il existait un chemin simple de x à y distinct de (x, y) alors ce chemin est de longueur au moins 2 donc ne peut pas comporter l'arête (x, y) car il est simple : c'est un chemin dans G'. En remplaçant (x, y) par ce chemin on voit que deux sommets équivalents dans g sont encore équivalents dans G' donc G' est connexe, ce qui est exclu.
  (x, y) est le seul chemin simple de G reliant x et y.
- 2) ⇒ 1) Si (x, y) est le seul chemin de G reliant x et y alors x et y ne sont pas équivalent dans G' donc G' n'est pas connexe.
- 2) ⇔ 3) L'existence d'un cycle simple contenant (x, y) est équivalente à l'existence d'un chemin simple entre x et y distinct de (x, y).

Solution de l'exercice V.1.4 - Détection
On calcule le nombre de composantes connexes avant et après avoir enlevé l'arête : $\mathcal{O}\big(|S| + |A|\big)$.

Solution de l'exercice V.2.1 -

Solution de l'exercice V.2.2
On note $r' = \texttt{isqrt}(n \div 4)$. On a $r'^2 \leqslant n \div 4 < (r'+1)^2$ donc, comme tous les nombres sont entiers, $r'^2 \leqslant n \div 4 \leqslant (r'+1)^2 - 1$.
De plus on a $n \div 4 \leqslant \frac{n}{4} < n \div 4 + 1$ d'où,
$r'^2 \leqslant n \div 4 \leqslant \frac{n}{4} < n \div 4 + 1 < (r'+1)^2 - 1 + 1$ c'est-à-dire $r'^2 \leqslant \frac{n}{4} < (r'+1)^2$.
On en déduit : $4r'^2 \leqslant n < 4(r'+1)^2$ donc $2r' \leqslant \lfloor\sqrt{n}\rfloor = \texttt{isqrt}(n) < 2r' + 2$.
On peut en déduire que $\texttt{isqrt}(n) = 2r'$ ou $\texttt{isqrt}(n) = 2r' + 1$.
Dans les cas on aboutit à $\texttt{isqrt}(n) \div 2 = r' = \texttt{isqrt}(n \div 4)$.

Solution de l'exercice V.2.3

```
let rec isqrt n =
  if n = 0 then 0
           else let r  = isqrt (n/4) in
                let rr = 2*r + 1 in
                if rr*rr > n then rr - 1
                            else rr;;
```

Solution de l'exercice V.3.1 - Propriété

Solution de l'exercice V.3.2 - Profondeur

```
let profondeur t =
  let rec prof_aux liste =
    match liste with
    |[] -> 0
    |(Noeud(k, fils)) :: freres
      -> max (1 + prof_aux fils) (prof_aux freres) in
  let Noeud(k, ll) = t in
  prof_aux ll;
```

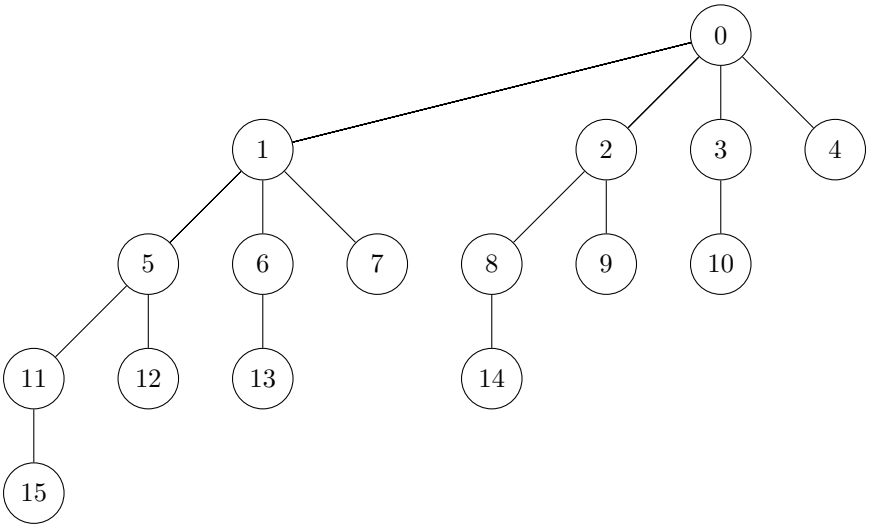
Solution de l'exercice V.3.3 - Noeud de profondeur maximale

```
let noeud_externe_max t =
  let rec noeud_aux liste =
    match liste with
    |[] -> (0, -1)
    |[Noeud (n, [])] -> (1, n)
    |Noeud (n, fils) :: freres ->
        let (p1, n1) = noeud_aux(fils) in
        let (p2, n2) = noeud_aux(freres) in
        if p1 + 1 > p2 then (p1 + 1, n1) else (p2, n2) in
  let Noeud (k, ll) = t in
  snd (noeud_aux ll);;
```

Solution de l'exercice V.3.4 - Chemin depuis la racine

```
let chemin t s =
  let rec chem_aux s liste =
    match liste with
    |[] -> []
    |Noeud(n, fils) :: freres
      -> if n = s
         then [s]
         else let chem = chem_aux s fils in
              if chem = []
              then chem_aux s freres
              else n::chem in
  chem_aux s [t];;
```

Solution de l'exercice V.4.1



Solution de l'exercice V.4.2
$\mathcal{B}_0$ a un nœud, qui est externe.
Si on note n_k (resp. e_k) le nombre de nœuds (resp. de nœuds externes) de $\mathcal{B}_k$ on a $n_{k+1} = 1 + n_0 + n_1 + \cdots + n_k$ et $e_{k+1} = e_0 + e_1 + \cdots + e_k$.
Comme on a $1 + n_0 + n_1 + \cdots + n_{k-1} = n_k$ on en déduit $n_{k+1} = 2n_k$ pour $k \geqslant 1$ avec $n_1 = 2$ donc $n_k = 2^k$ pour tout k.
De même, $e_k = 2e_{k-1}$ avec $e_2 = 1$ donc $e_k = 2^{k-1}$ pour $k \geqslant 1$.

Solution de l'exercice V.4.3
Si $\mathcal{B}_k = (r_k, l)$ où l est la liste des fils alors $\mathcal{B}_{k+1} = (r_{k+1}, \mathcal{B}_k :: l)$.

Solution de l'exercice V.4.4
Pour modifier les fils d'un arbre on travaille sur des listes d'arbres. Les fonction récursives auxiliaires prendront comme arguments des listes d'arbres.

```
let copie n t =
  let rec aux n liste
  match liste with
  |[] -> []
  |Noeud(k, ll)::q ->   Noeud(k+n, (aux n ll)) :: (aux n q) in
  List.hd (aux n [t]);;
```

Solution de l'exercice V.4.5

```
let rec bin k =
  match k with
  |0 -> Noeud (1, [])
  |_ -> let a = bin (k-1) in
        let Noeud(n, ll) = a in
        Noeud(2*n, (copie n a) :: a);;
```

Solution de l'exercice V.4.6
La profondeur de $\mathcal{B}_k$ est égale aux maximum des profondeur des $\mathcal{B}_i$ pour $i < k$, augmentée de 1 par récurrence elle est égale à k.

Comme la profondeur de $\mathcal{B}_k$ est k tout chemin est de longueur $2k$ au plus.
Par récurrence on montre aussi qu'il n'y a qu'un nœud à la profondeur k donc tout chemin est de de longueur $2k-1$ au plus.
Si on considère le nœud à la profondeur $k-1$ de $\mathcal{B}_{k-1}$ et le nœud à la profondeur $k-2$ de $\mathcal{B}_{k-2}$ le chemin qui les lie dans $\mathcal{B}_k$ est de longueur $k-1+1+1+k-2$ car il passe par la racine : la longueur maximale est $2k-1$.

Solution de l'exercice V.5.1

Solution de l'exercice V.5.2
$f(n) = 91$ pour $n \leqslant 101$, $f(n) = n - 10$ sinon

Solution de l'exercice V.6.1 - Chemin
La connexité donne l'existence, deux chemins permettraient de construire un cycle.

Solution de l'exercice V.6.2 - Construction

```
let arbre n g =
  let rec aux_arbre k pere =
    let liste = enlever pere (voisins g k) in
    Noeud (k, foret liste k)
  and foret liste p =
    match liste with
    |[] -> []
    |t::q -> (aux_arbre t p) :: (foret q p) in
  aux_arbre n n;;
```

Solution de l'exercice V.6.3 -
a et b sont deux sommets à distance D.
Le sommet de racine a admet b à la profondeur D et les autres sommets sont à profondeur moindre.
Ainsi il y a un arbre de profondeur D.
La profondeur d'un arbre est la distance entre deux sommets, elle est donc majorée par D.
Ainsi D est la profondeur maximale.

Dans un arbre tout chemin consiste à remonter puis descendre dans l'arbre : sa longueur est donc majorée par deux fois la profondeur. C'est vrai en particulier d'un chemin maximal d'où
$D \leqslant 2 \cdot \text{profondeur}(\mathcal{T})$ c'est-à-dire $\frac{D}{2} \leqslant \text{profondeur}(\mathcal{T})$.
Comme $\text{profondeur}(\mathcal{T})$ est un entier il majore aussi $\lceil \frac{D}{2} \rceil$: $\lceil \frac{D}{2} \rceil \leqslant \text{profondeur}(\mathcal{T})$ pour tout arbre.

Si on choisit la racine au milieu d'un chemin entre a et b, en le coupant en deux segments de longueurs respectives $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$ on obtient un arbre de profondeur au moins $\lceil \frac{n}{2} \rceil$. De plus il n'existe pas de nœud à une profondeur strictement supérieure car sinon on pourrait construire un chemin de longueur strictement supérieure : la profondeur $\lceil \frac{n}{2} \rceil$ est atteinte.
Ainsi $\lceil \frac{D}{2} \rceil$ est la profondeur minimale.

Solution de l'exercice V.6.4 -
On désigne par $d(n_1, n_2)$ la longueur du chemin de n_1 à n_2 dans le graphe.
$\mathcal{T}_0$ est l'arbre de sommet s_0, $\mathcal{T}_1$ est l'arbre de sommet s_1.
On considère deux sommets distincts, a et b, s_0 est un ancêtre commun à a, b et s_1. On peut donc considérer un ancêtre commun de profondeur maximale, c.
Pour tout nœud x descendant de c la profondeur de x dans $\mathcal{T}_0$, notée $p_0(x)$ vaut $p_0(c) + d(c, x)$.
On a $c \neq s_1$ car s_1 est de profondeur maximale donc ne peut être ancêtre d'aucun point.

- Si $c = a$ alors $d(a,b) = d(c,b) = p(b) - p(c) \leqslant p(s_1) - p(c) = d(a, s_1)$ car s_1 est de profondeur maximale. Or, dans $\mathcal{T}_1$, $d(a, s_1) = p_1(a)$ où p_1 est la profondeur dans $\mathcal{T}_1$ donc $d(a,b) \leqslant d(a, s_1) = p_1(a) \leqslant \text{prof}(\mathcal{T}_1)$.

- De même si $c = b$ alors $d(a,b) \leqslant \text{prof}(\mathcal{T}_1)$.
- Si $a \neq c$ et $b \neq c$ alors au moins un des deux nœuds a ou b ne peut appartenir au même fils de c car sinon ce fils serait un ancêtre commun de profondeur strictement supérieure, ce que contredit la définition de c.

  On suppose, par exemple que a et s_1 appartiennent à des fils distincts de c.

  Le chemin de s_1 à a dans $\mathcal{T}_1$ passe par c d'où

  $d(a,c) + d(c,b) \leqslant d(a,c) + d(c,s_1) = p_1(a) \leqslant \text{prof}(\mathcal{T}_1)$.

  On considère alors un ancêtre commun à a et b descendant de c, c', de profondeur minimale. a et b appartiennent à deux fils distincts de c' donc la chemin de a à b passe par c' et

  $d(a,b) = d(a,c') + d(c',b) \leqslant d(a,c) + d(c,b) \leqslant \text{prof}(\mathcal{T}_1)$

Dans tous les cas on a bien $d(a,b) \leqslant \text{prof}(\mathcal{T}_1)$ donc le diamètre de $\mathcal{T}$ (et de $\mathcal{T}_1$) est majoré par la profondeur de $\mathcal{T}_1$.

Solution de l'exercice VI.1.1
On note $r' = \texttt{isqrt}(n \div 4)$. On a $r'^2 \leqslant n \div 4 < (r'+1)^2$ donc, comme tous les nombres sont entiers, $r'^2 \leqslant n \div 4 \leqslant (r'+1)^2 - 1$.
De plus on a $n \div 4 \leqslant \frac{n}{4} < n \div 4 + 1$ d'où,
$r'^2 \leqslant n \div 4 \leqslant \frac{n}{4} < n \div 4 + 1 < (r'+1)^2 - 1 + 1$ c'est-à-dire $r'^2 \leqslant \frac{n}{4} < (r'+1)^2$.
On en déduit : $4r'^2 \leqslant n < 4(r'+1)^2$ donc $2r' \leqslant \lfloor \sqrt{n} \rfloor = \texttt{isqrt}(n) < 2r' + 2$.
On peut en déduire que $\texttt{isqrt}(n) = 2r'$ ou $\texttt{isqrt}(n) = 2r' + 1$.
Dans les cas on aboutit à $\texttt{isqrt}(n) \div 2 = r' = \texttt{isqrt}(n \div 4)$.

Solution de l'exercice VI.1.2

```
let rec isqrt n =
  if n = 0 then 0
           else let r  = isqrt (n/4) in
                let rr = 2*r + 1 in
                if rr*rr > n then rr - 1
                             else rr;;
```

Solution de l'exercice VI.1.3 - Parcours de recherche

Solution de l'exercice VI.1.4 - Fusion

Solution de l'exercice VI.1.5 - Successeurs et prédécesseurs

Solution de l'exercice VI.2.1
$f(n) = 91$ pour $n \leqslant 101$, $f(n) = n - 10$ sinon

Solution de l'exercice VI.2.2 - Addition
On peut commencer par gérer l'addition de 1 :

```
let rec retenue = function
  |[] -> [1]
  |c::n -> if (c+1) < base
           then (c+1)::n
           else 0::(retenue n);;
```

Puis appliquer le calcul "comme à l'école" :

```
let rec add_nat n m =
  match (n,m) with
  |([],m) -> m
  |(n,[]) -> n
  |(c::n',d::m') ->
      if (c+d) < base
      then (c+d)::(add_nat n' m')
      else (c+d-base)::(retenue (add_nat n' m'));;
```

Solution de l'exercice VI.2.3 - Comparaison
On peut commencer par écrire la comparaison de deux entiers

```
let comp_int c d =
 if c < d then -1
          else if c > d then 1 else 0;;
```

La comparaison de deux liste s'écrit alors de manière récursive : si les restes sont distincts ils donnent l'ordre, si ils sont égaux c'est le premier terme qui ordonne.

```
let rec comp_nat n m =
  match (n,m) with
  |([],[]) -> 0
  |([],m) -> -1
  |(n,[]) -> 1
  |(c::n',d::m') ->  let r =comp_nat n' m' in
                     if r = 0
                     then comp_int c d
                    else r;;
```

Solution de l'exercice VI.2.4 - Soustraction
Il ne faut pas oublier d'éliminer une fin de liste ne comportant que des 0.
On peut utiliser une construction

```
let cons_nat c n =
  if (c,n) = (0,[])
  then []
  else c::n;;
```

```
let rec sous_nat n m =
  match (n,m) with
  |(n,[]) -> n
  |([],_) -> failwith ("Soustraction impossible")
  |(c::n',d::m') ->  if (c-d) < 0
                     then cons_nat (base+c-d)
                                   (sous_nat n' (retenue m'))
                     else cons_nat (c-d) (sous_nat n' m');;
```

Solution de l'exercice VI.2.5 - Division par 2
On écrit un nombre sous la forme $a + \beta b$ où β est la base.
$a = 2a' + r$ avec $r \in \{0, 1\}$ et $b = 2b' + s$ avec $s \in \{0, 1\}$.
Si $s = 0$ le quotient et le reste sont $a' + \beta b'$ et r,
si $s = 1$ alors, comme β est pair, le quotient et le reste sont $a' + \frac{1}{2}\beta + \beta b'$ et r.

```
let rec div2_nat = function
  |[] -> ([],0)
  |c::n ->  let (q,r) = div2_nat n in
            if r= 0
            then (cons_nat (c/2) q,c mod 2)
            else (cons_nat ((c+base)/2) q,c mod 2);;
```

Solution de l'exercice VI.3.1 - Longueur du chemin

- $l = 1$ si $n = 1$,
- $l = 1 + l_1 + 1$ où l_1 est la longueur du chemin d'Euler du fils si l'arbre a un seul fils, dans ce cas $n_1 = n - 1$ donc $l = 1 + 2(n-1) - 1 + 1 = 2n - 1$
- $l = 1 + l_g + 1 + l_d + 1$ où l_g et l_d sont les longueurs des chemins d'Euler des fils donc $l = 3 + 2n_g - 1 + 2n_d - 1 = 2(n_g + n_d + 1) - 1 = 2n - 1$

Solution de l'exercice VI.3.2 - Calcul

Solution de l'exercice VI.3.3 - Profondeurs

Solution de l'exercice VI.3.4 - Plus petit ancêtre commun

Solution de l'exercice VI.4.1 - Calcul basique
Cubique

Solution de l'exercice VI.4.2 - Programmation dynamique

Solution de l'exercice VI.4.3 - Réduction de la complexité spatiale

Solution de l'exercice VI.4.4 - Fin du jeu
On enlève un jeton à chaque tour.
(1, 0), (0, 1) ou (1, 1)

Solution de l'exercice VI.4.5 - Pronostic
$p - q \mod 3$ est un invariant.

Solution de l'exercice VI.5.1 - Produit classique

```
let produit_matrices a b =
  (* Les matrices doivent avoir la même taille *)
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      let s = ref 0.0 in
      for k = 0 to (n-1) do
        s:= !s +. a.(i).(k) *. b.(k).(j) done;
      c.(i).(j) <- !s done done;
  c;;
```

$2n^3$

Solution de l'exercice VI.5.2 - Produit rapide de matrices

```
let somme a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) +. b.(i).(j) done done;
  c;;
```

```
let diff a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) -. b.(i).(j) done done;
  c;;
```

```
let decoupe m =
  let n = (Array.length m)/2 in
  let m1 = Array.make_matrix n n 0.0 in
  let m2 = Array.make_matrix n n 0.0 in
  let m3 = Array.make_matrix n n 0.0 in
  let m4 = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m1.(i).(j) <- m.(i).(j);
      m2.(i).(j) <- m.(i).(j+n);
      m3.(i).(j) <- m.(i+n).(j);
      m4.(i).(j) <- m.(i+n).(j+n) done done;
  m1, m2, m3, m4;;
```

```
let assemblage m1 m2 m3 m4 =
  let n = Array.length m1 in
  let m = Array.make_matrix (2*n) (2*n) 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m.(i).(j) <- m1.(i).(j);
      m.(i).(j+n) <- m2.(i).(j);
      m.(i+n).(j) <- m3.(i).(j);
      m.(i+n).(j+n) <- m4.(i).(j) done done;
  m;;
```

```
let rec prod a b =
  let n = Array.length a in
  if n = 1
  then [|[|a.(0).(0) *. b.(0).(0)|]|]
  else let a1, a2, a3, a4 = decoupe a in
       let b1, b2, b3, b4 = decoupe b in
       let p1 = prod (somme a1 a4) (somme b1 b4) in
       let p2 = prod (diff a1 a3) (somme b1 b2) in
       let p3 = prod (diff a2 a4) (somme b3 b4) in
       let p4 = prod a1 (diff b2 b4) in
       let p5 = prod (somme a1 a2) b4 in
       let p6 = prod (somme a3 a4) b1 in
       let p7 = prod a4 (diff b1 b3) in
       let c1 = diff (somme p1 p3) (somme p5 p7) in
       let c2 = somme p4 p5 in
       let c3 = somme p6 p7 in
       let c4 = somme (diff p1 p2) (diff p4 p6) in
       assemblage c1 c2 c3 c4;;
```

Solution de l'exercice VI.5.3 - Complexité
La méthode de STraßen effectuée 18 additions et soustractions.
On a $m(0) = 1$ et $m(r+1) = 7m(r)$ donc $m(r) = 7^r$.
On a $a(0) = 0$ et $a(r+1) = 7a(r) + 18 \cdot (2^r)^2$ car on calcule 18 fois la somme ou le produit de matrices carrées d'ordre 2^r. La partie $a(r+1) = 7a(r)$ permet de poser $a(r) = 7^r.u_r$.
On obtient alors $u_{r+1} = u_r + \frac{18}{7} \cdot \left(\frac{4}{7}\right)^r$ d'où

$$u_r = u_0 + \frac{18}{7} \cdot \sum_{k=0}^{r-1} \left(\frac{4}{7}\right)^k = \frac{18}{7} \cdot \frac{1 - \left(\frac{4}{7}\right)^r}{1 - \frac{4}{7}} = 6 \cdot \left(1 - \left(\frac{4}{7}\right)^r\right).$$

Ainsi $a(r) = 6 \cdot (7^r - 4^r)$.

Au total on effectue donc $7^{r+1} - 6.4^r$ opérations alors que la méthode classique en demande $2 \cdot (2^r)^3 = 2^{3r+1}$. La méthode classique est plus efficace pour $r \leqslant 9$; pour $r = 10$, $n = 1024$, le nombre d'opérations est de l'ordre de $2 \cdot 10^9$.

Solution de l'exercice VI.5.4 - Généralisation
On a cherché 2^p tel que $2^{p-1} < n \leqslant 2^p$.
On a vu qu'alors le nombre d'opérations est $7^{p+1} - 6.4^p \leqslant 7 \cdot 7^p$.
Or on a $p \leqslant \log_2(n) + 1$ donc le nombre d'opérations est majoré par $14 \cdot 7^{\log_2(n)} = 17. \cdot n^{\log_2(7)}$, c'est donc un $\mathcal{O}\left(n^{\log_2(7)}\right)$ avec $\log_2(7) \simeq 2,807....$ La complexité asymptotique est meilleure.

Solution de l'exercice VI.6.1 - Flottants 8 bits

Solution de l'exercice VI.6.2 - Prolongement vers 0
On peut remplacer le calcul pour $e = 0$ par $x = (-1)^{b_0} \cdot \frac{m}{8} \cdot 2^{-7}$

Solution de l'exercice VI.6.3