- Notes de cours -

Présentation du module

Objectifs du cours. Le but du cours est d'étudier des outils et des méthodes pour concevoir et analyser des algorithmes. Les problèmes algorithmiques que nous étudierons viendront de différents domaines : traitement informatique, recherche oprérationnelle, logique, algèbre, combinatoire...

Intervenants. S. Bessy (responsable du module), S. Daudé, J. Destombes, J.-C. Konig, N. Pompidor (mail : `prenom.nom@umontpellier.fr`)

Planning (a priori...) Les cours (10 séances) ont lieu le mardi de 8h00 à 9h30, les tds (10 séances) le mardi ou vendredi (selon votre groupe) de 9h45 à 11h15 et les tps (10 séances) le mardi ou le vendredi (selon votre groupe) de 11h30 à 13h00. Emploi du temps exact à consulter sur l'ENT.

Sem.	Date	Contenu
03	Ma 18/01	**CM1 :** *Chapitre 1 :* Introduction, modèle et outils mathématiques
04	Ma 25/01	**CM2 :** Fin du *Chapitre 1* et *Chapitre 2 :* Algorithmes gloutons
	et Ve 28/01	**TD1 :** *Fiche 1 :* Complexité, premiers algos
05	Ma 1/02	**CM3 :** Fin du *Chapitre 2* et *Chapitre 3 :* SdD Arborescentes
	et Ve 4/02	**TD2 :** Fin de la *Fiche 1* **TP1 :** Algos gloutons
06	Ma 8/02	**CM4 :** *Chapitre 3 :* SdD Arborescentes
	et Ve 11/02	**TD3 :** *Fiche 2 :* Algos gloutons **TP2 :** Algos gloutons
07	Ma 15/02	**CM5 :** Fin du *Chapitre 3* et *Chapitre 4 :* Diviser pour Régner
	et Ve 18/02	**TD4 :** Fin de la *Fiche 2* et *Fiche 3 :* SdD Arbo **TP3 :** SdD Arborescentes
08	Ma 22/02	**CM6 :** *Chapitre 4 :* Diviser pour régner et *Chapitre 5 :* Prog. dynamique
	et Ve 25/02	**TD5 :** *Fiche 3 :* SdD Arborescentes **TP4 :** SdD Arborescentes
09	Ma 01/03	VACANCES D'HIVER
10	Ma 08/03	**CM7 :** Fin du *Chapitre 5 :* Prog. dynamique
	et Ve 11/03	**TD6 :** Fin de la *Fiche 3* et *Fiche 4 :* DpR **TP5 :** Diviser pour Régner
11	Ma 15/03	**CM8 : ! ! ! Controle continu, type exam ! ! !**
	et Ve 18/03	**TD7 :** *Fiche 4 :* Diviser pour Régner **TP6 :** Diviser pour régner
12	Ma 22/03	**CM9 :** *Chapitre 6 :* Algorithmes de graphes
	et Ve 25/03	**TD8 :** *Fiche 5 :* Prog. dyn. **TP7 :** Prog. dyn.
13	Ma 29/03	Examen Master, pas de CM/Td/Tp (en HAI403...)
14	Ma 04/04	Pas de CM cette semaine....
	et Ve 07/04	**TD9 :** Fin de la *Fiche 5 :* Prog. dyn. et *Fiche 6 :* Graphes **TP8 :** Prog. dyn.
15	Ma 11/04	**CM10 :** Fin du *Chapitre 6 :* Algorithmes de graphes
	et Ve 14/04	**TD10 :** Fin de la *Fiche 6 :* Algo de graphes **TP9 :** Algo. de Graphes
16	Ma 18/04	Pas de CM, c'est fini :-(...
	et Ve 21/04	Pas de Td, c'est fini :-(... **TP10 :** Algo. de Graphes

Modalité de contrôle de connaissance. L'évaluation comportera un contrôle continu et un examen. Le contrôle continu est composé d'une partie type exam (sur 15 pts, le 15 mars a priori) et d'une partie tp (sur 5 pts, tout au long du semestre avec possiblement un rendu en fin de semestre). 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.

Prérequis. D'un point de vue algorithmique, sont attendues des connaissances sur les instructions classiques en pseudo-code, la récursivité, un algorithme de tri au moins, la capacité à déterminer la complexité en temps d'algorithmes simples et la connaissance des structures de données de tableaux, piles, files, liste chaînées.
En programmation, les tp se feront en Python. Il faut maîtriser les bases du langages.

Ressource. Les ressources pédagogiques seront disponibles sur le moodle de l'Université.
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 (version électronique gratuite).

1 Introduction, rappels

1.1 Exemple introductif

1.2 Modèle

- Le **pseudo-code** d'un algorithme comprend des **opérations élémentaires** : déclaration de variable, affectation, lecture, écriture de variables, opération arithmétique : $+, -, \times, \div$, test élémentaire et appel de fonction, ainsi que des **boucles** : *pour* et *tant que*.
- Dans le modèle étudié (WORD-RAM), on considère que chaque **opération élémentaire** prend un **temps constant** et que chaque **déclaration de variable (simple) et appel de fonction** consomme un **espace machine constant**.
- Dans ce modèle-là, on va compter (ou majorer) **le nombre d'opérations élémentaires** (pour établir la **complexité en temps** de l'algo), compter (ou majorer) **le nombre de déclarations de variables et d'appels de fonctions** (pour établir la **complexité en espace** de l'algo), exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme, de manière asymptotique et dans le **pire des cas**.

1.3 Conception et analyse d'un algorithme

- Pour concevoir et analyser un algorithme, on doit : **écrire le pseudo-code** de l'algorithme, choisir **les structures de données** à utiliser pour les variables puis **analyser l'algorithme**.
- Pour analyser un algorithme, on étudie sa **terminaison**, on établit sa **complexité en temps et en espace** et enfin sa **validité**. Pour ce dernier point, on cherche souvent un **invariant de l'algorithme** que l'on prouve généralement par **récurrence**.

1.4 Outils mathématiques, notations de Landau

- **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 (Calcul des O) *On a les propriétés suivantes : $O(f) + O(g) = O(f + g)$, si $h = O(f)$ alors $f + h = O(f)$, $O(f) \times O(g) = O(f \times g)$ et pour $\lambda \in \mathbb{R}^+$ on a $O(\lambda f) = O(f)$.*

Lemme 2 (O et limites) *Pour $f : \mathbb{N} \longrightarrow \mathbb{R}$ et $g : \mathbb{N} \longrightarrow \mathbb{R}^{+*}$, si il existe une constante $c > 0$ 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 $f \neq O(g)$.*

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

Lemme 3 (Limite de l'inverse) *Si f et g sont des fonctions strictements positives (c'est-à-dire, que pour tout $n \in I\!N \setminus \{0, 1\}$, 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 $\frac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} \frac{1}{c}$
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 si $\frac{f(n)}{g(n)} \underset{n\to+\infty}{\longrightarrow} \infty$ alors $\frac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} 0$

Lemme 4 (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 $p > q$.

Lemme 5 (Règles de calcul pour le log**)** *Pour $a, b \in I\!R^{+*}$ et $c \in I\!R$, 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 6 (Règles de calcul pour l'exp**)** *Pour $a, b \in I\!R^{+*}$, on a :*

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

Théorème 1 (Formule de Stirling) *Pour $n \in I\!N^*$, $n!$ vaut $n \times (n-1) \times \cdots 2 \times 1$ et vérifie $n! \underset{n\to+\infty}{\sim} \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.*

• Pour $x \in \mathbb{R}$, on définit $\lfloor x \rfloor$ comme le plus grand entier k vérifiant $k \leq x$. De même, $\lceil x \rceil$ est le plus petit entier k vérifiant $x \leq k$. On a $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$.
Pour n entier, si n est pair on a $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$ et si n est impair, on a $\lfloor n/2 \rfloor = (n-1)/2$ et $\lceil n/2 \rceil = (n+1)/2$. Toujours pour $n \in \mathbb{N}$, on a toujours $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.

Lemme 7 (Sommes arithmétique et géométrique) *Pour $a, b \in I\!N$ avec $a \leq b$ et $x \in I\!R \setminus \{1\}$, on a :*

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

2 Algorithmes gloutons

2.1 Heuristiques gloutonnes

• Une **statégie gloutonne** consiste, à chaque étape, à faire un choix qui semble optimal à ce moment-là. Les choix sont localement optimaux dans l'espoir d'obtenir une solution globalement optimale.

2.2 Exemple 1 : choix de cours

• On considère un ensemble $(C[1], \ldots, C[n])$ de n cours. Pour tout $i = 1, \ldots, n$, le cours i a une date de début $debut(C[i])$ et une date de fin $fin(C[i])$. Le cours i est **compatible** avec le cours j si $[debut(C[i]), fin(C[i])] \cap$

$[debut(C[j]), fin(C[j])] = \emptyset$. Le but du problème **choix de cours** est de choisir un plus grand nombre de cours deux-à-deux compatibles.

- On construit alors l'algorithme ChoixCoursGlouton comme suit.

Algorithme : ChoixCoursGlouton(liste de cours de 1 à n)

Trier les cours par fin croissante : $C[1], \ldots, C[n]$;
$I \leftarrow \{C[1]\}$; `// Liste des cours choisis`
$f \leftarrow fin(C[1])$; `// Indice du dernier cours choisi`
pour $i = 2$ *à* n **faire**
  si $debut(C[i]) \geq f$ **alors**
    $I \leftarrow I \cup \{C[i]\}$; `// On a trouvé le cours compatible suivant`
    $f \leftarrow fin(C[i])$;
Retourner A;

Lemme 8 (Sous-solution optimale) *Il existe une solution optimale au problème de choix de cours qui est formée du cours $C[1]$ se terminant le plus tôt et d'une solution optimale du problème restreint aux cours j avec $debut(C[j]) \geq fin(C[1])$.*

Théorème 2 (Choix d'activités) *L'algorithme* CHOIX-GLOUTON-D-ACTIVITES *résout le problème du choix d'activités de manière optimale, et en temps $O(n \log n)$ pour n activités.*

2.3 Algorithmes gloutons génériques

- Un problème qui peut se résoudre avec un algorithme glouton rentre souvent dans le cadre suivant :
  - **Entrée :** Un ensemble fini X, avec une valeur v_x pour tout $x \in X$
    Une propriété $\mathcal{P}$ que doivent vérifier les sous-ensembles de X qui sont solutions. De tels sous-ensembles sont dits **acceptables**.
  - **Hypothèse :** $\mathcal{P}$ est **monotone vers le bas** : Si $A \subseteq X$ vérifie $\mathcal{P}$ et $B \subseteq A$ alors B vérifie $\mathcal{P}$
  - **Sortie** Un sous-ensemble A de X acceptable et qui maximise/minimise $v_A = \sum_{x \in A} v_x$ parmi tous les sous-ensembles acceptables
- L'algorithme glouton générique pour tenter de résoudre un tel problème est :

Algorithme : GloutonGénérique($X, \mathcal{P}$)

Trier X par *valeurs* décroissantes/croissantes + critère glouton de choix;
$S \leftarrow \emptyset$;
pour $x \in X$ *(dans l'ordre du tri)* **faire**
  si $S \cup \{x\}$ *vérifie* $\mathcal{P}$ **alors** $S \leftarrow S \cup \{x\}$;
retourner S

- On a alors le théorème suivant :

Théorème 3 (Validité de GloutonGénérique**)** *Si pour toute instance X du problème avec la propriété $\mathcal{P}$, il existe une solution optimale S telle que :*
  - *le premier élément x_0 de X, trié, appartienne à S*
  - *$S \setminus x_0$ soit une solution optimale du problème sur $X \setminus x_0$ pour la propriété $\mathcal{P}'$ où $A \subseteq X \setminus x_0$ vérifie $\mathcal{P}'$ si $A \cup \{x_0\}$ vérifie $\mathcal{P}$.*

Alors GloutonGénérique *est optimal.*

2.4 Exemple 2 : problème du sac à dos fractionnaire

- Dans **le problème du sac à dos** $(T, (v_1, t_1), \ldots (v_n, t_n))$, on dispose d'un sac à dos ayant un poids maximum de charge T et d'un ensemble d'objets $O_1, \ldots, O_n$, chaque objet O_i ayant une valeur v_i et un

poids t_i. Les objets sont supposés triés par valeur au kilo décroissante, c'est-à-dire $v_1/t_1 \geq \cdots \geq v_n/t_n$.
Le but est de charger le sac à dos en maximisant la valeur emportée. Plus précisément, pour chaque objet O_i on pose $x_i = 0$ si on ne prend pas O_i et $x_i = 1$ si on le prend. Ainsi, on veut trouver $(x_1, \ldots, x_n)$ tel que $\sum_{i=1}^{n} x_i t_i \leq T$ et que $\sum_{i=1}^{n} x_i v_i$ soit maximale.
Dans la **version fractionnaire du sac à dos**, on peut prendre la portion que l'on souhaite de chaque objet, c'est-à-dire qu'on autorise x_i à être un réel de $[0, 1]$.

Lemme 9 (Sous-solution optimale) *Il existe une solution optimale au problème de sac à dos fractionnaire $(T, (v_1, t_1), \ldots (v_n, t_n))$ avec $v_1/t_1 \geq \cdots \geq v_n/t_n$ qui est donnée par :*
- si $t_1 \leq T$, alors $x_1 = 1$ et $(x_2, \ldots, x_n)$ est une solution optimale du problème $(T - t_1, (v_2, t_2), \ldots (v_n, t_n))$
- si $t_1 > T$, alors $x_1 = T/t_1$ et $x_2 = \cdots = x_n = 0$

Théorème 4 (Sac à dos fractionnaire) *L'algorithme* GloutonGénérique *résout le problème du sac à dos fractionnaire de manière optimale, et en temps $O(n \log n)$ pour n objets.*

2.5 Idées de stratégies gloutonnes

- On a vu qu'un algorithme glouton fonctionne quand **il existe une solution optimale du problème formée du choix glouton d'un élément et d'une solution optimale du sous-problème restant**. C'est le cas des problèmes dont la structure combinatoire sous-jacente est celle d'un *matroïde*.
- Mais, même si cette propriété n'est pas vérifiée par un problème algorithmique, il est possible qu'un algorithme glouton donne une solution optimale au problème (voir l'algo de Huffman), ou fournisse une solution avec une garantie sur la qualité de cette solution (voir juste après), ou au pire donne une solution qui empiriquement n'est pas trop mauvaise, sans garantie théorique. On parle alors de **stratégie gloutonne** ou **heuristique gloutonne**. Ce sont souvent des algorithmes simples à coder, et ça peut être une bonne idée pour commencer à avoir des résultats sur un problème...

2.6 Exemple 3, exemple spécial : approximation de SET-COVER dans le plan

- Une variante du problème de SET-COVER dans le plan correspond à la question suivante. On se donne un ensemble de n maison dans le plan. Si on place un émetteur sur une maison, alors toutes les maisons situées à moins de 500m sont couvertes par cet émetteur. Le but est de placer le moins d'émetteurs possibles pour couvrir toutes les maisons.
- Le choix glouton proposé est de placer à chaque étape l'émetteur qui couvre le plus de maisons non déjà couvertes.

Lemme 10 (Approximation de SET-COVER) *Si k_{opt} désigne le nombre d'émetteurs à placer dans une solution optimale, alors le choix glouton proposé place au plus $k_{\text{opt}} . \log n$ émetteurs.*

3 Structures de données arborescentes : ABR et tas

3.1 Arbres binaires

- Un **arbre binaire** A est défini récursivement :
  - soit A est l'arbre vide $\emptyset$.
  - soit A est un arbre binaire non-vide et contient une **racine** notée $\mathbf{rac}(\boldsymbol{A})$, un **sous-arbre gauche** noté $\mathbf{SAG}(\boldsymbol{A})$ et un **sous-arbre droit** noté $\mathbf{SAD}(\boldsymbol{A})$ qui sont eux-même des arbres binaires.
- Un **nœud** non vide, x, possède une valeur, $\mathbf{val}(\boldsymbol{x}) \in \mathbb{R}$ et un lien vers 3 autres nœuds : $\mathbf{pere}(\boldsymbol{x})$, $\mathbf{filsG}(\boldsymbol{x})$ et $\mathbf{filsD}(\boldsymbol{x})$ tels que :
  - Si $\text{filsG}(x) \neq \emptyset$ alors $\text{pere}(\text{filsG}(x)) = x$ - Si $\text{filsD}(x) \neq \emptyset$ alors $\text{pere}(\text{filsD}(x)) = x$
- Un nœud x est une feuille si $\text{filsG}(x) = \emptyset$ et $\text{filsD}(x) = \emptyset$

- La **hauteur** d'un nœud x, notée $\boldsymbol{h(x)}$ est donnée récursivement par : $h(\text{rac}(A)) = 0$ et si $x \neq \text{rac}(A)$ alors $h(x) = h(\text{pere}(x)) + 1$.
  La **hauteur de** A est $\boldsymbol{h(A)} = \max\{h(x) \ : \ x \in A\}$. Pour $0 \leq k \leq h(A)$, le k**ième niveau de** A est l'ensemble $\{x \in A \ : \ h(x) = k\}$, on le note $\boldsymbol{N_k}$.

Lemme 11 (Nombre de nœuds) *Pour $0 \leq k \leq h(A)$, on a $|N_k| \leq 2^k$.*
Pour tout arbre binaire A, on a $h(A) \leq n(A) \leq 2^{h(A)-1} + 1$.
Autrement, on a l'encadrement suivant : $\lfloor \log(n(A)) \rfloor \leq h(A) < n(A)$

- Un **parcours infixe** d'un arbre binaire A est obtenu par l'appel ParcoursInfixe(rac(A)), où ParcoursInfixe est l'algorithme suivant.

```
Algorithme : ParcoursInfixe(x)
si x ≠ null alors
    ParcoursInfixe(FilsG(x));
    Afficher val(x);
    ParcoursInfixe(FilsD(x));
```

- L'algorithme générique suivant s'utilise pour calculer des fonctions des valeurs (min, max,...) ou des nœud (nombres, hauteur...) d'un arbre binaire. On effectue l'appel de AlgoGen(rac(A)) avec :

```
Algorithme : AlgoGen(x)
res ← valeur pour l'arbre vide ;
si x ≠ ∅ alors
    res_G ← AlgoGen(filsG(x));
    res_D ← AlgoGen(filsD(x));
    res ← f(res, res_G, res_D, x);
retourner res
```

Lemme 12 (Complexité Parcours Infixe et Algo Générique sur les arbres binaires) *L'algorithme générique sur les arbres binaires a une complexité $O(n(A))$ si le calcul de f a une complexité en temps en $O(1)$. Il est en de même du parcours infixe.*

3.2 Arbre binaire de recherche

- Pour un nœud x d'un arbre binaire A, le **sous-arbre gauche** (resp. **droite**) de x est le sous-arbre de A enraciné en FilsG(x) (resp. FilsD(x)). On le note $\boldsymbol{\text{SaG}(x)}$ (resp. $\boldsymbol{\text{SaD}(x)}$) (avec $\text{SaG}(x) = \emptyset$ si x n'a pas de fils gauche et $\text{SaD}(x) = \emptyset$ si x n'a pas de fils droite).
- Un arbre binaire A est un **arbre binaire de recherche** (ou **ABR**) si pour tout nœud x de A, on a :
  - tout nœud y de $\text{SaG}(x)$ vérifie $\text{val}(y) \leq \text{val}(x)$, et
  - tout nœud z de $\text{SaD}(x)$ vérifie $\text{val}(x) \leq \text{val}(z)$.

Lemme 13 (Parcours infixe d'un ABR) *Le parcours infixe d'un arbre binaire A imprime les valeurs des nœuds de A dans l'ordre croissant si, et seulement si, A est un ABR.*

- Les algorithmes suivants permettent respectivement de : trouver la valeur minimum de l'ABR, rechercher une valeur dans un ABR, et rechercher le successeur d'une valeur dans l'ABR.

```
Algorithme : MinAbr(x)
// MinArb renvoie un nœud de valeur minimale dans le sous-arbre enraciné en x
tant que FilsG(x) ≠ null faire
    x ← FilsG(x);
retourner x;
```

Algorithme : RECHERCHE(x, k)

```
// L'algorithme recherche un nœud de valeur k dans le sous-arbre enraciné en x
```

tant que $x \neq null$ *et* $k \neq \text{val}(x)$ **faire**
  si $k < \text{val}(x)$ **alors** $x \leftarrow \text{FilsG}(x)$;
  sinon
    si $k > \text{val}(x)$ **alors** $x \leftarrow \text{FilsD}(x)$;

retourner x;

Algorithme : SUCCESSEUR(x)

```
// Retourne le nœud y de l'arbre dont la valeur est le successeur de val(x) parmi
   les valeurs de l'arbre, ou null si x a la valeur maximale dans l'arbre
```

si $\text{FilsD}(x) \neq null$ **alors**
  retourner MINARB$(\text{FilsD}(x))$;

$y \leftarrow \text{pere}(x)$;
tant que $y \neq null$ *et* $x = \text{FilsD}(y)$ **faire**
  $x \leftarrow y$;
  $y \leftarrow \text{pere}(x)$;

retourner y;

Lemme 14 (Recherche, min et successeur dans un ABR) *Les algorithmes* RECHERCHE, MINARB *et* SUCCESSEUR *sont valides et s'exécutent en temps* $O(h(A))$ *sur un arbre binaire de recherche* A.

- Les algorithmes suivants permettent respectivement d'insérer un nœud dans un ABR, de remplacer un sous-arbre par un autre puis, de supprimer un nœud d'un ABR.

Algorithme : INSÉRER(A, z)

```
// Insère le nœud z dans l'ABR A en conservant la structure d'ABR.
```

$x \leftarrow \text{rac}(A)$;
$p \leftarrow \emptyset$;
tant que $x \neq \emptyset$ **faire**
  $p \leftarrow x$;
  si $\text{val}(z) < \text{val}(x)$ **alors**
    $x \leftarrow \text{FilsG}(x)$;
  sinon $x \leftarrow \text{FilsD}(x)$;

père$(z) \leftarrow p$;
si $p = \emptyset$ **alors** $\text{rac}(A) \leftarrow z$;
sinon
  Attacher z comme une feuille, fils de p;

Algorithme : REMPLACER(A, x, z)

```
// Remplace dans A le sous-arbre enraciné en x par le sous-arbre enraciné en z
```

$p \leftarrow$ père(x);
père$(x) \leftarrow \emptyset$;
si $p = \emptyset$ **alors**
  rac$(A) \leftarrow z$;
sinon
  si $x =$ FilsG(p) **alors** FilsG$(p) \leftarrow z$;
  sinon FilsD$(p) \leftarrow z$;
si $z \neq \emptyset$ **alors** père$(z) \leftarrow p$;

Algorithme : SUPPRIMER(A, z)

```
// Insère le nœud z dans l'ABR A en conservant la structure d'ABR.
```

si FilsG$(z) = \emptyset$ **alors**
  REMPLACER$(A, z,$ FilsD$(z))$;
sinon si FilsD$(z) = \emptyset$ **alors**
  REMPLACER$(A, z,$ FilsG$(z))$;
sinon
  $y =$SUCCESSEUR(z);
  REMPLACER$(A, y,$ FilsD$(y))$;
  Remplacer dans A, le nœud z par le mœud y;

Lemme 15 (Insertion, suppression dans un ABR) *Les algorithmes* INSÉRER, REMPLACER *et* SUPPRIMER *sont valides et s'exécutent en temps* $O(h(A))$ *sur un arbre binaire de recherche* A.

3.3 Tas

- un **arbre binaire quasi-complet** est un arbre binaire dont tous les niveaux sont complets sauf potentiellement le dernier dont tous les nœuds sont 'rangés le plus à gauche possible'.

Lemme 16 (Hauteur d'un quasi-complet) *Si* A *est quasi-complet alors on a* $2^{h(A)} \leq n(A) \leq 2^{h(A)-1} + 1$, *c'est-à-dire* $h(A) = \lfloor \log n(A) \rfloor$

- On numérote les sommets d'un arbre binaire A par une fonction $\boldsymbol{num} : A \to \mathbb{N}$, en posant num(rac A) $= 0$ puis récursivement :
  - si FilsG$(x) \neq \emptyset$, alors num(FilsG(x)) $= 2\,$num$(x) + 1$
  - si FilsD$(x) \neq \emptyset$, alors num(FilsD(x)) $= 2\,$num$(x) + 2$

Lemme 17 (Numérotation des quasi-complets) *Un arbre binaire est quasi-complet si et seulement si ses nœuds sont numérotés de* 0 *à* $n(A) - 1$.

Lemme 18 (Numérotation des quasi-complet) *Dans* A *quasi-complet, si* num$(x) = i$ *pour un nœud* x *alors on a :* num(pere(x)) $= \lfloor (i-1)/2 \rfloor$ *(si* $i \neq 0$*), si* filsG$(x) \neq \emptyset$ *on a* num(filsG(x)) $= 2i + 1$ *et si* filsD$(x) \neq \emptyset$ *on a* num(filsD(x)) $= 2i + 2$ *et* $h(x) = \lfloor \log(i+1) \rfloor$. *On a aussi* num(rac$(A)$) $= 0$.

- Un **tas (max)** est un arbre binaire A quasi-complet vérifiant la **propriété de tas (max)** : pour tout $x \neq$ rac(A), val(père(x)) $\geq$ val(x).
  Ou, de façon équivalente, un tableau T est un tas (max) si pour tout $i \geq 1$, $T[\lfloor \frac{i-1}{2} \rfloor] \geq T[i]$.
- Les algorithmes suivants permettent respectivement d'insérer et de supprimer un nœud dans un tas encodé par un tableau T.

Algorithme : INSÉRER(T, x)
`// Insère le nœud` x `dans` T
$i \leftarrow n(T)$;
Agrandir T d'une case;
$T[i] \leftarrow x$;
REMONTER(T, i);

Algorithme : REMONTER(T, i)
`// Fait remonter le nœud` i `afin de retrouver une structure de tas`
tant que $i > 0$ *et* $T[\text{père}(i)] < T[i]$ **faire**
  Échanger $T[i]$ et $T[\text{père}(i)]$;
  $i \leftarrow \text{père}(i)$;

Algorithme : SUPPRIMER(T, i)
`// Supprime le nœud` i `de` T
$x \leftarrow T[i]$;
$T[i] \leftarrow T[n(T) - 1]$;
Réduire T d'une case;
si $(\text{père}(i) \neq \emptyset$ & $T[\text{père}(i)] < T[i])$
alors
  REMONTER(T, i);
sinon
  ENTASSER(T, i);
retourner x;

Algorithme : ENTASSER(T, i)
`// Entasse le nœud` i `afin de retrouver une structure de tas`
$(m, g, d) \leftarrow (i, \text{FilsG}(i), \text{FilsD}(i))$;
si $g < n(T)$ *et* $T[g] > T[m]$ **alors** $m \leftarrow g$;
si $d < n(T)$ *et* $T[d] > T[m]$ **alors** $m \leftarrow d$;
si $m \neq i$ **alors**
  Échanger $T[i]$ et $T[m]$;
  ENTASSER(T, m);

Lemme 19 (Insertion et suppresion dans un tas) *Les algorithmes* INSÉRER *et* SUPPRIMER *sur un tas sont valides et s'exécutent en temps* $O(\log n)$ *sur un tas à* n *nœuds.*

- L'algorithme suivant permet d'effectuer un tri par tas.

Algorithme : TRI-PAR-TAS(T)
`// Les valeurs du tableau` T `sont triées dans le tableau` S
$S \leftarrow$ tableau vide de taille $n(T)$;
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **faire**
  ENTASSER(T, i);
pour $i = n(T) - 1$ *à* 0 **faire**
  $S[i] \leftarrow$ SUPPRIMER$(T, 0)$;
retourner S;

Lemme 20 (Tri par tas) *L'algorithme* TRI-PAR-TAS *permet de trier un tableau de taille* n *en temps* $O(n \log n)$.

Lemme 21 (Borne inf pour le tri) *Tout algo de tri ne faisant que des comparaisons effectue* $\Omega(n \log n)$ *comparaisons dans le pire des cas pour trier* n *nombres quelconques.*

4 Diviser pour régner

4.1 Stratégie 'diviser pour régner'

- Dans cette méthode, on : *(1)* divise le problème en sous-problèmes, *(2)* résout récursivement ces sous-problèmes et *(3)* combine les solutions pour reconstruire la solution du problème original.
  Cette stratégie est principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.

4.2 Exemple 1 : algorithme de tri fusion

• L'algorithme TRI-FUSION est donné ci-dessous.

Algorithme : TRI-FUSION(T)
$n \leftarrow$`taille`(T);
si $n = 1$ **alors**
  Retourner T;
sinon
  $T_1 \leftarrow$TRI-FUSION$(T[0, \lfloor n/2 \rfloor - 1])$;
  $T_2 \leftarrow$TRI-FUSION$(T[\lfloor n/2 \rfloor, n-1])$;
  Retourner FUSION(T_1, T_2);

Algorithme : FUSION(T_1, T_2)
$n_1 \leftarrow$`taille`(T_1) ; $n_2 \leftarrow$`taille`(T_2);
S tableau de taille $n_1 + n_2$;
$i_1 \leftarrow 0$; $i_2 \leftarrow 0$; $i_S \leftarrow 0$;
tant que $i_S < n_1 + n_2$ **faire**
  si $i_1 = n_1$ *ou* $(T_2[i_2] < T_1[i_1]$ *et* $i_2 < n_2)$ **alors**
    $S[i_S] \leftarrow T_2[i_2]$; $i_2 \leftarrow i_2 + 1$;
  sinon
    $S[i_S] \leftarrow T_1[i_1]$; $i_1 \leftarrow i_1 + 1$;
  $i_S \leftarrow i_S + 1$;
Retourner S;

Lemme 22 (Tri-fusion) *L'algorithme* TRI-FUSION *trie le tableau T de taille n fourni en paramètre en temps $O(n \log n)$.*

4.3 Le théorème maître

Théorème 5 ('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 $d < \log a / \log b$).

4.4 Exemple 2 : multiplication d'entiers

• Pour cet exemple, on se place dans le modèle RAM (et pas WORD-RAM comme pour le reste du cours !).
On s'interesse au problème de multiplication d'entiers :
  Entrée Deux entiers A et B écrits en base 10
  Sortie L'entier $C = A \times B$, en base 10
La multiplication 'usuelle' demande $O(n^2)$ multiplications d'entiers.

• L'algorithme de Karatsuba est donné ci-dessous.

Algorithme : KARATSUBA(A, B)
si *A et B n'ont qu'un chiffre* **alors retourner** $a_0 b_0$;
Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$;
Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$;
$C_{00} \leftarrow$ KARATSUBA(A_0, B_0);
$C_{11} \leftarrow$ KARATSUBA(A_1, B_1);
$D \leftarrow$ KARATSUBA$(A_0 - A_1, B_0 - B_1)$;
retourner $C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$;

Lemme 23 (Algorithme de Karatsuba) *L'algorithme de Karatsuba permet de multiplier deux entiers de n chiffres en temps $O(n^{\log 3})$ (avec $\log 3 \sim 1,585$).*

4.5 Exemple 3, exemple spécial : calcul de rangs

- Pour un tableau T de n nombres, **le $k^{\text{ième}}$ rang de T** est le $k^{\text{ième}}$ plus petit élément de T. On le note $\textbf{rang}(k, T)$. Par exemple, $\text{rang}(1, T)$ est le minimum de T, $\text{rang}(n, T)$ est son maximum et $\text{rang}(\lfloor n/2 \rfloor, T)$ est **la médiane** de T.
- Le choix d'un élément v de T comme **pivot** permet de séparer T en 3 tableaux :
  - $\boldsymbol{T_{\text{inf}}}$ qui contient les éléments x de T vérifiant $x < v$,
  - $\boldsymbol{T_{\text{seq}}}$ qui contient les éléments x de T vérifiant $x = v$ et
  - $\boldsymbol{T_{\text{sup}}}$ qui contient les éléments x de T vérifiant $x > v$.

  On note $\boldsymbol{n_{\text{inf}}}$, $\boldsymbol{n_{\text{eq}}}$ et $\boldsymbol{n_{\text{sup}}}$ les tailles respectifs de ces trois tableaux avec $n_{\text{inf}} + n_{\text{eq}} + n_{\text{sup}} = n$
- On a alors les formules et l'algorithme suivants :
  - $\text{rang}(k, T) = \text{rang}(k, T_{\text{inf}})$ si $k \leq n_{\text{inf}}$,
  - $\text{rang}(k, T) = v$ si $n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}}$ et
  - $\text{rang}(k, T) = \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}})$ si $n_{\text{inf}} + n_{\text{eq}} < k$.

Algorithme : RANG(T, k)
si $k = 1$ **alors**
  Retourner $T[0]$;
sinon
  $n_{\text{inf}} \leftarrow n$; $n_{\text{sup}} \leftarrow n$;
  tant que $n_{\text{inf}} > 3n/4$ *ou* $n_{\text{sup}} > 3n/4$ **faire**
    Tirer p au hasard dans $\{0, \dots n-1\}$;
    Prendre $v = T[p]$ comme pivot;
    Calculer n_{inf} et n_{sup};
  si $k \leq n_{\text{inf}}$ **alors** Calculer T_{inf} et retourner RANG(T_{inf}, k);
  si $n_{\text{inf}} < k \leq n - n_{\text{sup}}$ **alors** Retourner v;
  si $n - n_{\text{sup}} < k$ **alors** Calculer T_{sup} et retourner RANG$(T_{\text{sup}}, k + n_{\text{sup}} - n)$;

Lemme 24 (Calcul de rang) RANG(T, k) *retourne le $k^{ième}$ rang de T en temps $O(n)$ en moyenne.*

5 Programmation dynamique

- Principe de la programmation dynamique : reconstruire la solution d'un problème à partir de solutions des sous-problèmes, calculées efficacement (en tout cas, plus efficacement que par des appels récursifs).

5.1 Premier exemple : plus longue sous-séquence croissante (PLSSC)

- Étant donné un tableau T de taille n, une **plus longue sous-séquence croissante (PLSSC)** de T correspond à une suite la plus grande possible d'indices $0 \leq i_1 < i_2 < \cdots < i_k \leq n-1$ telle que $T[i_1] \leq T[i_2] \leq \cdots \leq T[i_k]$.
  On note $\textbf{plssc}(\boldsymbol{T})$ la longueur de cette suite. Pour des indices $i, j \in \{0, \dots, n-1\}$ avec $i < j$, on note $\boldsymbol{T[i, j]}$ le sous-tableau de T formé des valeurs $T[i], T[i+1], \dots, T[j]$. Enfin, on note $\boldsymbol{L(i)}$ la longueur d'une PLSSC de $T[0, i]$ terminant par i.

Lemme 25 (Formule récursive pour la PLSSC) *On a $L[0] = 1$ et pour tout $i \in \{1, \dots, n-1\}$ on a la formule de récurrence suivante :*

$$L(i) = 1 + \max\{L(j) \ : j < i \text{ et } T[j] \leq T[i]\}$$

- L'algorithme suivant implémente le calcul de la formule récursive.

Algorithme : PLSSC(T)
$L(0) \leftarrow 1$; $ind[0] \leftarrow -1$; // ind stocke l'élément précédent d'une plssc finissant en i
pour $i = 1$ *à* $n-1$ **faire**
  $max \leftarrow 0$; $ind[i] \leftarrow -1$; // Calcul du max de la formule récursive
  pour $j = 0$ *à* $i-1$ **faire**
    si $T[j] \leq T[i]$ *et* $L(j) \geq max$ **alors**
      $max \leftarrow L(j)$; $ind[i] \leftarrow j$;
  $L(i) \leftarrow max + 1$;
$LMax \leftarrow 0$; // On calcule la valeur de $plssc(T)$
$indMax \leftarrow 0$; // On stocke l'indice de la dernière case d'une telle PLSSC
pour $i = 0$ *à* $n-1$ **faire**
  si $L(i) > Lmax$ **alors**
    $Lmax \leftarrow L(i)$; $indMax \leftarrow i$;
Retourner $Lmax$, $indMax$ et le tableau ind ;

Algorithme : PLSSC_REC(T, i_M, Ind)
$S \leftarrow \emptyset$; // On stocke dans S les indices d'une PLSSC
$i \leftarrow i_M$;
tant que $i \neq -1$ **faire**
  $S \leftarrow \{i\} \cup S$;
  $i \leftarrow Ind[i]$;
retourner S;

Lemme 26 (Calcul d'une PLSSC) *L'algorithme* PLSSC *calcule la longueur d'une PLSSC de T en temps $O(n^2)$. À l'aide du tableau ind,* PLSSC_REC *construit une PLSSC de T en temps $O(n)$.*

5.2 Éléments de programmation dynamique

- Un problème a des chances d'être résolu par programmation dynamique, si il existe une **formule récursive** pour ce problème, liant la valeur d'une solution optimale à la valeur de solutions optimales de sous-problèmes, pas forcément disjoints (contrairement à l'approche 'diviser pour régner').
  Les sous-problèmes sont alors résolu par taille d'instance croissante. On parle d'approche dîte '*bottom-up*', contrairement aux exemples illustrant le paradigme 'diviser pour régner' pour lesquels l'approche est dîte '*top-down*'.
- Très souvent, la valeur d'une solution optimale est facile à obtenir avec la formule récursive. Pour obtenir la solution en elle-même, il faut un traitement supplémentaire.

5.3 Deuxième exemple : choix de cours valués

- Le problème du **choix de cours valués** prend en entrée un ensemble C de cours $C_i = (d_i, f_i, e_i)$, où d_i est le début, f_i la fin et e_i le nombre de crédits ECTS. Sa sortie est un ensemble de cours $(C_{i_1}, \ldots, C_{i_k})$ compatibles qui maximise le nombre total de crédits ECTS.
  L'algorithme ChoixCoursGlouton est arbitrairement mauvais pour ce problème !

Lemme 27 (Formule récursive pour le choix de cours valué) *Soit $maxECTS(k)$ le nombre maximal de crédits ECTS atteignable avec les cours $C_0, \ldots, C_k$ ($maxECTS(-1) = 0$), et $pred(k) = \max\{j : f_j \leq d_k\}$ (avec $\max(\emptyset) = -1$).*
Alors $maxECTS(0) = e_0$ et pour $1 \leq k < n$, $maxECTS(k) = \max(maxECTS(k-1), e_k + maxECTS(pred(k)))$.

Théorème 6 (Choix de cours valués) *La formule récursive fournit un algorithme en $O(n^2)$ pour le problème du choix de cours valué.*

5.4 Troisième exemple : la distance d'édition

- La **distance d'édition** entre deux mots A et B est la **longueur de la plus courte suite de transformations** pour passer de A à B, en utilisant l'**insertion** d'une nouvelle lettre, la **suppression** d'une lettre, et le **remplacement** d'une lettre par une autre. C'est aussi le nombre minimal de désaccords dans un **alignement** de A et B.

Lemme 28 (Formule récursive pour la distance d'édition) *Soit* $\text{edit}(i,j)$ *la distance d'édition entre* $A[0,i]$ *et* $B[0,j]$. *Alors* $\text{edit}(i,j) = \min(\text{edit}(i-1,j)+1, \text{edit}(i,j-1)+1, \text{edit}(i-1,j-1)+\mathbf{1}_{A[i]\neq B[j]})$ *où* $\mathbf{1}_{A[i]\neq B[j]}$ *vaut* 1 *si* $A[i]\neq B[j]$ *et* 0 *sinon.* On utilise la convention $\text{edit}(i,-1) = i+1$ et $\text{edit}(-1,j) = j+1$.

- On obtient un algorithme pour le calcul de la distance et un algorithme de reconstruction de l'alignement.

Algorithme :
DistanceEdition(A, B)
$(m,n) \leftarrow$ tailles de A et B;
$E \leftarrow$ tableau de dimensions m par n;
pour $i = 0$ *à* $m-1$ **faire**
  pour $j = 0$ *à* $n-1$ **faire**
    $\epsilon \leftarrow 0$ si $A[i] = B[j]$, 1 sinon;
    $E[i,j] \leftarrow \min(E[i-1,j]+1,$
      $E[i,j-1]+1,$
      $E[i-1,j-1]+\epsilon)$;
retourner $E[m-1,n-1]$

Algorithme : Alignement(A, B, E)
$(i,j) \leftarrow (m-1, n-1)$;
tant que $i \geq 0$ *et* $j \geq 0$ **faire**
  si $E[i,j] = E[i-1,j-1]$ *et* $A[i] = B[j]$ **alors**
    $(i,j) \leftarrow (i-1, j-1)$;
  sinon si $E[i,j] = E[i-1,j-1]+1$ **alors**
    $(i,j) \leftarrow (i-1, j-1)$;
  sinon si $E[i,j] = E[i-1,j]+1$ **alors**
    Insérer « _ » après $B[j]$; $i \leftarrow i-1$;
  sinon si $E[i,j] = E[i,j-1]+1$ **alors**
    Insérer « _ » après $A[i]$; $j \leftarrow j-1$;
tant que $i \geq 0$ **faire** Insérer « _ » en tête de B ;
  $i \leftarrow i-1$;
tant que $j \geq 0$ **faire** Insérer « _ » en tête de A ;
  $j \leftarrow j-1$;
retourner A *et* B;

Théorème 7 (Distance d'édition) *La distance d'édition entre deux mots A et B de tailles respectives m et n peut être calculée en temps $O(m \times n)$. L'alignement ou la suite de transformations correspondants peuvent ensuite être calculés en temps $O(m+n)$.*

5.5 Exemple spécial : le voyageur de commerce

- Le problème du **voyageur de commerce** cherche, étant donné un ensemble $S = \{s_0, \ldots, s_{n-1}\}$ de points du plan, le chemin $s_{i_0} \to \cdots \to s_{i_{n-1}} \to s_{i_n} = s_{i_0}$ le plus court possible (en distance euclidienne). On note ℓ_S sa longueur.

Lemme 29 (Formule récursive pour le voyageur de commerce) *Si $U \subset S$ avec $s_0, s_j \in U$, on note* $\boldsymbol{\Delta(U, s_j)}$ *la longueur du plus court chemin de s_0 à s_j visitant chaque $s_i \in U$ une fois exactement. Alors* $\ell_S = \min_j \Delta(\{s_0, \ldots, s_{n-1}\}, s_j) + \delta_{j,0}$, $\Delta(\{s_0\}, s_0) = 0$, $\Delta(U, s_0) = +\infty$ *si* $|U| > 1$, *et* $\Delta(U, s_j) = \min_{i \in U : i \neq j} \Delta(U \setminus \{s_j\}, s_i) + \delta_{ij}$ *pour* $0 < j < n$.

- On obtient l'algorithme suivant pour la résolution du voyageur de commerce et le calcul de l'itinéraire

optimal.

Algorithme : TSP(S)
$\Delta \leftarrow$ tableau à deux dimensions, indexé par les sous-ensembles de S contenant $\{s_0\}$, et par les entiers de 0 à $n-1$;
Prec $\leftarrow$ tableau de mêmes dimensions;
$\Delta[\{s_0\}, 0] = 0$;
pour $s = 2$ *à* n **faire**
  pour *tous les* $U \subset S$ *de taille* s *tels que* $s_0 \in U$ **faire**
    $\Delta[U, s_0] = +\infty$;
    pour *tout* $s_j \in U$, $j \neq 0$ **faire**
      $\Delta[U, s_j] = \min\{\Delta[U \setminus \{s_j\}, s_i] + \delta_{ij} : s_i \in U, i \neq j\}$
      Prec$[U, s_j] \leftarrow$ indice de ce minimum;
retourner $\min_j(\Delta[\{s_0, \ldots, s_n\}, s_j]) + \delta_{j0}$, *indice de ce minimum et Prec;*

Algorithme : TSP-REC($S, \Delta, \text{Prec}, j$)
$i_0 \leftarrow 0$;
$i_1 \leftarrow j$;
$U \leftarrow S$;
pour $k = 2$ *à* $n-1$ **faire**
  $i_k \leftarrow \text{Prec}[U, i_{k-1}]$;
  $U \leftarrow S \setminus \{s_{i_{k-1}}\}$;
retourner
  $(i_0, i_1, \ldots, i_{n-1}, i_0)$

Théorème 8 (Voyageur de Commerce) *La longueur minimale d'un chemin passant par n points peut être calculée en temps $O(n^2 2^n)$. Le chemin lui-même peut être calculé en temps $O(n)$ supplémentaire.*

6 Algorithmes de graphes

6.1 Graphes

- Un **graphe** (fini) $\boldsymbol{G = (V, E)}$ est constitué :
  - d'un ensemble (fini) de **sommets** $\boldsymbol{V}$ (ou $\boldsymbol{V(G)}$) de taille $\boldsymbol{n}$ et
  - d'un ensemble d'**arêtes** $\boldsymbol{E}$ (ou $\boldsymbol{E(G)}$), paires d'éléments de V, de taille $\boldsymbol{m}$.
- Deux sommets $x, y \in V$ tels que $\{x, y\} \in E$ sont dits **voisins**, **reliés** ou **adjacents**. On note $\{x, y\} \in E$ ou $\boldsymbol{xy \in E}$ (ou $yx \in E$). L'arête xy est **incidente** aux sommets x et y qui sont ses **extrémités**. Si deux arêtes ont une extrémité en commun, elles sont **adjacentes**, sinon elles sont **disjointes**.
- Les graphes considérés ici ne contiennent ni **boucle** (arête de type xx) ni d'**arête multiple** (arête en plusieurs exemplaires).
- Exemples de bases : chemins, cycles, graphes vides, graphes complets, graphes bipartis...

Lemme 30 (Nbre max d'arêtes) *Tout graphe G vérifie $m \leq \frac{n(n-1)}{2}$.*

- Le **voisinage** de x, noté $\boldsymbol{N_G(x)}$, est l'ensemble des voisins du sommet x.
- Le **degré** d'un sommet v de G est le nombre de voisins de v dans G, on le note $\boldsymbol{deg_G(v)}$.

Lemme 31 (Formule des degrés) *Tout graphe G vérifie $\sum_{v \in V} deg_G(v) = 2m$.*

6.2 Codage

- Généralement, V est codé par $\{1, \ldots, n\}$ ou $\{0, \ldots, n-1\}$, et E peut classiquement être encodé par :
  - **Liste d'arêtes**, de taille $O(m)$, le test d'existence se faisant en $O(m)$.
  - **Liste de voisins** (pour chaque sommet v, on stocke la liste $L(v)$ de ses voisins), de taille $O(m)$, le test d'existence se faisant en $O(m)$.
  - **Matrice d'adjacence** A où $A_{i,j} = 1$ si les sommets i et j sont adjacents, 0 sinon, de taille $O(n^2)$,

le test d'adjacence se faisant en $O(1)$.

6.3 Sous-graphes

- Deux graphes G et G' sont **isomorphes** si il existe une bijection f de $V(G)$ dans $V(G')$ telle que pour tout $x, y \in V(G)$ on ait $xy \in E(G) \Leftrightarrow f(x)f(y) \in E(G')$. La fonction f est un **isomorphisme** entre G et G'. On considèrera (improprement...) que G et G' sont égaux si il existe un isomorphisme entre eux.
- Soient G et H deux graphes.
  Si $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$ alors H est un **sous-graphe** de G.
  Si $V(H) = V(G)$ et $E(H) \subseteq E(G)$ alors H est un **sous-graphe couvrant** de G.
  Si $V(H) \subseteq V(G)$ et $E(H) = \{uv \ : \ uv \in EG(G), \ u \in V(H), \ v \in V(H)\}$ alors H est un **sous-graphe induit** de G.
  On dira (improprement...) que G **contient** H si H est isomorphe à un sous-graphe de G.
- Pour $X \subseteq V(G)$ on note $\boldsymbol{G[X]}$ le sous-graphe induit de G par les sommets de X. On note $\boldsymbol{G \setminus X}$ le graphe $G[V(G) \setminus X]$.

6.4 Connexité, arbres

- Un chemin de G d'extrémités x et y est appelé un **xy-chemin**.
- Un graphe G est **connexe** si pour tous sommets x et y de G, le graphe G contient un xy-chemin.
  Une **composante connexe** de G est un ensemble de sommets de G qui induit un sous-graphe connexe de G et qui est maximal pour cela. Si G est connexe alors il possède une seule composante connexe.
- Un **arbre** est un graphe connexe et sans cycle. Une **forêt** est un graphe sans cycle. Une **feuille** est un sommet ayant exactement un voisin.

Lemme 32 (Propriétés des arbres) *Un arbre ayant au moins deux sommets contient au moins deux feuilles. Une forêt ayant c composantes connexes possède $n - c$ arêtes.*

Théorème 9 (Arbre couvrant) *Un graphe G est connexe ssi il possède un arbre couvrant.*

- Un chemin de longueur minimum entre deux sommets x et y est appelé **un plus court chemin de x à y** et sa longueur est la **distance de x et y**, notée $\boldsymbol{dist_G(x, y)}$.

6.5 Parcours de graphes

- Soit $G = (V, E)$ un graphe connexe, un **parcours** de G est donné par :
  - une énumération $v_1, \ldots, v_n$ des sommets de V
  - une fonction $pere : V \to V$ telle que $pere(v_1) = v_1$ et pour $i \geq 2$: $pere(v_i) \in \{v_1, \ldots, v_{i-1}\}$ et $v_i pere(v_i)$ est une arête de G.

  Le sommet v_1 est appelé la **racine** du parcours et $\{v_i pere(v_i) \ : \ i \geq 2\}$ forme **les arêtes du parcours.**

Lemme 33 (Parcours de graphes) *Tout graphe connexe G admet un parcours et les arêtes de tout parcours de G forment un arbre couvrant de G.*

6.6 Parcours en largeur

- Pour un graphe connexe $G = (V, E)$ et r un sommet de G, un **arbre des plus courts chemins depuis** r est un arbre T couvrant G et vérifiant : pour tout sommet x de G on a $dist_T(r, x) = dist_G(r, x)$. On

note $n_T(x)$ la valeur $dist_T(r, x)$.

Lemme 34 (Arbre des plus courts chemins) *Un arbre T couvrant de G est un arbre des plus courts chemins depuis r si, et seulement si, pour toute arête xy de G, on a $|n_T(x) - n_T(y)| \leq 1$.*

- Un **parcours en largeur** est un parcours obtnenu par application de l'algorithme suivant.

Algorithme : PARCOURS-EN-LARGEUR$(G = (V, E), r)$

pour tous les $v \in V$ **faire** $dv(v) \leftarrow 0$; // sommets déjà vus;
$dv(r) \leftarrow 1$; $ordre(r) \leftarrow 1$; $pere(r) \leftarrow r$; $niv(r) \leftarrow 0$; // la racine
Enfiler r dans AT ; // sommets à traiter, AT gérée comme une file
$t \leftarrow 2$; // le temps
tant que $AT \neq \emptyset$ **faire**
  Prendre v le premier sommet de AT l'enlever de AT ;
  pour tous les $x \in Vois(v)$ **faire**
    si $dv(x) = 0$ **alors**
      $dv(x) \leftarrow 1$; // on traite x pour la première fois
      Enfiler x dans AT, en dernière position ;
      $ordre(x) \leftarrow t$; $t \leftarrow t + 1$;
      $pere(x) \leftarrow v$; $niv(x) \leftarrow niv(v) + 1$;

retourner *ordre, pere et niv;*

Lemme 35 (Parcours en largeur) *L'appel* PARCOURS-EN-LARGEUR$(G = (V, E), r)$ *retourne un arbre des plus courts chemins de G de racine r en temps $O(n + m)$.*

6.7 Parcours en profondeur

- Etant donné un graphe $G = (V, E)$, r un sommet de G et T un arbre couvrant de G enraciné en r, on dit que x est un **ancêtre** de y si x appartient au chemin de r à y dans T.
  La **branche issue** de x est l'ensemble des sommets qui admette x comme ancêtre.
- Un arbre couvrant T de G, enraciné en r est **normal** si pour toute arête xy de G on a x appartient à la branche de y ou y appartient à la branche de x.

Algorithme : PARCOURS-EN-PROFONDEUR$(G = (V, E), r)$

pour tous les $x \in V$ **faire** $dv(v) \leftarrow 0$; // sommets déjà vus;
$dv(r) \leftarrow 1$; $debut(r) \leftarrow 1$; $pere(r) \leftarrow r$; // la racine
Empiler r sur AT ; // sommets à traiter, AT gérée comme une pile
$t \leftarrow 2$; // le temps
tant que $AT \neq \emptyset$ **faire**
  Noter x le sommet en haut de AT ;
  si $vois(x) = \emptyset$ **alors**
    Dépiler AT ;
    $fin(x) \leftarrow t$; $t \leftarrow t + 1$; // fin de traitement pour x
  sinon
    Noter y le sommet en haut de $vois(x)$ et dépiler $vois(x)$;
    si $dv(y) = 0$ **alors**
      $dv(y) \leftarrow 1$; // on traite y pour la première fois
      Empiler y sur AT ;
      $debut(y) \leftarrow t$; $t \leftarrow t + 1$;
      $pere(y) \leftarrow x$;

Lemme 36 (Parcours en profondeur) *L'appel* PARCOURS-EN-PROFONDEUR$(G = (V, E), r)$ *retourne un arbre normal de* G *de racine* r *en tmeps* $O(n + m)$.

6.8 Plus court chemins dans les graphes valués, algorithme de Dijkstra

- On se donne un graphe $G = (V, E)$ donné par liste de voisin avec l une fonction de longueur **positive** sur les arêtes, et r un sommet de G, la racine.

Algorithme : DIJKSTRA$(G = (V, E), r)$
pour tous les $v \in V$ **faire**
  $d(v) \leftarrow +\infty$;
  $traite(v) \leftarrow 0$; // pour marquer les sommets traités
$pere(r) \leftarrow r$; $d(r) \leftarrow 0$; // la racine
tant que *il existe* x *avec* $traite(x) = 0$ **faire**
  Choisir un tel x avec $d(x)$ minimum ;
  $traite(x) \leftarrow 1$;
  pour tous les $y \in Vois(x)$ **faire**
    si $traite(y) = 0$ *et* $d(y) > d(x) + l(xy)$ **alors**
      $d(y) \leftarrow d(x) + l(xy)$; // x est un raccourci pour atteindre y
      $pere(y) \leftarrow x$;

Lemme 37 (Algorithme de Dijkstra) *L'appel* DIJKSTRA$(G = (V, E, l), r)$ *retourne un arbre des plus courts chemins valués de* G *de racine* r. *Sa complexité est en* $O(n^2)$.

- Si on gère 'la frontière' des sommets traités (c-à-d les sommets v avec $traite(v) = 0$ et $dist(v) < +\infty$) par un tas alors on obtient une complexité en $O(m \log n)$.