Récapitulatif des algorithmes du cours

(version du 6 mai 2024)

Ce document reprend les algorithmes vus au cours. Par convention, ils sont implémentés sous la forme de fonctions avec un nom utilisant CETTE POLICE. Si vous la voyez apparaître dans un algorithme, cela signifie donc qu'on fait appel à un autre algorithme déjà vu.

1 Parcours et applications

Algorithme 1 : PROFONDEURARBRE(A)

Entrées : A = arbre binaire enraciné.
Résultat : l'affichage des sommets de A suivant un parcours en profondeur à partir de la racine.

```
si A.racine() ≠ NIL alors
    afficher(A.racine());
    ProfondeurArbre(A.sous_arbre_gauche());
    ProfondeurArbre(A.sous_arbre_droit());
```

Algorithme 2 : PARCOURSLARGEURARBRE(A)

Entrées : A = arbre binaire enraciné.
Sortie : les sommets de l'arbre ordonnés selon un parcours en largeur à partir de la racine.

```
résultat ← Liste();
a_traiter ← File();
a_traiter.ajouter(A.racine());
tant que a_traiter.pas_vide() faire
    v ← a_traiter.extraire_premier();
    résultat.ajouter_en_fin(v);
    pour chaque s ∈ A.successeurs(v) faire
        a_traiter.ajouter(s);
renvoyer résultat;
```

Algorithme 3 : PROFONDEUR(G, départ)

Entrées : un graphe non orienté G, un sommet de départ.
Sortie : les sommets de G accessibles depuis le départ dans l'ordre où le parcours en profondeur les a découverts.

```
résultat ← Liste();
visités ← tableau(V(G), FAUX);
a_traiter ← Pile();
a_traiter.ajouter(départ);
tant que a_traiter.pas_vide() faire
    u ← a_traiter.extraire_premier();
    si ¬ visités[u] alors
        résultat.ajouter_en_fin(u);
        visités[u] ← VRAI;
        pour chaque v ∈ N(u) faire
            si ¬ visités[v] alors a_traiter.ajouter(v) ;
renvoyer résultat;
```

Algorithme 4 : Largeur(G, départ)

Entrées : un graphe non orienté G, un sommet de départ.
Sortie : les sommets de G accessibles depuis le départ dans l'ordre où le parcours en largeur les a découverts.

```
résultat ← Liste();
visités ← tableau(V(G), FAUX);
a_traiter ← File();
a_traiter.ajouter(départ);
tant que a_traiter.pas_vide() faire
    u ← a_traiter.extraire_premier();
    si ¬ visités[u] alors
        résultat.ajouter_en_fin(u);
        visités[u] ← VRAI;
        pour chaque v ∈ N(u) faire
            si ¬ visités[v] alors a_traiter.ajouter(v) ;
renvoyer résultat;
```

Algorithme 5 : ComposantesConnexes(G)

Entrées : un graphe non orienté G.
Sortie : les composantes connexes de G, identifiées par la liste de leurs sommets.

```
résultat ← Liste();
visités ← tableau(V(G), FAUX);
pour chaque v ∈ V(G) faire
    si ¬ visités[v] alors
        composante ← Profondeur(G, v);
        visités[u] ← VRAI ∀ u ∈ composante;
        résultat.ajouter_en_fin(composante) ;
renvoyer résultat;
```

Algorithme 6 : EstBiparti(G)

Entrées : un graphe connexe G non orienté.
Sortie : VRAI si G est biparti, FAUX sinon.

```
bipartition ← tableau(V(G), −1);                    // −1 = pas visité
départ ← sommet arbitraire de G;
a_traiter ← File();
a_traiter.ajouter(départ);
bipartition[départ] ← FAUX;
tant que a_traiter.pas_vide() faire
    u ← a_traiter.extraire_premier();
    pour chaque v ∈ N(u) faire
        si bipartition[v] = −1 alors
            a_traiter.ajouter(v);
            bipartition[v] ← ¬ bipartition[u];
        sinon si bipartition[v] = bipartition[u] alors renvoyer FAUX ;
renvoyer VRAI;
```

2 Graphes pondérés non orientés

Algorithme 7 : StockerAretesValides(G, u, S, hors_arbre)

Entrées : un graphe pondéré non orienté G, un sommet u de G, un tas d'arêtes S et un tableau booléen hors_arbre.

Résultat : les arêtes valides incidentes à u sont ajoutées à S.

pour chaque $v \in N(u)$ **faire**
  si *hors_arbre[v]* **alors** S.insérer((u, v, G.poids_arête(u, v))) ;

Algorithme 8 : ExtraireAreteSure(S, hors_arbre)

Entrées : un tas S d'arêtes et un tableau booléen hors_arbre.

Résultat : une arête sûre (ou factice s'il n'y en a pas) est extraite de S et renvoyée ; les arêtes invalides éventuellement rencontrées sont éliminées.

tant que S.*pas_vide()* **faire**
  $(u, v, p) \leftarrow S$.extraire_minimum();
  si *hors_arbre[u]* $\neq$ *hors_arbre[v]* **alors** **renvoyer** (u, v, p) ;
renvoyer (NIL, NIL, $+\infty$);

Algorithme 9 : Prim(G, départ)

Entrées : un graphe pondéré non orienté G, un sommet de départ.

Sortie : un ACPM pour la composante connexe de G contenant départ.

arbre $\leftarrow$ GraphePondéré();
arbre.ajouter_sommet(départ);
hors_arbre $\leftarrow$ tableau($V(G)$, VRAI);
hors_arbre[départ] $\leftarrow$ FAUX;
candidates $\leftarrow$ Tas() ;
StockerAretesValides(G, départ, candidates, hors_arbre) ;
tant que VRAI **faire**
  $(u, v, p) \leftarrow$ ExtraireAreteSure(candidates, hors_arbre) ;
  si u = NIL **alors** **renvoyer** arbre ;
  si *hors_arbre[v]* **alors** échanger u et v;
  arbre.ajouter_arête(u, v, p); `// rajoute aussi` u
  hors_arbre[u] $\leftarrow$ FAUX;
  StockerAretesValides(G, u, candidates, hors_arbre) ;

Algorithme 10 : Kruskal(G)

Entrées : un graphe pondéré non orienté G.

Sortie : une forêt couvrante de poids minimum pour G consistant en un arbre couvrant de poids minimum pour chaque composante connexe de G.

forêt $\leftarrow$ GraphePondéré($V(G)$);
classes $\leftarrow$ UnionFind($V(G)$);
pour chaque *(u, v, p)* $\in$ *tri_par_poids_croissant(E(G))* **faire**
  si *classes.find(u)* $\neq$ *classes.find(v)* **alors**
    forêt.ajouter_arête(u, v, p);
    classes.union(u, v);
renvoyer forêt;

3 Plus courts chemins ; graphes orientés

Algorithme 11 : ExtraireSommetLePlusProche(S, distances)

Entrées : S = ensemble de sommets, distances = la distance de chaque sommet de S.
Résultat : le sommet le plus proche est extrait de S et renvoyé.

```
sommet ← NIL;
distance_min ← +∞;
pour chaque candidat ∈ S faire
    si distances[candidat] < distance_min alors
        sommet ← candidat;
        distance_min ← distances[candidat];
si sommet ≠ NIL alors S ← S\ sommet;
renvoyer sommet;
```

Algorithme 12 : Dijkstra(G, source)

Entrées : G = graphe non orienté, simple et pondéré (poids ≥ 0), source = sommet de départ.
Sortie : la longueur d'un plus court chemin de la source à chacun des sommets de G ($+\infty$ pour les sommets non accessibles).

```
a_traiter ← V(G);
dist ← tableau(V(G), +∞);
dist[source] ← 0;
tant que a_traiter.pas_vide() faire
    u ← ExtraireSommetLePlusProche(a_traiter, dist);
    si u = NIL alors renvoyer dist;
    pour chaque v ∈ N(u) faire
        dist[v] ← min(dist[v], dist[u] + G.poids_arête(u, v));
renvoyer dist;
```

Algorithme 13 : ProfondeurOrienté(G, v, visités=NIL)

Entrées : G = graphe orienté, v = sommet de départ ; en option, un tableau visités indiquant les sommets déjà traités.
Sortie : les sommets de G accessibles au départ de v dans l'ordre où le parcours en profondeur les a découverts.

```
si visités = NIL alors visités ← tableau(V(G), FAUX);
résultat ← Liste();
résultat.ajouter_en_fin(v);
visités[v] ← VRAI;
pour chaque u ∈ N⁺(v) faire
    si ¬ visités[u] alors
        résultat ← résultat + ProfondeurOrienté(G, u, visités);
renvoyer résultat;
```

Algorithme 14 : CONTIENTCYCLEORIENTÉ(G, v, statuts)

Entrées : G = graphe orienté, v = sommet de départ, statuts = tableau indicé par $V(G)$.
Sortie : VRAI si un cycle de G est accessible à partir de v, FAUX sinon.

si *statuts[v]* = *"en cours"* **alors renvoyer** VRAI;
si *statuts[v]* = *"fini"* **alors renvoyer** FAUX;
statuts[v] ← "en cours";
pour chaque $u \in N^+(v)$ **faire**
  si CONTIENTCYCLEORIENTÉ(*G, u, statuts*) **alors renvoyer** VRAI;
statuts[v] ← "fini";
renvoyer FAUX;

Algorithme 15 : KAHN(G)

Entrées : un graphe orienté acyclique G.
Sortie : les sommets de G ordonnés selon un ordre topologique.

`/* stocker les degrés entrants et les sources */`
résultat ← Liste();
sources ← File();
degrés_entrants ← tableau($V(G)$, 0);
pour chaque $v \in V(G)$ **faire**
  degrés_entrants[v] ← $\deg^-(v)$;
  si *degrés_entrants[v] = 0* **alors** sources.ajouter(v);
`/* sortir les sources, les ajouter au résultat, et traiter les nouvelles sources */`
tant que *sources.pas_vide()* **faire**
  u ← sources.extraire_premier();
  résultat.ajouter_en_fin(u);
  pour chaque $v \in N^+(u)$ **faire**
    degrés_entrants[v] ← degrés_entrants[v] −1;
    si *degrés_entrants[v] = 0* **alors** sources.ajouter(v);
renvoyer résultat;

Algorithme 16 : FERMETURETRANSITIVE(G)

Entrées : un graphe orienté G.
Sortie : la fermeture transitive de G.

$F \leftarrow G$;
pour chaque $u \in V(F)$ **faire**
  pour chaque $v \in$ PROFONDEURORIENTÉ(*F, u*) **faire**
    si $u \neq v$ **alors** F.ajouter_arc(u, v);
renvoyer F;

4 Plus courts chemins ; graphes orientés (2)

Algorithme 17 : ProfondeurDates(G, v, dates, instant)

Entrées : G = graphe orienté, v = sommet de départ, un tableau de dates, et un instant.
Résultat : dates contient les dates de fin de visite des sommets de G accessibles depuis v dans l'ordre où le parcours en profondeur les a découverts.

dates[v] $\leftarrow$ 0 ; `// marquer le début de l'exploration`;
pour chaque $u \in N^+(v)$ **faire**
  | **si** *dates[u]* = NIL **alors** ProfondeurDates(G, u, dates, instant);
dates[v] $\leftarrow$ instant ; `// marquer la fin de l'exploration`;
instant $\leftarrow$ instant + 1;

Algorithme 18 : KosarajuSharir(G)

Entrées : G = graphe orienté.
Sortie : les composantes fortement connexes de G.

CFC $\leftarrow$ Liste();
a_traiter $\leftarrow$ ProfondeurPileFin(G) ; `// pile`
visités $\leftarrow$ tableau($V(G)$, FAUX);
tant que *a_traiter.pas_vide()* **faire**
  | $v \leftarrow$ a_traiter.extraire_premier();
  | **si** $\neg$ *visités[v]* **alors**
  | | CFC.ajouter_en_fin(ParcoursInversé(G, v, visités));
renvoyer CFC;

Algorithme 19 : BellmanFord(G, s)

Entrées : G = graphe pondéré orienté, s = sommet de départ.
Sortie : la longueur d'un plus court chemin de s à chacun des sommets de G ($+\infty$ pour les sommets non accessibles), ou NIL si G contient un cycle négatif.

distances $\leftarrow$ tableau($V(G)$, $+\infty$);
distances[s] $\leftarrow$ 0;
`// parcourir chaque arc` $|V| - 1$ `fois`
pour i *allant de* 1 *à* $|V(G)| - 1$ **faire**
  | **pour chaque** $(u, v, p) \in A(G)$ **faire**
  | | distances[v] $\leftarrow$ min(distances[v], distances[u] + p);
`// vérifier la présence d'un cycle négatif`
pour chaque $(u, v, p) \in A(G)$ **faire**
  | **si** *distances[v]* > *distances[u]* + p **alors renvoyer** NIL;
renvoyer distances;

Algorithme 20 : FloydWarshall(G)

Entrées : G = graphe orienté pondéré.
Sortie : les distances entre toute paire de sommets de G.

```
// initialiser la matrice de distances
dist ← matrice(|V(G)|, |V(G)|, +∞);
pour i allant de 0 à |V(G)| − 1 faire dist[i][i] ← 0;
pour chaque (u, v, p) ∈ A(G) faire dist[u][v] ← p;
// chercher les améliorations en passant par k = 0, 1, 2, ...
pour k allant de 0 à |V(G)| − 1 faire
    pour i allant de 0 à |V(G)| − 1 faire
        pour j allant de 0 à |V(G)| − 1 faire
            dist[i][j] ← min(dist[i][j], dist[i][k] + dist[k][j]);
renvoyer dist;
```

5 Flots et applications (1)

Algorithme 21 : FordFulkerson(G, s, t)

Entrées : G = réseau de flot, s = sa source, t = son puits.
Sortie : un flot maximum pour G.

```
flot ← tableau(A(G), 0);
G_f ← G;                                  // au départ, réseau = résiduel
chemin ← CheminAugmentant(G_f, s, t);
tant que chemin ≠ NIL faire
    AugmenterFlot(flot, chemin);
    MettreAJourRésiduel(G, G_f, A(chemin), flot);
    chemin ← CheminAugmentant(G_f, s, t);
renvoyer flot;
```

Algorithme 22 : CheminAugmentant(G, s, t)

Entrées : G = graphe orienté, s = source, t = puits.
Sortie : un chemin de s à t dans G, ou NIL s'il n'en existe pas.

```
déjà_visités ← tableau(V(G), FAUX);
a_traiter ← File();
a_traiter.ajouter(s);
parents ← tableau(V(G), NIL);
tant que a_traiter.pas_vide() faire
    u ← a_traiter.extraire_premier();
    si u = t alors arrêter;
    si ¬ déjà_visités[u] alors
        déjà_visités[u] ← VRAI;
        pour chaque v ∈ N⁺(u) faire
            si ¬ déjà_visités[v] alors
                a_traiter.ajouter(v);
                si parents[v] = NIL alors parents[v] ← u;
renvoyer ReconstruireChemin(G, s, t, parents);
```

Algorithme 23 : ReconstruireChemin(G, début, fin, parents)

Entrées : G = graphe orienté pondéré, deux sommets début et fin, et les parents des sommets du graphe.

Sortie : un chemin de début à fin dans G, ou NIL s'il n'en existe pas.

chemin ← GrapheOrientéPondéré();
v ← fin;
tant que $v \neq$ *début* **faire**
  si *parents[v]* = NIL **alors renvoyer** NIL;
  chemin.ajouter_arc(parents[v], v, G.poids_arc(parents[v], v));
  v ← parents[v];
renvoyer chemin;

Algorithme 24 : AugmenterFlot(f, P)

Entrées : f = flot, P = chemin orienté pondéré par des capacités.

Résultat : f augmente au maximum le long du chemin P.

capacité_min ← min $\{c \mid (u, v, c) \in A(P)\}$;
pour chaque $(u, v, c) \in A(P)$ **faire**
  si $(u, v) \in f$ **alors** $f(u, v) \leftarrow f(u, v)$ + capacité_min ;
  sinon $f(v, u) \leftarrow f(v, u)$ − capacité_min ;

Algorithme 25 : MettreAJourRésiduel(G, G_f, S, f)

Entrées : G = réseau de flot, G_f = résiduel correspondant, $S \subseteq A(G_f)$, et un flot f.

Résultat : met à jour les arcs de G_f concernés par S.

c_f ← tableau associatif;
pour chaque $(u, v) \in S$ **faire**
  `// calculer les capacités résiduelles`
  si $(u, v) \in A(G)$ **alors**
    $c_f(u, v) \leftarrow G$.poids_arc(u, v) $- f(u, v)$; $c_f(v, u) \leftarrow f(u, v)$;
  sinon
    $c_f(u, v) \leftarrow f(v, u)$; $c_f(v, u) \leftarrow G$.poids_arc(v, u) $- f(v, u)$;
  `// mettre à jour les arcs`
  si $c_f(u, v) > 0$ **alors** G_f.ajouter_arc(u, v, $c_f(u, v)$) ;
  sinon G_f.supprimer_arc(u, v);
  si $c_f(v, u) > 0$ **alors** G_f.ajouter_arc(v, u, $c_f(v, u)$) ;
  sinon G_f.supprimer_arc(v, u);

Algorithme 26 : EdmondsKarp(G, s, t)

Entrées : G = réseau de flot, s = sa source, t = son puits.

Sortie : un flot maximum pour G.

flot ← tableau($A(G)$, 0); `//` $O(|A|)$
$G_f \leftarrow G$; `//` $O(|V| + |A|)$
chemin ← CheminAugmentant(G_f, s, t); `//` $O(|V| + |A|)$
tant que *chemin* $\neq$ NIL **faire**
  AugmenterFlot(flot, chemin); `//` $O(|V|)$
  MettreAJourRésiduel(G, G_f, A(chemin), flot); `//` $O(|V|)$
  chemin ← CheminAugmentant(G_f, s, t); `//` $O(|V| + |A|)$
renvoyer flot;

6 Flots et applications (2)

Algorithme 27 : DINITZ(G, s, t)

Entrées : G = réseau de flot, s = sa source, t = son puits.
Sortie : un flot maximum pour G.

flot $\leftarrow$ tableau($A(G)$, 0);
$G_f \leftarrow G$;
$G_L \leftarrow$ DAGLARGEUR(G_f, s, t);
arcs $\leftarrow$ FLOTBLOQUANT(G, G_L, flot, s, t);
tant que *arcs* $\neq \emptyset$ **faire**
  METTREAJOURRESIDUEL(G, G_f, arcs, flot);
  $G_L \leftarrow$ DAGLARGEUR(G_f, s, t);
  arcs $\leftarrow$ FLOTBLOQUANT(G, G_L, flot, s, t);
renvoyer flot;

Algorithme 28 : FLOTBLOQUANT(G, G_L, f, s, t)

Entrées : G = réseau de flot, G_L = DAG pondéré, f = flot, s = source, t = puits.
Résultat : le flot f augmente au maximum le long de chaque chemin de G_L, qui est mis à jour au fur et à mesure ; renvoie l'ensemble des arcs qui ont subi un changement de flot.

arcs $\leftarrow \emptyset$;
chemin $\leftarrow$ CHEMINAUGMENTANT(G_L, s, t);
tant que *chemin* $\neq$ NIL **faire**
  AUGMENTERFLOT(f, chemin);
  METTREAJOURDAGLARGEUR(G, G_L, A(chemin), f);
  arcs $\leftarrow$ arcs $\cup$ A(chemin);
  chemin $\leftarrow$ CHEMINAUGMENTANT(G_L, s, t);
renvoyer arcs ;

Algorithme 29 : METTREAJOURDAGLARGEUR(G, G_L, arcs, f)

Entrées : G = réseau de flot, G_L = graphe orienté acyclique pondéré, un ensemble d'arcs, un flot f.
Résultat : le poids des arcs spécifiés de G_L est mis à jour.

pour chaque $(u, v, c) \in$ *arcs* **faire**
  si $(u, v) \in A(G)$ **alors**
    si G.*poids_arc*(u, v)$-f(u, v) > 0$ **alors**
      G_L.ajouter_arc(u, v, G.poids_arc(u, v) $-f(u, v)$)
    sinon G_L.supprimer_arc(u, v) ;
    si $f(u, v) > 0$ **alors** G_L.ajouter_arc(v, u, $f(u, v)$) ;
    sinon G_L.supprimer_arc(v, u) ;
  sinon
    si G.*poids_arc*(v, u) $-f(v, u) > 0$) **alors**
      G_L.ajouter_arc(v, u, G.poids_arc(v, u) $-f(v, u)$))
    sinon G_L.supprimer_arc(v, u);
    si $f(v, u) > 0$ **alors** G_L.ajouter_arc(u, v, $f(v, u)$) ;
    sinon G_L.supprimer_arc(u, v) ;

Algorithme 30 : CouplageBiparti(G)

Entrées : un graphe biparti connexe $G = (V_1 \cup V_2, E)$.
Sortie : un couplage maximum pour G.

$H \leftarrow$ GrapheOrientéPondéré();
pour chaque $\{u, v\} \in E(G)$ *avec* $u \in V_1$ *et* $v \in V_2$ **faire**
  H.ajouter_arcs($((s,\ u,\ 1),\ (u,\ v,\ 1),\ (v,\ t,\ 1))$) ;
flot $\leftarrow$ Dinitz($H,\ s,\ t$);
renvoyer $\{\{u, v\} \mid \{u, v\} \in E$ et $\text{flot}(u, v) = 1\}$;

7 Programmation dynamique

Algorithme 31 : Fibonacci récursif avec stockage

Algorithme récursif `FiboRec(`$val,\ n$`)`
  si $val[n] = -1$ **alors**
    val[n] $\leftarrow$ `FiboRec(`$val,\ n-1$`)` + `FiboRec(`$val,\ n-2$`)` ;
  renvoyer val[n]
Algorithme auxiliaire `FiboMieux(`n`)`
  val $\leftarrow$ Tableau($n+1,\ -1$) ; val[0] $\leftarrow$ 0 ; val[1] $\leftarrow$ 1 ;
  renvoyer `FiboRec(`$val,\ n$`)`

Algorithme 32 : FibonacciStockage(n)

Entrées : n = entier ≥ 0.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
F $\leftarrow$ Tableau($n+1,\ 0$);
F[1] $\leftarrow$ 1;
pour chaque $i \in (2, 3, \ldots, n)$ **faire** F[i] $\leftarrow$ F[$i-1$] + F[$i-2$];
renvoyer F[n];

Algorithme 33 : PlusCourtsCheminsDAG($G,\ s$)

Entrées : G = DAG pondéré, s = sommet de départ.
Sortie : la longueur d'un plus court chemin de s à chaque sommet de G ($+\infty$ pour les sommets inaccessibles).

dist $\leftarrow$ Tableau($V(G),\ +\infty$);
dist[s]$\leftarrow$ 0;
pour chaque $v \in$ Kahn*(G)* **faire**
  pour chaque $w \in N^+(v)$ **faire**
    dist[w] $\leftarrow$ min(dist[w], dist[v] + G.poids_arc($v,\ w$));
renvoyer dist;

Algorithme 34 : DécoupeNaïve(n, prix)

Entrées : n = longueur naturelle de tige, prix = tableau donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximal réalisable.

`// aucune découpe possible: renvoyer directement le prix`
si $n \leq 1$ **alors renvoyer** prix[n];
`// tester toutes les découpes possibles et garder la meilleure`
meilleur $\leftarrow$ prix[n];
pour chaque $i \in (1, 2, \ldots, n-1)$ **faire**
    meilleur $\leftarrow$ max(meilleur, prix[i] + DécoupeNaïve($n - i$, prix));
renvoyer meilleur;

Algorithme 35 : DécoupeOptimaleProgDyn(n, prix)

Entrées : n = longueur naturelle de tige, prix = tableau donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximal réalisable.

profits $\leftarrow$ prix;
pour chaque $k \in (2, 3, \ldots, n)$ **faire**
    pour chaque $i \in (1, 2, \ldots, k-1)$ **faire**
        profits[k] $\leftarrow$ max(profits[k], prix[i] + profits[$k - i$]);
renvoyer profits[n];

Algorithme 36 : DistanceEditionNaïve(S, k, T, n)

Entrées : deux chaînes S et T avec $k = |S|$ et $n = |T|$.
Sortie : la distance d'édition entre S et T.

si $k = 0$ **alors renvoyer** n;
si $n = 0$ **alors renvoyer** k;
option1 $\leftarrow$ 1 + DistanceEditionNaïve(S, $k-1$, T, n);
option2 $\leftarrow$ 1 + DistanceEditionNaïve(S, k, T, $n-1$);
option3 $\leftarrow$ $1_{S_k \neq T_n}$ + DistanceEditionNaïve(S, $k-1$, T, $n-1$);
renvoyer min(option1, option2, option3);

Algorithme 37 : DistanceEditionProgDyn(S, T)

Entrées : S = chaîne de longueur k, T = chaîne de longueur n.
Sortie : la distance d'édition entre S et T.

`/* initialiser la matrice de coûts avec les cas de base */`
$k \leftarrow |S|$;
$n \leftarrow |T|$;
coûts $\leftarrow$ Tableau($k+1, n+1, 0$);
pour chaque $i \in (0, 1, \ldots, k)$ **faire** coûts[i][0] $\leftarrow i$;
pour chaque $j \in (0, 1, \ldots, n)$ **faire** coûts[0][j] $\leftarrow j$;
`/* remplir la matrice de coûts à l'aide de la récurrence */`
pour chaque $i \in (1, 2, \ldots, k)$ **faire**
    pour chaque $j \in (1, 2, \ldots, n)$ **faire**
        option1 $\leftarrow$ 1 + coûts[$i-1$][j];
        option2 $\leftarrow$ 1 + coûts[i][$j-1$];
        option3 $\leftarrow$ $1_{S_i \neq T_j}$ + coûts[$i-1$][$j-1$];
        coûts[i][j] $\leftarrow$ min(option1, option2, option3);
renvoyer coûts[k][n];

Algorithme 38 : DécoupeOptimaleSolution(n, prix)

Entrées : n = longueur naturelle de tige, prix = tableau donnant pour chaque longueur possible le prix correspondant.

Sortie : les profits maximaux réalisables pour chaque longueur de tige, et la longueur du dernier morceau pour chaque découpe optimale.

profits $\leftarrow$ prix;
taille_dernière_pièce $\leftarrow$ tableau($n+1$, 0);
pour chaque k *allant de* 2 *à* n **faire**
    bénéfice $\leftarrow$ prix[k];
    taille_dernière_pièce[k] $\leftarrow k$;
    pour chaque i *allant de* 1 *à* $k-1$ **faire**
        si *prix[i] + profits[k − i] > bénéfice* **alors**
            bénéfice $\leftarrow$ prix[i] + profits[$k-i$];
            taille_dernière_pièce[k] $\leftarrow i$;
    profits[k] $\leftarrow$ bénéfice;
renvoyer (profits, taille_dernière_pièce);

Algorithme 39 : SubsetSumRec(M, S, μ)

Entrées : M = entier, S = ensemble d'entiers, $\mu = \min(S)$.

Sortie : une séquence d'éléments de S dont la somme vaut M, ou NIL si cela n'existe pas.

si $M = 0$ **et** $S \neq \emptyset$ **alors renvoyer** NIL;
si $M = 0$ **et** $S = \emptyset$ **alors renvoyer** 0;
pour chaque $v \in S$ **faire**
    sol_partielle $\leftarrow$ SubsetSumRec($M - v$, $S \setminus \{v\}$, μ);
    si *sol_partielle* $\neq$ NIL **alors renvoyer** v + sol_partielle;
renvoyer NIL;

Algorithme 40 : SubsetSumProgDyn(M, S)

Entrées : M = entier, S = ensemble d'entiers.

Sortie : une séquence d'éléments de S dont la somme vaut M, ou NIL si cela n'existe pas.

sol $\leftarrow$ Tableau($M+1$, NIL);
pour chaque $v \in (0, 1, \ldots, M)$ **faire**
    si $v \in S$ **alors** sol[v] $\leftarrow (v)$;
    sinon
        pour chaque $w \in S$ **faire**
            si *sol[v − w]* $\neq$ NIL **alors** sol[v] $\leftarrow (v)$ + sol[$v-w$];
renvoyer sol[M];

Algorithme 41 : VoyageurProgDyn(G)

Entrées : G = graphe complet pondéré.

Sortie : un cycle de poids minimum visitant tous les sommets de G exactement une fois.

OPT $\leftarrow$ Tableau();
pour chaque $i \in (2, 3, \ldots, |V|)$ **faire** OPT[v_i, v_i] $\leftarrow w(v_1, v_i)$;
pour chaque $j \in (2, 3, \ldots, |V|-1)$ **faire**
    pour chaque *sous-ensemble* $S \subseteq V$ *de taille* j **faire**
        OPT[S, v_j] $\leftarrow \min_{v_k \in S \setminus \{v_j\}} OPT[S \setminus \{v_j\}, v_k] + w(v_k, v_j)$;
renvoyer $\min_{i \in \{2,3,\ldots,|V|\}} \mathrm{OPT}[V \setminus \{v_1\}, v_i] + w(v_i, v_1)$;