Notes de cours

Présentation du module

Chapitre 1 Introduction, modèle et outils mathématiques (1CM, 2TD)
Chapitre 2 Structures de données : arbres binaires et graphes (2CM, 3TD, 2TP)
Chapitre 3 Algorithmes gloutons (2CM, 3TD, 1TP)
Chapitre 4 Diviser pour régner (2CM, 2TD, 1TP)
Chapitre 5 Programmation dynamique (2CM, 2TD, 1TP)

Prérequis. Il est nécessaire de connaître les structures de données linéaires présentées dans les cours précédents : tableaux, piles, files, liste chaînées. Les structures d'arbres (binaires), de tas et de graphes, sont étudiées en détail dans ce cours, donc une connaissance superficielle peut aider. Il faut d'autre part maîtriser les structures de contrôles classiques (boucles et branchements) et la récursivité.

Bibliographie. 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 et de nombreux exemplaires sont disponibles à la BU).
— S. Dasgupta, C. Papadimitriou and U. Vazirani. **Algorithms**, *McGraw-Hill Higher Education*, 2006.
— J. Erickson. **Algorithms**, University of Illinois at Urbana-Champaign, 2019. *En ligne :* `http://algorithms.wtf`

1 Introduction, rappels

1.1 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 **conditionnelles** (*si* ... *alors* ... *sinon* ...) et des **boucles** (*pour* et *tant que*).
- Deux modèles étudiées :
  – *Word-RAM* : chaque **opération élémentaire** prend un **temps constant** (quasi-systématiquement utilisé) ;
  – *RAM* : chaque **opération sur un chiffre ou un bit** prend un **temps constant** (utilisé une fois dans le cours).
- Dans un modèle, on compte (ou majore) **le nombre d'opérations élémentaires** (pour établir la **complexité en temps** de l'algo), exprimer ces valeurs **en fonction des paramètres d'entrée** de l'algorithme, **de manière asymptotique** et dans le **pire des cas**.

1.2 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 enfin sa **correction**. Pour ce dernier point, on cherche souvent un **invariant de l'algorithme** que l'on prouve généralement par **récurrence**. La terminaison est souvent omise car obtenue avec la complexité et la correction.

1.3 Outils mathématiques pour l'analyse de complexité

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

Lemme 1 (Calcul avec des « grand O »). *$O(f) + O(g) = O(f+g)$; si $h = O(f)$ alors $f + h = O(f)$; $O(f) \times O(g) = O(f \times g)$; pour $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$.*

Lemme 2 (« Grand O » et limites). *Soit $f : \mathbb{N} \to \mathbb{R}_+$ et $g : \mathbb{N} \to \mathbb{R}_+^*$ et $\ell = \lim_{+\infty} \frac{f(n)}{g(n)}$. Alors $f = O(g)$ si $\ell < +\infty$ et $f \neq O(g)$ si $\ell = +\infty$.*

Lemme 3 (Limite de l'inverse). *Si f et g sont des fonctions strictement positives telles que $f/g \to_\infty \ell$, alors $g/f \to_\infty 1/\ell$ (où $1/0 = +\infty$ et $1/\infty = 0$).*

- **Autre notation** : Soient $f, g : \mathbb{N} \to \mathbb{R}_+$. On dit que $\boldsymbol{f = \Omega(g)}$ si $g = O(f)$.

Lemme 4 (Croissances comparées). *Pour toutes **constantes** $\alpha, \beta > 0$,*

$(\log n)^\alpha / n^\beta \underset{n\to+\infty}{\longrightarrow} 0$ $\qquad$ $n^\alpha/(2^n)^\beta \underset{n\to+\infty}{\longrightarrow} 0$ $\qquad$ $(2^n)^\alpha/n! \underset{n\to+\infty}{\longrightarrow} 0$

Si $f(n) \to_\infty +\infty$ et $0 < \alpha < \beta$,

$$\lim_{+\infty} f(n)^\alpha / f(n)^\beta = \lim_{+\infty} f(n)^{\alpha-\beta} = 0.$$

Lemme 5 (Règles de calcul pour le log). *Pour $a, b \in \mathbb{R}_+^*$ et $k \in \mathbb{R}$,*

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

Lemme 6 (Règles de calcul pour l'exp). *Pour $a, b \in \mathbb{R}_+$,*

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

Lemme 7 (Formule de Stirling). *Pour $n \in \mathbb{N}^*$, $n! = n \times (n-1) \times \cdots 2 \times 1$ et vérifie $n! \underset{n\to+\infty}{\sim} \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.*

- Pour $x \in \mathbb{R}$, $\lfloor x \rfloor$ est 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, $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.

Lemme 8 (Sommes arithmétique et géométrique). *Pour $a, b \in \mathbb{N}$ avec $a \leq b$ et $x \in \mathbb{R} \setminus \{1\}$,*

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

2 Structures de données arborescentes

2.1 Arbres binaires

- Mathématiquement, un **arbre binaire** est défini récursivement : soit l'arbre vide $\emptyset$; soit une **racine**, un **sous-arbre gauche** G et d'un **sous-arbre droit** D qui sont deux arbres binaires.
- Informatiquement, un **nœud** x est soit le **nœud vide** $\emptyset$, soit un nœud non vide défini par une **valeur** val(x) et trois **pointeurs** vers d'autres nœuds père(x), filsG(x) et filsD(x) tels que père(filsG(x)) $= x$ si filsG(x) $\neq \emptyset$, et père(filsD(x)) $= x$ si filsD(x) $\neq \emptyset$.
  Un **arbre binaire** A est représenté par une **racine** rac(A) qui est un nœud tel que père(rac(A)) $= \emptyset$.
- Une **feuille** est un nœud dont les fils gauche et droit sont tous les deux le nœud vide $\emptyset$. Un **nœud interne** est un nœud qui n'est ni la racine, ni une feuille. La **hauteur** $\boldsymbol{h(x)}$ d'un nœud x est définie récursivement par

$h(\text{rac}(A)) = 0$ et $h(x) = h(\text{père}(x)) + 1$ sinon. La **hauteur de** A est $\boldsymbol{h(A)} = \max\{h(x) \ : \ x \in A\}$. Pour $0 \leq k \leq h(A)$, le $k^{\text{ème}}$ **niveau de** A est l'ensemble $\boldsymbol{N_k} = \{x \in A \ : \ h(x) = k\}$. On note $\boldsymbol{n(A)}$ le **nombre de nœuds** de A.

Lemme 9 (Nombre de nœuds). *Pour $0 \leq k \leq h(A)$, $|N_k| \leq 2^k$. Pour tout arbre binaire A, $h(A)+1 \leq n(A) \leq 2^{h(A)-1}+1$.*

Algorithme : ParcoursInfixe(x)

si $x \neq \emptyset$ **:**
- ParcoursInfixe(filsG(x))
- Afficher val(x)
- ParcoursInfixe(filsD(x))

Algorithme : AlgoGénérique(x)

res ← valeur pour l'arbre vide
si $x \neq \emptyset$ **:**
- $\text{res}_G \leftarrow$ AlgoGénérique(filsG(x))
- $\text{res}_D \leftarrow$ AlgoGénérique(filsD(x))
- res $\leftarrow f(\text{res}, \text{res}_G, \text{res}_D, x)$

renvoyer *res*

Algorithme : ParcoursLargeur(x)

F ← file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
- y ← défiler un élément de F
- Afficher val(y)
- **si** filsG(y) $\neq \emptyset$ **:** l'ajouter à F
- **si** filsD(y) $\neq \emptyset$ **:** l'ajouter à F

Théorème 1. ParcoursInfixe*,* AlgoGénérique *et* ParcoursLargeur *ont une complexité $O(n(A))$ (si f est $O(1)$).*

2.2 Graphes

- Un **graphe** $G = (S, A)$ est constitué d'un ensemble S de **sommets** et d'un ensemble A d'**arêtes**. Un arête est un ensemble $\{u, v\} \subset S$ de deux sommets, notée uv ou vu. Les **voisins** d'un sommet u sont les sommets v tels que $uv \in A$. Le **degré** d'un sommet est son nombre de voisins. Un **chemin de longueur** $\boldsymbol{k}$ de u à v est un ensemble d'arêtes uu_1, u_1u_2, ..., $u_{k-1}v$. Un **cycle** est un chemin de u à u. Deux sommets u et v sont à **distance** $\boldsymbol{d}$ s'il existe un chemin de longueur d de u à v et aucun chemin de longueur $< d$.
- Un graphe est **connexe** s'il existe toujours un chemin entre deux sommets u et v. La **composante connexe** d'un sommet u est l'ensemble de sommets v tels qu'il existe un chemin entre u et v.
- Informatiquement, un graphe admet deux représentations. En numérotant les sommets de 1 à n,
  - **matrice d'adjacence** : matrice M telle que $M_{ij} = 1$ si $ij \in A$ et $M_{ij} = 0$ sinon ;
  - **listes d'adjacence** : tableau R de taille n, qui contient en case $T_{[i]}$ la liste chaînée des voisins de i.

Algorithme : ParcoursLargeur(G, s)

F ← file vide
Ajouter s à F et *marquer* s
tant que F *est non vide* **:**
- u ← défiler un élément de F
- Afficher u
- **pour** *tout voisin* non marqué v *de* u **:**
  - Ajouter v à F et *marquer* v

Algorithme : ParcoursProfondeur(G, s)

P ← pile vide
Ajouter s à P et *marquer* s
tant que P *est non vide* **:**
- u ← défiler un élément de P
- Afficher u
- **pour** *tout voisin* non marqué v *de* u **:**
  - Ajouter v à P et *marquer* v

Théorème 2. ParcoursLargeur(G, s) *et* ParcoursProfondeur(G, s) *affichent une fois et une seule chaque sommet de la composante connexe de s. Leur complexité est $O(n^2)$ si le graphe est représenté par matrice d'adjacence, $O(m + n)$ si le graphe est représenté par listes d'adjacence, où m est le nombre d'arêtes et n le nombre de sommets.*

- Un arbre binaire **est** un graphe particulier : connexe, sans cycle, et avec des sommets de degrés 1, 2 ou 3. Mais un arbre binaire **n'est pas** un graphe : un sommet est désigné comme racine, les fils gauche et droit sont distingués, et la défintion récursive et la représentation informatiques sont bien différentes.

Algorithme : MINIMUM(x)

```
tant que filsG(x) ≠ ∅ :
    x ← filsG(x)
renvoyer x
```

Algorithme : INSÉRER(A, z)

```
x ← rac(A)
p ← ∅
tant que x ≠ ∅ :
    p ← x
    si val(z) < val(x) :
    x ← filsG(x)
    sinon : x ← filsD(x)
père(z) ← p
si p = ∅ : rac(A) ← z
sinon si val(z) < val(p) :
    filsG(p) ← z
sinon : filsD(p) ← z
```

Algorithme : SUCCESSEUR(x)

```
si filsD(x) ≠ ∅ :
    renvoyer MINIMUM(filsD(x))
y ← père(x)
tant que p ≠ ∅ et x = filsD(y) :
    x ← p
    y ← père(x)
renvoyer y
```

Algorithme : REMPLACE(A, x, z)

```
p ← père(x) ; père(x) ← ∅
si p = ∅ : rac(A) ← z
sinon si x = filsG(p) :
    filsG(p) ← z
sinon : filsD(p) ← z
si z ≠ ∅ : père(z) ← p
```

Algorithme : RECHERCHER(x, k)

```
tant que x ≠ ∅ et val(x) ≠ k :
    si k < val(x) :  x ← filsG(x)
    sinon :  x ← filsD(x)
renvoyer x
```

Algorithme : SUPPRIMER(A, z)

```
si filsG(z) = ∅ :
    REMPLACE(A, z, filsD(z))
sinon si filsD(z) = ∅ :
    REMPLACE(A, z, filsG(z))
sinon :
    y = SUCCESSEUR(z)
    REMPLACE(A, y, filsD(y))
    filsD(y) ← filsD(z) ; filsD(z) ← ∅
    filsG(y) ← filsG(z) ; filsG(z) ← ∅
    si filsD(y) ≠ ∅ : père(filsD(y)) = y
    si filsG(y) ≠ ∅ : père(filsG(y)) = y
    REMPLACE(A, z, y)
```

2.3 Arbres binaires de recherche

- On veut stocker un **ensemble ordonné** de n valeurs avec les opérations INSÉRER et SUPPRIMER, MINIMUM et MAXIMUM, RECHERCHER, et SUCCESSEUR et PRÉDÉCESSEUR.
- Si A est un arbre binaire, on note **saG(A)** son **sous-arbre gauche** et **saD(A)** son **sous-arbre droit**.
- Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que pour tout nœud x,
  - — pour tout nœud $y \in \text{saG}(x)$, $\text{val}(y) \leq \text{val}(x)$, et
  - — pour tout nœud $z \in \text{saD}(x)$, $\text{val}(z) \geq \text{val}(x)$.

Lemme 10 (PARCOURSINFIXE d'un ABR)**.** *Le parcours infixe d'un arbre binaire A affiche les valeurs de A triées si et seulement si A est un ABR.*

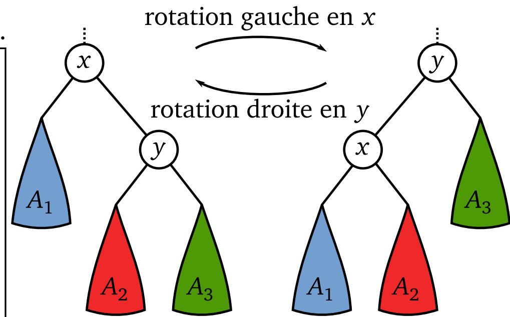
Lemme 11 (Complexité des opérations sur les ABR)**.** *Les algorithmes* RECHERCHER*,* MINIMUM*,* SUCCESSEUR*,* INSÉRER *et* SUPPRIMER *ont complexité* $O(h(A))$.

Lemme 12 (Correction de SUCCESSEUR)**.** *Si A est un ABR et x un nœud de A,* SUCCESSEUR(x) *renvoie un nœud de valeur minimale parmi l'ensemble* $\{y \in A : \text{val}(y) \geq \text{val}(x)\}$. *Si l'ensemble est vide, l'algorithme renvoie* $\emptyset$.

Lemme 13 (Conservation de la structure)**.** *Si A est un ABR, il garde la structure d'ABR après application de* INSÉRER(A, z) *et* SUPPRIMER(A, z).

- Un arbre binaire de recherche est **équilibré** si $\boldsymbol{h(A) = O(\log(n(A)))}$.

> **Lemme 14** (Rotation). *Après une rotation gauche en un nœud x d'un arbre binaire de recherche A, l'arbre conserve la structure d'ABR et*
> — *la hauteur de chaque nœud de A_1 est augmentée de 1,*
> — *la hauteur de chaque nœud de A_2 est inchangée, et*
> — *la hauteur de chaque nœud de A_3 est diminuée de 1.*
> *Les résultats sont symétriques pour la rotation droite. Les rotations s'effectuent en temps $O(1)$.*



2.4 Tas

- Un arbre binaire est **quasi-complet** si pour tout $k < h(A)$, $|N_k| = 2^k$ et les nœuds de $N_{h(A)}$ sont « le plus à gauche possible ». Il est **complet** si $|N_{h(A)}| = 2^{h(A)}$.

> **Lemme 15** (Haut. d'un quasi-complet). *Si A est quasi-complet, $2^{h(A)} \leq n(A) \leq 2^{h(A)+1} - 1$, et $h(A) = \lfloor \log n(A) \rfloor$.*

- On numérote tout nœud x d'un arbre en définissant récursivement $\mathrm{num}(x)$ par $\mathrm{num}(\mathrm{rac}(A)) = 0$, $\mathrm{num}(\mathrm{filsG}(x)) = 2\,\mathrm{num}(x) + 1$ si $\mathrm{filsG}(x) \neq \emptyset$ et $\mathrm{num}(\mathrm{filsD}(x)) = 2\,\mathrm{num}(x) + 2$ si $\mathrm{filsD}(x) \neq \emptyset$ (**parcours en largeur**).

> **Lemme 16** (Num des liens). *Un arbre est quasi-complet si et seulement si ses nœuds sont numérotés de 0 à $n(A) - 1$.*

- Un arbre quasi-complet est représenté par un tableau A tel que $A[\mathrm{num}(x)] = \mathrm{val}(x)$ pour tout nœud x.
  On identifie un arbre quasi-complet et le tableau qui le représente : on pose $\mathrm{rac}(A) = 0$, $\mathrm{filsG}(i) = 2i + 1$, $\mathrm{filsD}(i) = 2i + 2$, $\mathrm{père}(i) = \lfloor (i-1)/2 \rfloor$, $\mathrm{val}(i) = A[i]$ et $h(i) = \lfloor \log(i+1) \rfloor$.
- Un arbre a la **propriété de tas max** si pour tout $x \neq \mathrm{rac}(A)$, $\mathrm{val}(\mathrm{père}(x)) \geq \mathrm{val}(x)$.
  Un arbre a la **propriété de tas min** si pour tout $x \neq \mathrm{rac}(A)$, $\mathrm{val}(\mathrm{père}(x)) \leq \mathrm{val}(x)$.
- Un **tas max** est un arbre binaire quasi-complet A qui vérifie la propriété de tas max.
  Un **tas min** est un arbre binaire quasi-complet A qui vérifie la propriété de tas min.

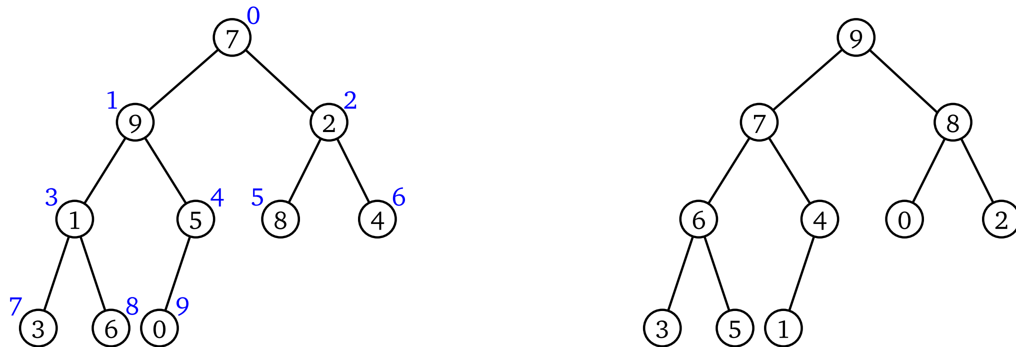


FIGURE 1 – À gauche : Arbre quasi-complet numéroté, représenté par le tableau $[7, 9, 2, 1, 5, 8, 4, 3, 6, 0]$.
À droite : Tas max, représenté par le tableau $[9, 7, 8, 6, 4, 0, 2, 3, 5, 1]$.

> **Lemme 17.** REMONTER(T, i) *a une complexité $O(\log(n(T)))$. Si $i = n(T) - 1$ et que T privé de i est un tas, alors T est un tas après exécution de* REMONTER(T, i).

Algorithme : ENTASSER(T, i)
tant que filsG$(i) < n(T)$:
 $(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
 si $T_{[g]} > T_{[m]}$: $m \leftarrow g$
 si $d < n(T)$ *et* $T_{[d]} > T_{[m]}$: $m \leftarrow d$
 si $m \neq i$: Échanger $T_{[i]}$ et $T_{[m]}$
 sinon : $i \leftarrow n(T)$

Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)

Algorithme : SUPPRIMER(T, i)
$x \leftarrow T_{[i]}$
$T_{[i]} \leftarrow T_{[n(T)-1]}$
Réduire T d'une case
ENTASSER(T, i)
REMONTER(T, i)
renvoyer x

Algorithme : REMONTER(T, i)
tant que $i > 0$ *et* $T_{[\text{père}(i)]} < T_{[i]}$:
 Échanger $T_{[i]}$ et $T_{[\text{père}(i)]}$
 $i \leftarrow \text{père}(i)$

Algorithme : TRITAS(T)
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 : ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 : $S[i] \leftarrow$ SUPPRIMER$(T, 0)$
renvoyer S

Lemme 18. ENTASSER(T, i) *a une complexité* $O(\log(n(T)))$. *Si les sous-arbres gauche et droit de* i *sont des tas intialement, l'arbre enraciné en* i *est un tas après exécution de* ENTASSER(T, i).

Théorème 3 (Tri par tas)**.** *Si* T *est un tableau quelconque,* TRITAS(T) *renvoie le tableau* T *trié. Sa complexité est* $O(n \log n)$ *où* n *est la taille de* T.

- Le tri par tas peut être implémenté de sorte à modifier le tableau T lui-même plutôt qu'en créer un autre.

Lemme 19 (Borne inf. pour le tri)**.** *Tout algorithme de tri ne faisant que des comparaisons effectue* $\Omega(n \log n)$ *comparaisons dans le pire des cas pour trier* n *nombres quelconques.*

- Une **file de priorité** est une structure de donnée contenant des éléments ayant des priorités et possédant des fonctions pour les opérations suivantes : AJOUTER un élément, EXTRAIRE l'élément de priorité maximale (ou minimale), CHANGER la priorité d'un élément.

Lemme 20 (Tas pour les files de priorité)**.** *On peut implanter toutes les opérations d'une file de priorité à l'aide d'un tas, et chacune a alors complexité* $O(\log n)$ *pour une file de* n *éléments.*

3 Algorithmes gloutons

3.1 Qu'est-ce qu'un algorithme glouton ?

- Un **algorithme glouton** fait à chaque étape un choix **localement optimal** dans le but d'obtenir à la fin un **optimum global**.
- La conception d'un algorithme glouton se fait en quatre étapes : décider d'un **choix glouton** ; chercher un cas où **ça ne marche pas** et retourner à l'étape 1 ; sinon, **démontrer** que l'algorithme est correct ; étudier sa **complexité**.
- Le problème générique des algorithmes glouton et son algorithme sont les suivants :

Entrée Un ensemble fini X, avec une valeur v_x pour tout $x \in X$.

Paramètre Une propriété $\mathscr{P}$ définissant les sous-ensembles $A \subset X$ **acceptables**, telle que si A vérifie $\mathscr{P}$, toute sous-solution $B \subset A$ vérifie également $\mathscr{P}$ (propriété *monotone vers le bas*).

Sortie Une sous-ensemble acceptable $A \subset X$ qui maximise/minimise $v_A = \sum_{x \in A} v_x$ (parmi les sous-ensembles acceptables).

> **Algorithme :** GLOUTONGÉNÉRIQUE$(X, \mathscr{I})$
> Trier X par *valeurs* croissantes + critère glouton
> $S \leftarrow \emptyset$
> **pour** $x \in X$ *(dans l'ordre du tri)* **:**
>   **si** $S \cup \{x\}$ *vérifie* $\mathscr{P}$ **:**
>   ajouter x à S // $S \leftarrow S \cup \{x\}$
> **renvoyer** S

Théorème 4 (Théorème des algorithmes gloutons)**.** *On considère le problème générique avec la propriété $\mathscr{P}$. Si pour toute entrée X, il existe une solution optimale S tq*

- *le premier élément x_0 de X, trié, appartienne à S, et*
- *$S \setminus x_0$ soit une solution optimale sur l'entrée $X \setminus x_0$ du problème de paramètre $\mathscr{P}'$ où A vérifie $\mathscr{P}'$ ssi $A \cup \{x_0\}$ vérifie $\mathscr{P}$*

alors GLOUTONGÉNÉRIQUE *est optimal.*

3.2 Premier exemple : choix de cours

- Le problème du **choix de cours** et son algorithme glouton sont les suivants :

Entrée Un ensemble C de cours $C_i = (d_i, f_i)$, où d_i est l'horaire de début et f_i l'horaire de fin.

Sortie Un ensemble ordonné maximal de cours $(C_{i_1}, \ldots, C_{i_k})$ tels que pour tout $j < k$, $f_{i_j} \leq d_{i_{j+1}}$ (les cours sont *compatibles* entre eux).

- **Choix glouton** : choisir le cours qui finit le plus tôt.

> **Algorithme :** CHOIXCOURSGLOUTON(C)
> Trier C en fonction des fins
> $I \leftarrow \{0\}$ `// Indice des cours choisis`
> $f \leftarrow$ FIN$(C_{[0]})$ `// Fin du dernier cours choisi`
> **pour** $i = 1$ *à* $n-1$ **:**
>   **si** DÉBUT$(C_{[i]}) \geq f$ **:**
> $I \leftarrow I \cup \{i\}$
> $f \leftarrow$ FIN$(C_{[i]})$
> **renvoyer** I

Lemme 21 (Sous-solution optimale)**.** *Il existe une solution optimale au problème de choix de cours qui est formée du cours C_{i_1} se terminant le plus tôt et d'une solution optimale du problème restreint aux cours i avec $d_i > f_{i_1}$.*

Théorème 5 (Choix de cours)**.** *L'algorithme* CHOIXCOURSGLOUTON *résout le problème du choix de cours de manière optimale, et en temps $O(n \log n)$ pour n activités.*

3.3 Deuxième exemple : problème du sac à dos fractionnaire

- Le problème du **sac-à-dos fractionnaire** et son algorithme glouton sont les suivants :

Entrée Un ensemble d'objets O_0, ..., O_{n-1} ayant chacun une taille t_i et une valeur v_i, et une taille T de sac-à-dos.

Sortie Une fraction $x_i \in [0,1]$ pour chaque objet, telle que le total ne dépasse pas la taille du sac ($\sum_{i=0}^{n-1} x_i t_i \leq T$) et qui maximise la valeur totale ($V = \sum_{i=0}^{n-1} x_i v_i$).

> **Algorithme :** SÀDFRACGLOUTON(O, T)
> Trier les objets $O_i = (t_i, v_i)$ par v_i / t_i **décroissant**
> $R \leftarrow T$ `// Reste libre dans le sac-à-dos`
> **pour** $i = 0$ *à* $n-1$ *(dans l'ordre du tri)* **:**
>   **si** $t_i \leq R$ **:** $x_i \leftarrow 1$; $R \leftarrow R - t_i$
>   **sinon :** $x_i \leftarrow R / t_i$; $R \leftarrow 0$
> **renvoyer** $(x_0, \ldots, x_{n-1})$

- **Choix glouton** : choisir l'objet de meilleur rapport valeur-poids.

Lemme 22 (Sous-solution optimale)**.** *Il existe une solution optimale au problème de sac à dos fractionnaire* $(T,(v_0,t_0),\dots,(v_{n-1},t_{n-1}))$ *avec* $v_1/t_1 \geq \cdots \geq v_n/t_n$ *qui est donnée par :*
- *— si* $t_0 \leq T$*, alors* $x_0 = 1$ *et* $(x_1,\dots,x_{n-1})$ *est une solution optimale du problème* $(T - t_0,(v_1,t_1),\dots,(v_{n-1},t_{n-1}))$*;*
- *— si* $t_0 > T$*, alors* $x_0 = T/t_0$ *et* $x_1 = \cdots = x_{n-1} = 0$.

Théorème 6 (Sac-à-dos fractionnaire)**.** *L'algorithme* SÀDFRACGLOUTON *résout le problème du sac-à-dos fractionnaire de manière optimale, et en temps* $O(n\log n)$ *pour* n *objets.*

3.4 Exemple spécial : approximation de SETCOVER dans le plan

- Le problème **SETCOVER dans le plan** est le suivant :

  Entrée n maisons placées dans le plan.

  Sortie Un ensemble minimal de maisons où placer une antenne Wifi, tel que chaque antenne a une portée de 500 m et toutes les maisons doivent être couvertes.

- **Choix glouton** : placer à chaque étape l'antenne qui couvre le plus de maisons non déjà couvertes.

Théorème 7 (Approximation de SETCOVER)**.** *Si* k_{opt} *désigne le nombre d'antennes à placer dans une solution optimale, alors le choix glouton proposé place au plus* $k_{opt}\ln n$ *antennes.*

3.5 Dernier exemple : arbre couvrant de poids minimal

- Un **arbre couvrant** d'un graphe connexe $G = (S,A)$ est un sous-ensemble d'arêtes $B \subset A$ tel que $T = (S,B)$ soit un arbre (en particulier connexe).
- Le problème de l'**arbre couvrant de poids minimum** est le suivant, il est résolu par l'**algorithme de KRUSKAL** :

  Entrée Un graphe **pondéré** $G = (S,A,p)$ où $p : A \to \mathbb{R}_+$.

  Sortie Un arbre couvrant $T = (S,B)$ de G qui minimise $P = \sum_{e\in B} p(e)$

Algorithme : KRUSKAL(S,A,p)

```
Trier les arêtes par poids croissants
B ← ∅                                   // aucune arête
pour chaque sommet v ∈ S :
    c[v] ← v
    P_c[v] ← Pile {v}                   // un seul élément
    t_c[v] ← 1                          // taille de la pile
pour chaque arête e = uv ∈ A dans l'ordre :
    si c[u] ≠ c[v] :                    // e ne crée pas de cycle
        Ajouter e à B
        UNION(c[u], c[v])
```

Algorithme : UNION(x,y)

```
si t_x > t_y : UNION(y, x)
sinon :
    t_y ← t_y + t_x
    tant que P_x est non vide :
        w ← dépiler P_x
        c[w] ← y
        Empiler w sur P_y
```

Théorème 8 (Algorithme de KRUSKAL)**.** *L'algorithme* KRUSKAL(S,A,p) *calcule un arbre couvrant de poids minimum du graphe pondéré* $G = (S,A,p)$ *en temps* $O(m\log n)$ *où* m *est le nombre d'arêtes et* n *le nombre de sommets.*

4 Diviser pour régner

4.1 Qu'est-ce que « diviser pour régner » ?

- La stratégie « **diviser pour régner** » consiste à résoudre un problème en trois étapes :
    1. **diviser** le problème en sous-problèmes,
    2. **résoudre récursivement** ces sous-problèmes, et
    3. **combiner** les solutions pour reconstruire la solution du problème original.
- L'ensemble des appels récursifs définit l'**arbre de récursion** de l'algorithme, dont chaque nœud représente un des sous-problèmes à résoudre. Les feuilles sont les cas de base de l'algorithme.
  On peut noter sur chaque nœud le temps nécessaire pour combiner les solutions des sous-problèmes du nœud : la somme de ces temps fournit le temps de calcul total de l'algorithme.
- Les preuves de correction, terminaison et complexité s'effectuent par récurrence sur la taille du problème à traiter. Pour la complexité, on utilise le « *master theorem* ».

Théorème 9 (*Master Theorem*)**.** *S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$, $T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$, alors*

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

4.2 Premier exemple : algorithme du tri fusion

```
Algorithme : TriFusion(T)
n ← taille(T)
si n = 1 : renvoyer T
sinon :
    T_1 ← TriFusion(T_[0,⌊n/2⌋[)
    T_2 ← TriFusion(T_[⌊n/2⌋,n[)
    renvoyer Fusion(T_1, T_2)
```

- La complexité de TRIFUSION vérifie l'équation de récurrence

$$t(n) \leq 2t(\lceil n/2 \rceil) + O(n).$$

```
Algorithme : Fusion(T_1, T_2)
n_1 ← taille(T_1) ; n_2 ← taille(T_2)
S ← tableau de taille n_1 + n_2
i_1 ← 0 ; i_2 ← 0
pour i_S = 0 à n_1 + n_2 − 1 :
    si i_2 ≥ n_2 ou (i_1 < n_1 et T_1[i_1] ≤ T_2[i_2]) :
        S[i_S] ← T_1[i_1]
        i_1 ← i_1 + 1
    sinon :                          // i_1 ≥ n_1 ou T_1[i_1] > T_2[i_2]
        S[i_S] ← T_2[i_2]
        i_2 ← i_2 + 1
renvoyer S
```

Théorème 10 (Tri fusion)**.** *L'algorithme* TRIFUSION *trie le tableau T de taille n fourni en paramètre en temps* $O(n \log n)$.

4.3 Deuxième exemple : multiplication d'entiers

- L'algorithme de l'école primaire pour multiplier deux nombres écrits en base 10 effectue n^2 multiplications chiffre à chiffre, pour des entrées de taille n, et a une complexité $O(n^2)$.
- L'**algorithme de Karatsuba** est basé sur les égalités suivantes : si $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$ et $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$, alors

$$A \times B = A_0B_0 + 10^{\lfloor n/2 \rfloor}(A_0B_1 + A_1B_0) + 10^{2\lfloor n/2 \rfloor}A_1B_1 \qquad \text{et} \qquad A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1).$$

Algorithme : KARATSUBA(A, B)

si *A et B n'ont qu'un chiffre* **: renvoyer** $a_0 b_0$
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$(|A_0 - A_1|, |B_0 - B_1|)$
$s \leftarrow$ signe$(A_0 - A_1) \times$ signe$(B_0 - B_1)$
renvoyer $C_{00} + 10^{\lfloor n/2 \rfloor}(C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$

- L'algorithme de Karatsuba s'étudie dans le *modèle RAM*.
- Sa complexité vérifie l'équation de récurrence

$$K(n) \leq 3K(\lceil n/2 \rceil) + O(n).$$

- En pratique, l'algorithme de Karatsuba est utilisé avec des *mots machine* comme chiffres, c'est-à-dire que les entiers sont écrits en base 2^{64} (ou 2^{32} pour des ordinateurs plus anciens) plutôt qu'en base 10.

Théorème 11 (Algorithme de Karatsuba)**.** *L'algorithme de* KARATSUBA *permet de multiplier deux entiers de n chiffres en temps* $O(n^{\log 3})$ *(avec* $\log 3 \simeq 1,585$*).*

4.4 Exemple spécial : calcul de rang

- Pour un tableau T de n nombres, on note **rang(k, T)** le $\boldsymbol{k}^{\text{ème}}$ **plus petit élément** de T. Par exemple, rang$(1, T)$ est le minimum de T, rang(n, T) est son maximum et rang$(\lfloor n/2 \rfloor, T)$ est **la médiane** de T.
- Le choix d'un élément p de T comme **pivot** permet de séparer T en 3 tableaux :
  — $\boldsymbol{T}_{\text{inf}}$ qui contient les éléments x de T vérifiant $x < p$,
  — $\boldsymbol{T}_{\text{eq}}$ qui contient les éléments x de T vérifiant $x = p$ et
  — $\boldsymbol{T}_{\text{sup}}$ qui contient les éléments x de T vérifiant $x > p$.
  On note $\boldsymbol{n}_{\text{inf}}$, $\boldsymbol{n}_{\text{eq}}$ et $\boldsymbol{n}_{\text{sup}}$ les tailles respectives de ces trois tableaux avec $n_{\text{inf}} + n_{\text{eq}} + n_{\text{sup}} = n$
- On a alors les formules et l'algorithme suivants :

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Algorithme : RANG(T, k)

si $k = 1$ **: renvoyer** $T_{[0]}$
$p \leftarrow$ CHOIXPIVOT(T)
$n_{\text{inf}} \leftarrow 0$, $n_{\text{eq}} \leftarrow 0$
pour $i = 0$ *à* $n - 1$ **:**
  si $T_{[i]} < p$ **:** $n_{\text{inf}} \leftarrow n_{\text{inf}} + 1$
  sinon si $T_{[i]} = p$ **:** $n_{\text{eq}} \leftarrow n_{\text{eq}} + 1$
si $k \leq n_{inf}$ **:** Calculer T_{inf} et **renvoyer** RANG(T_{inf}, k)
sinon si $n_{inf} < k \leq n_{inf} + n_{eq}$ **: renvoyer** p
sinon : Calculer T_{sup} et **renvoyer** RANG$(T_{sup}, k - n_{inf} - n_{eq})$

Algorithme : CHOIXPIVOT(T)

renvoyer $T_{[0]}$

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$
renvoyer $T_{[j]}$

- Il existe d'autres CHOIXPIVOT possibles.

Théorème 12 (Calcul de rang)**.** RANG*(T, k) retourne le $k^{\text{ème}}$ plus petit élément de T. En fonction de* CHOIXPIVOT*, sa complexité peut être*
  — $O(n^2)$ *dans le pire des cas mais* $O(n)$ *« en moyenne » (pivot* $T_{[0]}$*),*
  — $O(n)$ *avec bonne probabilité, pour tout tableau (pivot aléatoire),*
  — $O(n)$ *de manière déterministe, pour tout tableau (pivot « médiane des médianes »).*

5 Programmation dynamique

- Les ingrédients d'un algorithme de programmation dynamique pour un problème d'optimisation sont :
  1. une **formule récursive** pour la valeur optimale, en fonction de sous-problèmes possiblement non disjoints ;
  2. un **algorithme itératif** implantant la formule en commençant par les sous-problèmes les plus petits ;
  3. éventuellement un **algorithme de reconstruction** de la solution optimale *a posteriori*.
- Les algorithmes de programmation dynamique peuvent être **gourmands en mémoire**. En l'absence de reconstruction, on peut souvent **minimiser** l'espace mémoire utilisé.

5.1 Premier exemple : plus longue sous-suite croissante

- Une **plus longue sous-suite croissante (PLSSC)** d'un tableau T d'entiers est une suite la plus grande possible d'indices $0 \le i_1 < i_2 < \cdots < i_k \le n-1$ telle que $T_{[i_1]} \le T_{[i_2]} \le \cdots \le T_{[i_k]}$. Le problème de la plus longue sous-suite croissante consiste à calculer la longueur $\ell(T)$ d'une PLSSC de T, puis une PLSSC de longueur maximale.

Lemme 23 (Formule récursive pour la PLSSC)**.** *Si $\boldsymbol{\ell_i}$ est la longueur des* PLSSC *de T* ***finissant en case $\boldsymbol{T_{[i]}}$****, alors $\ell(T) = \max_i \ell_i$, $\ell_0 = 1$ et $\ell_i = 1 + \max\{\ell_j : j < i \text{ et } T_{[j]} \le T_{[i]}\}$ pour $1 \le i < n$.*

Algorithme : PLSSC(T)

```
L ← tableau de taille n, initialisé à 1
Prec ← tableau de taille n, initialisé à −1
i_M ← 0                          // L[i_M] contient max_i L[i]
pour i = 1 à n−1 :
    m ← 0      // m contient max{L[j] : j < i, T[j] ≤ T[i]}
    pour j = 0 à i−1 :
        si T[j] ≤ T[i] et L[j] > m : m ← L[j] ; Prec[i] ← j
    L[i] ← 1 + m
    si L[i] > L[i_M] : i_M ← i
renvoyer L[i_M], i_M et Prec
```

Algorithme : PLSSC_REC(T, i_M, Prec)

```
S ← tab. de raille n, initialisé à 0
i ← i_M
tant que i ≠ −1 :
    S[i] ← 1
    i ← Prec[i]
renvoyer S
```

Théorème 13. *L'algorithme* PLSSC *calcule $\ell(T)$ en temps $O(n^2)$. L'algorithme* PLSSC_REC *permet de calculer une* PLSSC *de T en temps $O(n)$.*

5.2 Deuxième exemple : le retour du choix de cours

- Le problème du **choix de cours valué** 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 arbitrairement mauvais pour ce problème !

Lemme 24 (Formule récursive pour le choix de cours valué)**.** *Soit max*ECTS(k) *le nombre maximal de crédits ECTS atteignable avec les cours $C_0, \ldots, C_k$ (max*ECTS$(-1) = 0$*), et pred$(k) = \max\{j : f_j \le d_k\}$ (avec $\max(\emptyset) = -1$). Alors max*ECTS$(0) = e_0$ *et pour $1 \le k < n$, max*ECTS$(k) = \max($*max*ECTS$(k-1), e_k +$ *max*ECTS$($*pred*$(k)))$.

Théorème 14. *La formule récursive fournit un algorithme en $O(n^2)$ pour le problème du choix de cours valué.*

5.3 Troisième exemple : la distance d'édition

- La **distance d'édition** entre deux mots A et B est le **nombre minimal de désaccords** dans un **alignement** de A et B. C'est aussi 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.

Lemme 25 (Formule récursive pour la distance d'édition)**.** *Soit* $\mathrm{edit}(i,j)$ *la distance d'édition entre* $A_{[0,i[}$ *et* $B_{[0,j[}$. *Alors* $\mathrm{edit}(i,0)=i$, $\mathrm{edit}(0,j)=j$ *et* $\mathrm{edit}(i,j)=\min(\mathrm{edit}(i-1,j)+1, \mathrm{edit}(i,j-1)+1, \mathrm{edit}(i-1,j-1)+\epsilon_{ij}$ *où* ϵ_{ij} *vaut* 1 *si* $A_{[i]}\neq B_{[j]}$ *et* 0 *sinon.*

Algorithme : DISTANCEEDITION(A,B)

$(m,n)\leftarrow$ tailles de A et B
$E\leftarrow$ tableau de dimensions $m+1$ par $n1$
pour $i=0$ *à* m **:** $E_{[i,0]}\leftarrow i$
pour $j=0$ *à* n **:** $E_{[0,j]}\leftarrow j$
pour $i=1$ *à* m **:**
  pour $j=1$ *à* n **:**
    $\epsilon\leftarrow 0$
    si $A_{[i-1]}\neq B_{[j-1]}$ **:** $\epsilon\leftarrow 1$
    $E_{[i,j]}\leftarrow\min(E_{[i-1,j]}+1,$
      $E_{[i,j-1]}+1,\ E_{[i-1,j-1]}+\epsilon)$
renvoyer $E_{[m,n]}$ `// E si reconstruction`

Algorithme : EDITIONMINMEMOIRE(A,B)

$(m,n)\leftarrow$ tailles de A et B
$P\leftarrow$ tableau de dimension $n+1$
$C\leftarrow$ tableau de dimension $n+1$
pour $j=0$ *à* n **:** $P_{[j]}\leftarrow j$
pour $i=1$ *à* m **:**
  $C_{[0]}\leftarrow i$
  pour $j=1$ *à* n **:**
    $\epsilon\leftarrow 0$ **si** $A_{[i-1]}=B_{[j-1]}$, 1 **sinon**
    $C_{[j]}\leftarrow\min(P_{[j]}+1, C_{[j-1]}+1, P_{[j-1]}+\epsilon)$
  pour $j=0$ *à* n **:** $P_{[j]}\leftarrow C_{[j]}$
renvoyer $C_{[n]}$

Algorithme : ALIGNEMENT(A,B,E)

$(i,j)\leftarrow(m,n)$
tant que $i>0$ *et* $j>0$ **:**
  si $E_{[i,j]}=E_{[i-1,j-1]}$ *et* $A_{[i-1]}=B_{[j-1]}$ **:** $(i,j)\leftarrow(i-1,j-1)$
  sinon si $E_{[i,j]}=E_{[i-1,j-1]}+1$ **:** $(i,j)\leftarrow(i-1,j-1)$
  sinon si $E_{[i,j]}=E_{[i-1,j]}+1$ **:** Insérer « _ » en $j^{\text{ème}}$ position dans B ; $i\leftarrow i-1$
  sinon si $E_{[i,j]}=E_{[i,j-1]}+1$ **:** Insérer « _ » en $i^{\text{ème}}$ position dans A ; $j\leftarrow j-1$
Insérer j symboles « _ » en tête de A ou i symboles « _ » en tête de B
renvoyer A *et* B

Théorème 15. DISTANCEEDITION *calcule la distance d'édition entre* A *et* B *de tailles respectives* m *et* n *en temps et espace* $O(mn)$. ALIGNEMENT *calcule ensuite l'alignement optimal de* A *et* B *en temps* $O(m+n)$. EDITIONMINMEMOIRE *calcule la distance d'édition entre* A *et* B *en temps* $O(mn)$ *et espace* $O(\min(m,n))$*, sans permettre la reconstruction.*

5.4 Quatrième exemple : plus courts chemins dans un graphe

- Un **graphe pondéré** $G=(S,A,p)$ est donné par un ensemble de sommets, un ensemble d'arêtes, et une fonction de poids sur les arêtes $p:A\to\mathbb{R}_+$. La **longueur pondérée** d'un chemin dans G est la somme des poids des arêtes qui le constitue.

Algorithme : DIJKSTRA(G, s, p)

```
F ← file de priorité vide
D ← tableau de n entiers, initialisé à +∞
P ← tableau de n entiers                // Prédecesseurs
pour chaque sommet u de G : AJOUTER(F, u, +∞)
CHANGERPRIORITÉ(F, s, 0) ; D[s] ← 0
tant que F est non vide :
    u ← EXTRAIREMIN(F)
    pour tout voisin v de u :
        si D[u] + p(u, v) < D[v] :
            D[v] ← D[u] + p(u, v) ; P[v] ← u
            CHANGERPRIORITÉ(F, v, D[v])
renvoyer D et P
```

Algorithme : DIJKSTRA_REC(G, P, u)

```
C ← {u}                        // Chemin
tant que u ≠ s :
    u ← P[u]
    Ajouter u à C
renvoyer C
```

Algorithme : DIJKSTRA_ARBRE(G, P)

```
T ← arbre vide
pour tout sommet u ≠ s :
    Ajouter l'arête P[u]u à T
renvoyer T
```

Théorème 16 (Algorithme de DIJKSTRA)**.** *Si G est connexe à n sommets et m arêtes,* DIJKSTRA(G, s, p) *calcule les longueurs pondérées des plus courts chemins de s à chaque sommet de G, en temps $O(m \log n)$ si G est représenté par listes d'adjacence et $O(n^2)$ si G est représenté par matrice d'adjacence.* DIJKSTRA_REC(G, P, u) *reconstruit un plus court chemin pondéré de s à u, et* DIJKSTRA_ARBRE *reconstruit l'**arbre des plus courts chemins** pondérés depuis s, en temps $O(n)$.*

5.5 Exemple spécial : le voyageur de commerce

- Le problème du **voyageur de commerce** cherche, étant donné un ensemble $S = \{s_0, \dots, s_{n-1}\}$ 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 26 (Formule récursive pour le voyageur de commerce)**.** *Si $U \subset S$ avec $s_0, s_j \in U$, on note $\mathbf{\Delta(U, s_j)}$ la longueur du plus court chemin de s_0 à s_j visitant chaque $s_i \in U$ une fois exactement. Alors $\ell_S = \min_j \Delta(\{s_0, \dots, s_{n-1}\}, s_j) + \delta_{j,0}$, $\Delta(\{s_0\}, s_0) = 0$, $\Delta(U, s_0) = +\infty$ 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 $0 < j < n$.*

Algorithme : TSP(S)

```
Δ ← tableau à deux dimensions, indexé par les sous-ensembles de S
      contenant {s0}, et par les entiers de 0 à n−1
Prec ← tableau de mêmes dimensions
Δ[{s0},0] = 0
pour s = 2 à n :
    pour tous les U ⊂ S de taille s tels que s0 ∈ U :
        Δ[U,s0] = +∞
        pour tout sj ∈ U, j ≠ 0 :
            Δ[U,sj] = min{Δ[U\{sj},si] + δij : si ∈ U, i ≠ j} Prec[U,sj] ←
            indice de ce minimum
renvoyer minj(Δ[{s0,...,sn},sj] + δj0, indice de ce minimum et Prec
```

Algorithme : TSP-REC$(S, \Delta, \text{Prec}, j)$

```
i0 ← 0
i1 ← j
U ← S
pour k = 2 à n−1 :
    ik ← Prec[U,i(k−1)]
    U ← S \ {s_i(k−1)}
renvoyer (i0, i1, ..., i(n−1), i0)
```

Théorème 17. *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.*