- Notes de cours -

Présentation du module

Objectifs du cours. Ce cours fait suite au cours d'algorithmique de L2. Son but est de poursuivre l'étude de la conception et de l'analyse d'algorithmes. Les problèmes algorithmiques que nous étudierons viendront de différents domaines : traitement informatique, recherche oprérationnelle, logique, graphes, algèbre, combinatoire...

Intervenants. S. Bessy (responsable du module), S. Daudé, M. Mari (mail : `prenom.nom@umontpellier.fr`)

Contenu. Quatre chapitres principaux :
- **Algorithmes exhaustifs et backtrack**,
- **Analyse en moyenne et algorithmes probabilistes**,
- **Hachage** et
- **Algorithmes d'approximation**.

Planning (a priori...) Les cours (10 séances d'1h30) ont lieu le mercredi à 13h15, les tds et tps (10 séances de 3h) le mardi à 8h00 ou le jeudi matin à 9h45 (selon votre groupe).
L'emploi du temps exact est à consulter sur l'ENT.

Sem.	Date	Contenu
37	Me 13/09	**CM1 :** *Chapitre 1 :* **Complexité et Probabilités : aléatoire en algorithmique**
38	Me 20/09	**CM2 :** *Chapitre 2 :* **Recherche exhaustive et backtrack** **TD1 :** Complexité, aléatoire en algorithmique
39	Me 27/09	**CM3 :** Fin du *Chapitre 2* **TP1 :** Algorithmes probabilistes
40	Me 04/10	**CM4 :** *Chapitre 3 :* **Analyse amortie, en moyenne et algorithmes probabilistes** **TD2 :** Recherche exhaustive
41	Me 11/10	**CM5 :** Fin du *Chapitre 3* **TP2 :** Recherche exhaustive
42	Me 18/10	**CM6 :** *Chapitre 4 :* **Hachage** **TD3 :** Analyse amortie, en moyenne et algos probabilistes
43	Me 25/10	RIEN
44	Me 02/11	VACANCES D'AUTOMNE
45	Me 08/11	**CM7 :** Fin du *Chapitre 4* **TD4 :** Fin des Algos probabilistes et début des exos sur le Hachage
46	Me 15/11	**CM8 :** **!!! Controle continu, type exam !!!** (Contenu : Chap. 1,2 et 3) **TD5 :** Fin des exercices sur le Hachage
47	Me 22/11	**CM9 :** *Chapitre 5 :* **Algorithmes d'approximation** **TP3 :** Hachage et **TP4 :** Algorithmes d'approximation
48	Me 29/11	**CM10 :** Fin du *Chapitre 5* **TD6 :** Algorithmes d'approximation
49	Me 5/12	**TP-SWERC :** Concours de programmation

Modalité de contrôle de connaissance. L'évaluation comportera un contrôle continu et un examen. Le contrôle continu est constitué à 90% d'une évaluation sur table (type exos de TD, sur le créneau de cours du 15 novembre a priori) et à 10% d'une note sur l'avancée en TP. Le contrôle continu compte pour 30% dans la note finale avec la 'règle du max', c'est-à-dire que la note finale sera $\max\{exam; 0.7 \times exam + 0.3 \times cc\}$. Une deuxième session d'examen est prévue, pas de contrôle continu.
Pour les exams et le contrôle continu, les seuls documents autorisés sont ces notes de cours, annotées à loisir.

Prérequis. D'un point de vue algorithmique, sont attendues des connaissances sur les instructions classiques en pseudo-code, la récursivité, les heuristiques classiques de complexité (glouton, diviser pour régner, programmation dynamique), le calcul de la complexité d'un algorithme et la connaissance des structures de données usuelles (tableaux, piles, files, liste chaînées, structures arborescentes).
En programmation, les tp se feront en Python. Il faut maîtriser a minima le langage.

Ressource. Les ressources pédagogiques seront disponibles sur le moodle de l'Université (HAI503I). Les ouvrages suivant contiennent l'essentiel du cours (et même plus...) :

- T.H. Cormen, C.E. Leiserson, R. Rivest and C. Stein. **Introduction to Algorithms**, 3rd Edition, *MIT Press*, 2009 (une version française existe).
- S. Dasgupta, C. Papadimitriou and U. Vazirani. **Algorithms**, *McGraw-Hill Higher Education*, 2006 (version électronique gratuite).
- J. Erickson. **Algorithms**, *University of Illinois at Urbana-Champaign*, 2019, ainsi que le chapitre supplémentaire sur les probabilités (version électronique gratuite).
- J. Kleinberg, É. Tardos. **Algorithm design**, *Pearson education*, 2005.
- V.V. Vazirani. **Approximation algorithms**, *Springer*, 2001.

1 Rappels de complexité, probabilité

1.1 Rappels de complexité

- **Une notation de Landau** : Soient $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$. On dit que $\boldsymbol{f = O(g)}$ si il existe une constante $c > 0$ et un rang $n_0 \in \mathbb{N}$ tels que : $\forall n \geq n_0$ on ait $f(n) \leq c.g(n)$

Lemme 1 (O et limites) *Pour $f : \mathbb{N} \longrightarrow \mathbb{R}$ et $g : \mathbb{N} \longrightarrow \mathbb{R}^{+*}$, si il existe une constante $c \geq 0$ telle que $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} c$ alors on a $f = O(g)$. Et si $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} +\infty$ alors on a $f \neq O(g)$.*

Lemme 2 (Limite de l'inverse) *Si f et g sont des fonctions strictements positives (c'est-à-dire, que pour tout $n \in \mathbb{N} \setminus \{0, 1\}$, on a $f(n) > 0$ et $g(n) > 0$), alors on a :*
Si il existe une constante $c > 0$ telle que $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} c$, alors $\frac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} \frac{1}{c}$, **et donc** $f = O(g)$ **et** $g = O(f)$**.**
Si $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} 0$ alors $\frac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} \infty$, **et donc** $f = O(g)$ **et** $g \neq O(f)$**.**
Et si $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} \infty$ alors $\frac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} 0$, **et donc** $f \neq O(g)$ **et** $g = O(f)$**.**

Lemme 3 (Limites comparées) *- Pour $\alpha, \beta > 0$ on a :*
$\frac{(\log n)^\alpha}{n^\beta} \underset{n\to+\infty}{\longrightarrow} 0$ $\quad$ $\frac{n^\alpha}{(2^n)^\beta} \underset{n\to+\infty}{\longrightarrow} 0$ $\quad$ $\frac{(2^n)^\alpha}{n!} \underset{n\to+\infty}{\longrightarrow} 0$
- Soit $u(n)$ est une fonction telle que $u(n) \underset{n\to+\infty}{\longrightarrow} \infty$ (typiquement, $u(n) = n$ ou $u(n) = \log n$ ou $u(n) = 2^n$).
Si $a.X^p$ et $b.X^q$ avec $a > 0$ et $b > 0$ sont respectivement les termes de plus haut degré de deux polynomes P et Q alors $\lim_{n\to\infty} P(u(n))/Q(u(n))$ vaut : 0 si $q > p$, a/b si $q = p$ et $+\infty$ si $p > q$.

Théorème 1 ('Master Theorem') *Si il existe trois entiers $a \geq 0$, $b > 1$ et $d \geq 0$ tels que pour tout $n > 0$ on ait $t(n) \leq a.t(\lceil n/b \rceil) + O(n^d)$ alors :*
- $t(n) = O(n^d)$ *si $b^d > a$ (c'est-à-dire si $d > \log a / \log b$),*
- $t(n) = O(n^d \log n)$ *si $b^d = a$ (c'est-à-dire si $d = \log a / \log b$), ou*
- $t(n) = O(n^{\frac{\log a}{\log b}})$ *si $b^d < a$ (c'est-à-dire si $d < \log a / \log b$).*

Lemme 4 (Règles de calcul pour le log**)** *Pour $a, b \in \mathbb{R}^{+*}$ et $c \in \mathbb{R}$, on a :*

$\log 0$ *est non défini*	$\log 1 = 0$	$\log 2 = 1$
$\log(a \times b) = \log a + \log b$	$\log(\frac{a}{b}) = \log a - \log b$	$\log(a^c) = c \times \log a$

Lemme 5 (Règles de calcul pour l'exp**)** *Pour $a, b \in \mathbb{R}^{+*}$, on a :*

$2^0 = 1$	$2^{a+b} = 2^a \times 2^b$	$2^{a \times b} = (2^a)^b$
$2^{\log a} = a$	$\log 2^a = a$	

Lemme 6 (Sommes arithmétique et géométrique) *Pour $a, b \in \mathbb{N}$ avec $a \leq b$ et $x \in \mathbb{R} \setminus \{1\}$, on a :*

$$\sum_{i=a}^{b} i = (b - a + 1).\frac{b+a}{2} \qquad \text{et} \qquad \sum_{i=a}^{b} x^i = \frac{x^{b+1} - x^a}{x - 1}$$

1.2 Rappel de probabilités discrètes

- L'ensemble des résultats possibles d'une expérience probabiliste est appelé son **univers**, souvent noté Ω. Un **évènement primitif** (ou **élémentaire**) est un élément de l'univers, c'est-à-dire, un résultat possible. Plus généralement, un **évènement** est un sous-ensemble de l'univers, donc un ensemble de résultats possibles.
- Étant donné un univers Ω, une **probabilités** sur Ω est une valeur pour chaque élément primitif x, notée $\Pr[x]$, telle que $\sum_{x \in \Omega} \Pr[x] = 1$.
  La **probabilité d'un évènement** est la somme des probabilités de ses éléments.

- Si tous les évènements primitifs d'un univers ont la même probabilité (c'est-à-dire que pour tout $x \in \Omega$ on a $\Pr[x] = 1/\Omega$), on parle de **probabilité uniforme** sur Ω. Dans ce cas, un évènement A a pour probabilité $|A|/|\Omega|$ ('nombres de cas favorables/nombre de cas possibles').
- Une **variable aléatoire** est une fonction $X : \Omega \to V$ avec $V \subseteq \mathbb{R}$.
  "$X = v$" est l'évènement $\{\omega \in \Omega : X(\omega) = v\}$ et $\Pr[X = v] = \sum_{\omega : X(\omega) = v} \Pr[\omega]$
  "$X \leq v$" est l'évènement $\{\omega \in \Omega : X(\omega) \leq v\}$ et $\Pr[X \leq v] = \sum_{\omega : X(\omega) \leq v} \Pr[\omega]$
- Soit $X : \Omega \to V$ une variable aléatoire. **L'espérance de X** est : $\mathbb{E}[X] = \sum_{v \in V} v \times \Pr[X = v]$.
- La **probabilité de E sachant F** est $\Pr[E \mid F] = \frac{\Pr[E \cap F]}{\Pr[F]}$. L'**espérance de X sachant F** est $\mathbb{E}[X \mid F] = \sum_{v \in V} t \times \Pr[X = v \mid F]$.
- Deux évènements sont **indépendants** si $\Pr[E \wedge F] = \Pr[E].\Pr[F]$. Deux variables aléatoires sont indépendantes

si $\Pr[X = u \wedge Y = v] = \Pr[X = u].\Pr[Y = v]$ pour tous u, v.

Théorème 2 (Propriétés) *Soient E et F deux évènements et $X, Y : \Omega \to V$ deux variables aléatoires. On a les propriétés suivantes :*

- $Pr[\neg E] = 1 - Pr[E]$
- $Pr[E \vee F] \leq Pr[E] + Pr[F]$ — **Inégalité de Boole** *ou* **'Union bound'**
- $\sum_{v \in V} Pr[X = v] = 1$
- $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ — **Linéarité de l'espérance**

Si, de plus, $\Omega = \bigsqcup_i F_i$, partition de Ω en $(F_i)_i$, on a aussi :

- $Pr[E] = \sum_i Pr[E \mid F_i].Pr[F_i]$ — *Formule des* **probabilités totales**
- $\mathbb{E}[X] = \sum_i \mathbb{E}[X \mid F_i].Pr[F_i]$ — *Formule de* **l'espérance totale**

1.3 Bits et entiers aléatoires ou pseudo-aléatoires

- Les générateurs **pseudo-aléatoires** implantés dans les machines sont considérés d'un point de vue théorique et pratique comme des générateurs acceptables de bits, d'entiers et de réels aléatoires.

1.4 Lois de probabilités usuelles

- La loi **Uniforme** sur l'ensemble $V = \{1, \ldots, n\}$ est donné par $\Pr[X = v] = 1/|V| = 1/n$ pour tout $v \in V$. Elle se simule par le tirage d'un entier entre 1 et $|V|$. Son espérance est $\mathbb{E}[X] = \frac{n+1}{2}$.
- La loi **Bernoulli**(p) sur l'ensemble $V = \{0, 1\}$ est donnée par $\Pr[X = 1] = p$ et $\Pr[X = 0] = 1 - p$. Elle se simule par le tirage d'un bit aléatoire biaisé. Son espérance est $\mathbb{E}[X] = p$.
- La loi **Binomiale**(p, n) sur l'ensemble $V = \{0, \ldots, n\}$ est donnée par $\Pr[X = k] = \binom{n}{k} p^k (1-p)^k$ pour $k = 0, \ldots, n$. Elle se simule comme la somme de n variables de Bernoulli de paramètre p. Son espérance est $\mathbb{E}[X] = np$.
- La loi **Géométrique**(p) sur l'ensemble $V = \mathbb{N}$ est donnée par $\Pr[X = n] = p(1-p)^{n-1}$ pour $n \in \mathbb{N}$. Elle se simule comme la loi de la première apparition du '1' dans une suite de variables de Bernoulli de paramètre p. Son espérance est $\mathbb{E}[X] = 1/p$.

1.5 Borne des queues de distribution

Lemme 7 (Inégalité de Markov) *Soit $X : \Omega \to V$ une variable aléatoire* **positive** *($V \subseteq \mathbb{R}^+$).*
Pour tout $t > 0$, on a : $Pr[X \geq t] \leq \frac{1}{t}.\mathbb{E}[X]$

2 Recherche exhaustive et backtrack

2.1 Recherche exhaustive

- L'**odomètre** (ou compteur kilométrique) **binaire** permet de parcourir tous les mots de $\{0, 1\}^n$. Un mot est stocké dans un tableau T de dimension n. La complexité de l'algorithme est en $O(n)$, mais on verra

plus tard une complexité amortie en $O(1)$.

Algorithme : MotSuivant(T)
$i \leftarrow n-1$;
tant que $i \geq 0$ *et* $T[i] = 1$ **faire**
  $T[i] \leftarrow -1$;
  $i \leftarrow i-1$;
  si $i = -1$ **alors retourner** « Fin »;
$T[i] \leftarrow 1$;
retourner T;

- Un **littéral** est une variable logique ou sa négation. Une **clause** est une disjonction de de littéraux. Et une **formule SAT** est une conjonction de clauses.
  Ex : $\varphi(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2$
- Le problème SAT prend en entrée une formule SAT φ et décide si il existe ou non une affection des variables à `vrai` ou `faux` qui satifasse φ.
  Chaque clause C de φ est codé par un tableau où chaque case contient l'indice de la variable correspondante, précédée d'un signe moins si la variable est négative dans C. Une affectation A est codée par un tableau tel que $A[i] = 1$ (resp. -1) si x_i est affecté à `vrai` (resp. `faux`).
- Les algorithmes TestAff et AffSuivante respectivement testent une affectation et passe à l'affectation suivante.

Algorithme : TestAff(φ, A)
pour C *dans* φ **faire**
  $OK \leftarrow$ FAUX;
  pour ℓ *dans* C **faire**
    si $\ell \times A_{[|\ell|-1]} > 0$ **alors**
      $OK \leftarrow$ VRAI;
  si $OK =$ FAUX **alors retourner** FAUX;
retourner VRAI;

Algorithme : AffSuivante(A)
$i \leftarrow n-1$;
tant que $i \geq 0$ *et* $A_{[i]} = 1$ **faire**
  $A_{[i]} \leftarrow -1$;
  $i \leftarrow i-1$;
  si $i = -1$ **alors**
    retourner « Fin »
$A_{[i]} \leftarrow 1$;
retourner A;

- L'algorithme RechercheExhaustiveSAT cherche une affectation positive pour une formule SAT.

Algorithme : RechercheExhaustiveSAT(φ)
$A \leftarrow$ tableau de longueur n (nb de variables dans φ), initialisé à -1;
tant que non(TestAff(φ, A)) **faire**
  $A \leftarrow$ AffSuivante(A);
  si AffSuivante *a renvoyé « Fin »* **alors**
    retourner « Insatisfiable »
Renvoyer A;

Théorème 3 (Resolution SAT **exhaustive)** *L'algorithme* RechercheExhaustiveSAT *trouve une affectation satisfaisante s'il en existe une, et renvoie « Insatisfiable » sinon, en temps* $O(|\varphi|2^n)$.

- Le principe de la recherche exhaustive est de parcourir toutes les solutions possibles et pour chacune de tester si elle répond au problème.
  La complexité des algos correspondants est O (NombreSolutions × (CoûtTest + CoûtPassageSuivant))
- Le problème du **Voyageur de Commerce** (euclidien) a pour entrée un ensemble U de points du plan, et pour sortie une numérotation $u_0, \ldots, u_{n-1}$ des points de U qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$, où $\ell(-,-)$ désigne la distance euclidienne usuelle (c-à-d, $\ell(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$).

La résolution du Voyageur de commerce par recherche exhaustive demande de générer toutes les permutations des entiers 0 à $n-1$. Pour cela, on part de la permutation $\pi = 0, 1, 2, \ldots, n-1$ et on applique l'algorithme suivant.

Algorithme : PermSuivante(π)
si $\pi_{[0]} > \cdots > \pi_{[n-1]}$ **alors retourner** *'Fin'*;
Trouver j maximal tel que $\pi_{[j]} < \pi_{[j+1]}$;
Trouver $\ell > j$ maximal tel que $\pi_{[j]} < \pi_{[\ell]}$;
Échanger $\pi_{[j]}$ et $\pi_{[\ell]}$;
pour $0 < k < \frac{n-j}{2}$ **faire**
  Échanger $\pi_{[j+k]}$ et $\pi_{[n-k]}$; *//Retournement de* $\pi_{[j+1,n]}$
retourner π;

- L'algorithme suivant résout alors le problème du voyageur de commerce sur le graphe $G = (S, A)$ muni d'une fonction de longeur ℓ sur ses arêtes.

Algorithme : VoyageurDeCommerce(U)
$\pi \leftarrow$ tableau de taille n, initialisé à $[0, 1, \ldots, n-1]$;
$L_{\min} \leftarrow +\infty$;
$\pi_{\min} \leftarrow \pi$;
tant que PermSuivante *ne retourne pas 'Fin'* **faire**
  $L \leftarrow \sum_{i=0}^{n-1} \ell(U[\pi_{[i]}], U[\pi_{[i+1 \bmod n]}])$;
  si $L < L_{\min}$ **alors**
    $(L_{\min}, \pi_{\min}) \leftarrow (L, \pi)$;
  $\pi \leftarrow$ PermSuivante(π);
retourner $\pi_{\min}$;

Théorème 4 (Résolution exhaustive de VoyageurDeCommerce**)** *L'algorithme* VoyageurDeCommerce *résout le problème du voyageur de commerce en temps* $O(n \times n!)$.

2.2 Backtrack ou 'retour sur trace'

- La résolution de SAT par backtrack se fait en affectant récursivement chaque variable en testant à chaque étape la solution partielle obtenue. On utilise les algorithmes suivants, où φ est une formule SAT à n variables et b est une valeur booléenne (*Vrai* ou *Faux*).

Théorème 5 (Resolution SAT **par backtrack)** *L'algorithme* SatBacktrack *trouve une affectation satisfaisante s'il en existe une, et renvoie « Insatisfiable » sinon, en temps* $O(|\varphi| 2^n)$ *(mais est beaucoup plus efficace en pratique que l'algorithme* RechercheExhaustiveSAT*).*

Algorithme : ÉLIMINATION(φ, n, b)
$\psi \leftarrow$ formule vide;
pour C *dans* φ **faire**
  $C' \leftarrow$ clause vide;
  $sat \leftarrow$ FAUX;
  pour ℓ *dans* C **faire**
    si $|\ell| = n$ *et* $\ell \times b > 0$ **alors**
      $sat \leftarrow$ VRAI
    sinon
      si $|\ell| \neq n$ **alors**
        Ajouter ℓ à C';
  si NON(sat) **alors**
    Ajouter C' à ψ;
retourner ψ;

Algorithme : SATBACKTRACK(φ, n)
si φ *est vide* **alors**
  retourner $A = [1, \ldots, 1]$;
si φ *possède une clause vide* **alors**
  retourner *"Insatisfiable"*;
pour $b \in \{1, -1\}$ **faire**
  $\psi \leftarrow$ ÉLIMINATION(φ, n, b);
  $A \leftarrow$ SATBACKTRACK$(\psi, n-1)$;
  si ψ *n'est pas insatisfiable* **alors**
    retourner $A + [b]$;
retourner *"Insatisfiable"*;

- Le principe des algorithmes de Backtrack est de constuire récursivement des solutions partielles et de tester celles-ci à chaque étape (ce qui permet d'éliminer des branches de l'exploration). Cela correspond à parcourir l'arbre des solutions en profondeur en évitant de parcourir certaines branches.
- Le problème de **Sudoku généralisé** est le suivant.
  ENTRÉE : Une grille G de dimensions $n^2 \times n^2$, remplie d'entiers de 0 (= vide) à n^2
  SORTIE : La même grille G sans 0, tel que $G[i,j] \neq G[k,\ell]$ dès que : $i = k$ (ligne), ou $j = \ell$ (colonne), ou $\lfloor i/n \rfloor = \lfloor k/n \rfloor$ et $\lfloor j/n \rfloor = \lfloor \ell/n \rfloor$ (zone), ou alors *'aucune solution'*.
- Pour parcourir la grille du Sudoku, on utilise la primitive CASESUIVANTE(u, v) avec $0 \leq u, v \leq n^2$ où CASESUIVANTE(u, v) retourne **Fin** si $u = n^2 - 1$ et $v = n^2 - 1$, $(u+1, 0)$ si $v = n^2 - 1$ et $u \neq n^2 - 1$ et $(u, v+1)$ sinon.
- La résolution par backtrack du Sodoku généralisé se fait par les deux algorithmes suivants.

Algorithme :
VALIDE$(G, n, (u, v), x)$
pour $k = 0$ *à* $n^2 - 1$ **faire**
  si $(k \neq v$ *et* $G[u, k] = x)$ **alors**
    retourner Faux;
  si $(k \neq u$ *et* $G[k, v] = x)$ **alors**
    retourner Faux;
$(z_u, z_v) \leftarrow (n\lfloor u/n \rfloor, n\lfloor v/n \rfloor)$;
pour $k = 0$ *à* $n - 1$ **faire**
  pour $\ell = 0$ *à* $n - 1$ **faire**
    si $(z_u + k, z_v + \ell) \neq (u, v)$ *et* $G[z_u + k, z_v + \ell] = x$ **alors**
      retourner Faux
retourner Vrai;

Algorithme : SUDOKUBACKTRACK$(G, n, (u, v))$
tant que $(u, v) \neq$ **Fin** *et* $G[u, v] \neq 0$ **faire**
  $(u, v) \leftarrow CaseSuivante(u, v)$
si $(u, v) =$ **Fin alors**
  retourner Vrai;
pour x *de* 1 *à* n^2 **faire**
  si VALIDE$(G, n, (u, v)x)$ **alors**
    $G[u, v] \leftarrow x$;
    si SUDOKUBACKTRACK$(G, n, CaseSuivante(u, v))$ **alors**
      retourner Vrai;
$G[u, v] \leftarrow 0$;
retourner Faux;

- L'algorithme VALIDE prend notamment en entrée les coordonnées (u, v) de la case dans laquelle on teste la valeur x. Sa complexité est en $O(n^2)$.
  L'algorithme SUDOKUBACKTRACK est récursif et prend notamment en entrée les coordonnées (u, v) de la

case à partir de laquelle le backtrack est lancé (le premier appel se fait par SudokuBacktrack$(G, n, (0, 0))$). La complexité totale de l'algorithme est en $O(n^{4n^4})$.

3 Analyse amortie, analyse d'algorithmes probabilistes

3.1 Analyse amortie

- On revient sur le compteur binaire, qui prend un tableau T de taille k en entier représentant l'entier N encodé en binaire et retourne le tableau correspondant à l'entier $N + 1$.

Algorithme : Incrément(T)
$i \leftarrow 0$;
tant que $i < k$ *et* $T[i] = 1$ **faire**
  $T[i] \leftarrow 0$;
  $i \leftarrow i + 1$;
  si $i = k$ **alors retourner** 'Fin';
si $i < k$ **alors**
  $T[i] \leftarrow 1$;
retourner T;

Théorème 6 (Compteur binaire armorti) Incrément *est correct, a une complexité en* $O(k)$ *et une complexité amortie en* $O(1)$ *lorsqu'on l'appelle* 2^k *fois depuis l'appel initial sur* $T = [0, 0, \ldots, 0]$.

- Si on appelle un algorithme sur N instances résultant en un nombre c_i d'opérations élémentaires sur la $i$ème instance, le **coût amorti** (ou **complexité amortie**) sur la séquence d'instances est $\frac{1}{N}\sum_{i=1}^{N} c_i$. Classiquement, il y a trois méthode pour calculer ou borner cela.
- La **méthode de l'agrégat** consiste à calculer explicitement $\sum_{i=1}^{N} c_i$ et à retourner $\frac{1}{N}\sum_{i=1}^{N} c_i$.
- La **méthode comptable** consiste à considérer un 'compte' qui ne doit jamais être en négatif et surlequel, à l'étape i, on verse la somme a_i et on prélève la somme c_i. Plus précisément, on détermine des entiers a_i positifs ou négatifs, vérifiant, pour tout $i = 0, \ldots, N-1$, l'inégalité $\sum_{0\leq j\leq i} c_j \leq \sum_{0\leq j\leq i} a_j$. Le coût amorti est alors borné par $\frac{1}{N}\sum_{0\leq i\leq N-1} a_i$.
- La **méthode du potentiel** consiste à associer à l'instance produite après le $i$ème appel un potentiel Φ_i et à estimer le coût $a_i = c_i + (\Phi_i - \Phi_{i-1})$ après chaque appel. Le coût amorti est alors borné par $\frac{1}{N}\sum_{0\leq i\leq N-1} a_i$
- Une **liste dynamique** est donné par un tableau T de taille N et ayant une taille utile n. Les algorithmes

d'ajout et de suppression d'un élément dans une telle liste sont données ci-dessous.

Algorithme : AJOUT(T, N, n, x)
si $n < N$ **alors**
  $T[n] \leftarrow x$;
  retourner (T,N,n+1)
$U \leftarrow$ tableau de taille $2N$;
pour $i = 0$ *à* $N - 1$ **faire**
  $U[i] \leftarrow T[i]$;
$U[N] \leftarrow x$;
retourner $(U, 2N, n + 1)$

Algorithme : SUPPRESSION(T, N, n)
si $n = 1$ *ou* $n > N/4$ **alors**
  retourner $(T, N, n - 1)$
$U \leftarrow$ tableau de taille $N/2$;
pour $i = 0$ *à* $n - 2$ **faire**
  $U[i] \leftarrow T[i]$
retourner $(U, N/2, n - 1)$

Théorème 7 (Tableaux dynamiques) AJOUT *et* SUPPRESSION *sont corrects, de complexité* $O(n)$ *dans le pire des cas, mais avec un coût amorti en* $O(1)$ *pour n'importe quelle séquence d'opérations en partant d'une liste vide.*

3.2 Analyse d'algorithmes probabilistes

- Le problème de **sélection** (ou du $k$ième rang) consiste à trouver le $k$ième plus petit élément d'un tableau T donné, de taille n. L'algorithme suivant permet de répondre au problème.

Algorithme : QUICKSELECT(T, k)
$p \leftarrow T[i]$ avec i choisi aléatoirement entre 0 et $n - 1$;
$n_0 \leftarrow$ nombre d'élément $< p$ dans T *(boucle Pour)*;
si $n_0 = k - 1$ **alors retourner** p;
si $n_0 \geq k$ **alors**
  $T_0 \leftarrow$ tableau des éléments de T qui sont $< p$ *(boucle Pour)*;
  retourner QUICKSELECT(T_0, k)
$T_1 \leftarrow$ tableau des éléments de T qui sont $> p$ *(boucle Pour)*;
retourner QUICKSELECT$(T_1, k - n_0 - 1)$;

Théorème 8 (Espérance de complexité pour QUICKSELECT**)** *Soit* C_n *le nombre de comparaisons effectuées par* QUICKSELECT(T, k) *où* T *est de taille* n*. Alors* $\mathbb{E}[C_n] \leq 4n$*, quelque soit* k*.*

- Un **multigraphe** est un graphe dans lequel chaque paire de sommets est relié par aucune, une *ou plusieurs* arêtes.
- Une **coupe** dans un multigraphe $G = (V, A)$ est une partition (V_1, V_2) de l'ensemble de ses sommets en deux ensembles non vides.
  La **taille** de la coupe (V_1, V_2) est le nombre d'arêtes entre V_1 et V_2, c'est-à-dire $|\{u_1u_2 \in A : u_1 \in V_1, u_2 \in V_2\}|$
  Le **problème de la coupe minimale** prend en entrée un multigraphe G et retourne une coupe de taille minimale.
- Soit $G = (V, A)$ un multigraphe et uv une arête de G. Le multigraphe G/uv, obtenu par contraction de l'arête uv, a pour sommets $V \setminus v$ et pour ensemble d'arêtes $(A \setminus \{xv \ : \ xv \in A\}) \cup \{xu \ : \ xv \in A, x \neq u\}$

• l'algorithme suivant prend en entrée un multigraphe G et retourne une coupe de G.

Algorithme : CoupeMin(G)
tant que G *possède au moins* 3 *sommets* **faire**
  Choisir une arête uv de G, aléatoirement et uniformément;
  Contracter l'arête uv dans G;
retourner la coupe définie par les deux sommets restants

Théorème 9 (Analyse de CoupeMin**)** *En admettant que l'opération de contraction d'arête puisse s'effectuer en temps $O(n)$, où n est le nombre de sommets de G, alors* CoupeMin *retourne une coupe de G en temps $O(n^2)$.*
De plus, cette coupe est une coupe minimale de G avec probabilité $\geq \frac{2}{n(n-1)}$.

Lemme 8 (Lemme de répétition) *Si on répète N fois* CoupeMin *et qu'on garde la plus petite coupe renvoyée, cette coupe est minimale avec probabilité $\geq 1 - e^{-2N/n(n-1)}$*

Théorème 10 (CoupeMin **répété)** *Si on répète $N = 2n^2$ fois l'algorithme* CoupeMin *et que l'on retourne la coupe trouvée de plus petite taille, alors, en temps $O(n^4)$ une coupe minimale est retournée avec probabilité $\geq 98\%$.*

• Les algorithmes probabilistes se divisent classiquement en deux grandes familles :
  - Les algorithmes de type **Las Vegas** dont le résultat ne dépend pas des choix aléatoires, mais la complexité si (ex : QuickSelect).
  - Les algorithmes de type **Monte Carlo** dont le résultat dépend des choix aléatoires, mais pas la complexité (ex : CoupeMin).

• L'algorithme suivant prend en entrée un tableau T et retourne ce tableau, trié.

Algorithme : TriRapide(T)
si *taille*$(T) = 1$ **alors retourner** T;
$p \leftarrow T_{[i]}$ avec i choisi aléatoirement entre 0 et $n-1$; *# p est appelé 'le pivot'*
$n_p \leftarrow$ nombre d'indices i tels que $T_{[i]} = p$; *# Boucle 'Pour' pour cela*
$T_0 \leftarrow$ tableau des éléments de T qui sont $< p$; *# Boucle 'Pour' pour cela*
$T_1 \leftarrow$ tableau des éléments de T qui sont $> p$; *# Boucle 'Pour' pour cela*
$T_0 \leftarrow$ TriRapide(T_0);
$T_1 \leftarrow$ TriRapide(T_1);
retourner la concaténation T_0, n_p fois p, et T_1

Théorème 11 (TriRapide**)** TriRapide *retourne bien le tableau initial, trié. L'espérance du nombre de comparaisons effectuées par* TriRapide *est $O(n \log n)$*

4 Tables de hachage

4.1 Introduction

• Une **table de hachage** est donnée par :
  - Un ensemble de **clefs** : un **univers** U des clefs possibles : $U = \{0, \ldots, N-1\}$ et un ensemble $K \subset U$, de taille n, de **clefs utilisées**.
  - Une **table** T **de taille** m, dont chaque cas contient aucune, une ou plusieurs valeurs.

- Une **fonction de hachage** $h : U \to \{0, \ldots, m-1\}$, qui permet d'insérer le couple (k, v) dans la case $T[h(k)]$.

• Un ensemble $\mathcal{H}$ de fonctions de $U = \{0, \ldots, N-1\}$ dans $\{0, \ldots, m-1\}$ est **universel** si pour tout $k_1 \neq k_2 \in U$, lorsqu'on tire uniformément h dans $\mathcal{H}$ on a $\Pr[h(k_1) = h(k_2)] \leq 1/m$.

4.2 Résolution des collisions

• Il y a **collision** entre les clefs k_1 et k_2 si $h(k_1) = h(k_2)$.

Lemme 9 (Collisions dans une grande table) *Si $m = n^2$, et h est tirée uniformément dans un ensemble universel $\mathcal{H}$, alors la probabilité qu'il existe deux clefs $k_1 \neq k_2$ telles que $h(k_1) = h(k_2)$ est $\leq \frac{1}{2}$.*

• Dans la **résolution par chaînage**, chaque case i de T contient une liste chaînée. On désigne cette liste par $T[i]$ et pour une clef $k \in K$, la longueur de la liste $T[h(k)]$ est noté $\ell(k)$.

Algorithme : RECHERCHE(k, h, T)
Calculer $h(k)$;
Parcourir la liste $T[h(k)]$;
si on trouve k **alors retourner** (k, v);
sinon retourner NULL;

Algorithme : INSERTION(k, v, h, T)
Calculer $h(k)$;
Parcourir la liste $T[h(k)]$;
si on trouve k **alors** Changer la valeur de k en v;
sinon Ajouter (k, v) à la fin de la liste $T[h(k)]$;

• Les algorithmes RECHERCHE et INSERTION permettent de rechercher et d'insérer une clef dans la table, tous deux de complexité $O(L)$, où $L = \max\{\ell(k) \ : \ k \in K\}$.

Théorème 12 (Résolution par chaînage) *Soit T une table de hachage de taille m, avec h tirée uniformément dans un ensemble $\mathcal{H}$ universel. Si T contient n éléments et que les collisions sont résolues par chaînage, l'espérance de la complexité de* RECHERCHE *et de* INSERTION *est en $O(\alpha)$, où $\alpha = n/m$ est le* **taux de remplissage** *de la table.*

• Dans l'**adressage ouvert**, on se donne m fonctions de hachage $h_1, h_2, \ldots, h_m$, telles que pour tout $k \in U$ $(h_1(k), h_2(k), \ldots, h_m(k))$ est une permutation de $(0, 1, \ldots, m-1)$.

Algorithme : RECHERCHE$(k, h_1, \ldots, h_m, T)$
pour $i = 1$ *à* m **faire**
  Calculer $h_i(k)$;
  si $T[h_i(k)]$ contient k **alors retourner** (k, v);
  si $T[h_i(k)]$ est vide **alors retourner** NULL;

Algorithme : INSERTION$(k, v, h_1, \ldots, h_m, T)$
$i \leftarrow 1$; Calculer $h_1(k)$;
tant que $T[h_i(k)]$ *est non vide* **faire**
  $i \leftarrow i + 1$;
  Calculer $h_i(k)$;
Insérer (k, v) dans $T[h_i(k)]$;

Théorème 13 (Adressage ouvert) *Sous l'hypothèse que pour tout k, $(h_1(k), \ldots, h_m(k))$ est une permutation aléatoire et si le facteur de remplissage $\alpha = n/m$ est < 1, alors l'espérance du nombre de cases visitées pour une* RECHERCHE *infructueuse ou une* INSERTION *est $\leq \frac{1}{1-\alpha}$.*

4.3 Une famille universelle de fonctions de hachage

• **Conditions sur les familles universelles de fonctions de hachage.** On va demader qu'une famille universelle H de fonctions de hachage vérifie les conditions suivantes :
- la **taille de $\mathcal{H}$** est polynomiale en N et est $\geq \binom{N}{2}$
- la **représentation d'une fonction** h de $\mathcal{H}$ se fait en $O(\log N)$ bits

- le **tirage aléatoire** d'une fonction de $\mathcal{H}$ se fait en temps $O(\log N)$.

• Le **hachage multiplicatif** est la famille universelle $\mathcal{H}_p^{N,m}$ définie pour p premier avec $p > N$ et formée des fonctions suivantes pour $0 < a < p$ et $0 \leq b < p$:

$$h_{a,b} : \left| \begin{array}{ccc} \{0,\ldots,N-1\} & \to & \{0,\ldots,m-1\} \\ k & \to & ((ak+b) \bmod p) \bmod m \end{array} \right.$$

Lemme 10 (Système linéaire *modulo* p) *Soit $k_1 \neq k_2$ et $u \neq v$ dans $\mathbb{Z}/p\mathbb{Z}$, alors il existe un unique couple a, $b \in \mathbb{Z}/p\mathbb{Z}$ tel que $u = ak_1 + b$ et $v = ak_2 + b$.*

Théorème 14 (Hachage universel multiplicatif) *Pour tout $k_1 \neq k_2$, on a $Pr[h_{a,b}(k_1) = h_{a,b}(k_2)] \leq 1/m$. Autrement dit, la famille $\mathcal{H}_p^{N,m}$ est universelle (pour tout N, m et $p \geq N$ avec p premier).*

5 Algorithmes d'approximation

5.1 Premiers exemples

• Étant donné un graphe $G = (V, E)$, une **couverture par sommets** de G est un sous-ensemble de sommets de G qui touche toutes les arêtes de G. Autrement dit, X est une couverture par sommets de G si pour toute arête $uv \in E(G)$, on a $u \in X$ ou $v \in X$.

Algorithme : CouvApprox($G = (V, E)$)
$C \leftarrow \emptyset$;
tant que G *contient encore au moins une arête* **faire**
  Choisir une arête uv de G;
  $C \leftarrow C \cup \{u, v\}$;
  Supprimer u et v (et leurs arêtes incidentes) de G;
retourner C ;

Théorème 15 (Approximation de la couverture minimum par sommet) *L'algorithme* CouvApprox *retourne une couverture C du graphe G donné en paramètre en temps $O(n^2)$ (où n est le nombre de sommets de G). De plus, si* opt *désigne la taille d'une couverture de taille minimale de G, alors C vérifie $|C| \leq 2.\text{opt}$.*

• Le problème de la Somme Partielle prend en entrée un ensemble E d'entiers strictement positifs et un entier cible t et cherche un sous-ensemble $S \subset E$ dont la somme est $\leq t$ et tel que la somme des éléments de S soit la plus grande possible.

Pour un sous-ensemble $S \subset E$, on note ΣS la somme $\sum_{x \in S} x$.

Algorithme : SommePartApprox(E, t)

Trier E par ordre décroissant;
$S \leftarrow \emptyset$;
pour $i = 0$ *à* $|E| - 1$ **faire**
  si $E[i] \leq t$ **alors**
    Ajouter $E[i]$ à S;
    $t \leftarrow t - E[i]$;
retourner S ;

Théorème 16 (Analyse de SommePartApprox**)** *L'appel* SommePartApprox(E, t) *retourne en temps* $O(n \log n)$*, une solution* S *vérifiant* $\Sigma S \geq \frac{1}{2}$ OPT*, où* OPT *est la valeur d'une solution optimale au problème* Somme Partielle *pour* (E, t)*.*

5.2 Les algorithmes d'approximation

- On se donne un ensemble I des instances (entrées) et pour chaque $x \in I$, l'ensemble S des solutions **acceptables** (sorties possibles). De plus, il existe une **fonction de coût** $c : S \to \mathbb{R}$ qui définit la valeur d'une solution.
- Un problème est un **problème de maximisation** si il faut trouver $s \in S$ telle que $c(s)$ soit maximum, c'est-à-dire, telle que : $\forall s' \in S, c(s') \leq c(s)$.
  Symériquement, un problème est un **problème de minimisation** si il faut trouver $s \in S$ telle que $c(s)$ soit minimum, c'est-à-dire, telle que : $\forall s' \in S, c(s') \geq c(s)$.
- On note OPT la valeur d'une solution optimale, et on dit que OPT est la **valeur optimale** du problème. Si on traite un problème de maximisation, on a OPT $= \max_{s \in S} c(s)$, et si on traite un problème de minimisation, on a OPT $= \min_{s \in S} c(s)$.
- Un **algorithme d'α-approximation** est un algorithme qui *pour tout entrée* x renvoie une solution $s \in S$ telle que
  - $\alpha \cdot$ OPT $\leq c(s) \leq$ OPT pour un problème de maximisation, avec $0 < \alpha < 1$
  - OPT $\leq c(s) \leq \alpha \cdot$ OPT pour un problème de minimisation, avec $\alpha > 1$

  Le réel α est appelé **facteur d'approximation** de l'algorithme.

5.3 Exemples plus avancés

- Le problème d'Équilibrage prend en entrée un tableau D d'entiers positifs, de taille n, correspondant aux *durées* des tâches à répartir et un entier m correspondant aux nombres de processeurs disponibles. En sortie, on cherche un tableau A d'affectation de chaque tâche à un processeur (tâche i affectée au processeur j : $A[i] = j$). L'objectif est de minimiser le temps total correspondant à l'affectation, calculé

comme : $t(A) = \max_{1 \leq j \leq m} \left(\sum_{i:A[i]=j} D[i] \right)$.

Algorithme : ÉquilibrageGlouton(D, m)
$T \leftarrow$ tableau de taille m, intialisé à 0 (*temps total par processeur*);
pour $i = 1$ *à* n **faire**
  $j \leftarrow$ indice du minimum de T;
  $A_{[i]} \leftarrow j$;
  $T_{[j]} \leftarrow T_{[j]} + D_{[i]}$;
retourner A ;

Théorème 17 (Analyse de ÉquilibrageGlouton**)** *L'algorithme* ÉquilibrageGlouton *est une 2-approximation pour le problème* Équilibrage *et a une complexité $O(nm)$ (ou $O(n \log m)$ avec un tas).*

Théorème 18 (Analyse de ÉquilibrageGlouton**)** *Si le tableau D est connu à l'avance et est trié avant d'appeler* ÉquilibrageGlouton*, alors cet algorithme est une $\frac{3}{2}$-approximation pour le problème* Équilibrage *et a une complexité $O(n(m + \log n))$ (ou $O(n \log n)$ avec un tas).*

- Le problème d'MaxSat prend en entrée un ensemble $C_1, \ldots C_m$ de m clauses disjonctives sur n variables booléennes. On cherche une affectation des variables qui maximise le nombre de clauses satisfaites.

Théorème 19 (Analyse de MaxSatRand**)** *L'algorithme* MaxSatRand*, qui affecte chaque variable aléatoirement à* `vrai` *ou* `faux` *avec probabilité $\frac{1}{2}$, a une complexité de $O(n)$ et l'espérance du nombre de clauses qu'il satisfait est $\geq \frac{1}{2}$* OPT*, où* OPT *est le* OPT *est le nombre maximum de clauses qui peuvent être satisfaites par une affectation.*

- Le problème du VoyageurDeCommerce sur un graphe prend en entrée un graphe $G = (S, A)$ avec une longueur $\ell(u, v)$ pour chaque arête vérifiant **l'inégalité triangulaire** : $\ell(u, w) \leq \ell(u, v) + \ell(v, w)$ pour tous u, v, w si les arêtes correspondantes existent. On cherche une **tournée**, c'est-à-dire une numérotation u_0, …, u_{n-1} des sommets telle que pour tout $i = 0, \ldots, n-2$ l'arête $u_i u_{i+1}$ existe dans G, ainsi que l'arête $u_{n-1}u_0$, et qui minimise la longueur totale $\sum_{k=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$.
- Pour un graphe $G = (V, E)$, un **chemin** de G est une suite $v_1, v_2, \ldots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \ldots, k-1$.
  Un **cycle** de G est une suite $v_1, v_2, \ldots, v_k$ de sommets de G telle que $v_i v_{i+1}$ est une arête de G pour tout $i = 1, \ldots, k-1$ **et** que $v_k v_1$ soit aussi une arête de G.
- Un graphe G est **connexe** si pour tous sommets u et v de G, il existe un chemin de u à v.
  Un **arbre** est un graphe connexe sans cycle. Et un **arbre couvrant** de G est un sous-graphe de G, contenant tous ses sommets et qui est un arbre.

Théorème 20 (Connexité et arbre couvrant) *Un graphe G est connexe si, et seulement si, il admet un arbre couvrant.*
Un arbre sur n sommets contient $n-1$ arêtes.

- Étant donné un graphe $G = (V, E)$ connexe muni d'une **fonction de poids** (ou **de distance**) $\ell : E \to \mathbb{R}$ sur ses arêtes de G. Le **poids d'un arbre couvrant** T **de** G est la somme des poids des arêtes de T : $\ell(T) = \sum_{e \text{ arête de } T} \ell(e)$.
- Dans le problème de l'ArbreCouvrantPoidsMin un graphe $G = (V, E)$ connexe avec une fonction de poids ℓ sur ses arêtes est fourni en entrée, et on cherche un arbre couvrant de **de poids minimal** parmi

tous les arbres couvrants de G.

Algorithme : KRUSKAL(G, ℓ)

```
Trier les arêtes de G par poids croissant selon ℓ;
T ← ∅;
i ← 1;
pour x ∈ V faire
    comp(x) ← i ; i ← i + 1;
pour arête uv de G selon l'ordre calculé faire
    si comp(u) ≠ comp(v) alors
        T ← T ∪ {uv};
        aux ← comp(u);
        pour w ∈ V faire
            si comp(w) = aux alors
                comp(w) ← comp(v);
retourner T ;
```

Théorème 21 (Analyse de KRUSKAL) *L'algorithme de Kruskal peut s'implémenter en temps $O(m \log n)$, où n désigne le nombre de sommets du graphe G et m son nombre d'arêtes. Il retourne un arbre couvrant de G de poids minimum.*

Algorithme : VOYAGEURDECOMMERCE$_{2APPROX}(G, \ell)$

```
A ← arbre couvrant de poids minimum de G;
P ← parcours en profondeur de A;
Calculer (u_0, ..., u_{n-1}) ← sommets de G, dans l'ordre de première
  apparition dans P;
retourner (u_0, ..., u_{n-1}) ;
```

Théorème 22 (Analyse de VoyageurDeCommerce$_{2APPROX}$) *L'algorithme* VOYAGEURDECOMMERCE$_{2APPROX}$ *s'exécute en temps $O(m \log n)$ et fournit une 2-approximation au problème du* VOYAGEUR DE COMMERCE