PARTIE 1 : INTRODUCTION

CHAPITRE 1 : Rôle des algorithmes en informatique

CHAPITRE 2 : Premier pas

2.1 TRI PAR INSERTION

TRI-INSERTION(A)

```
pour j ← 2 à longueur[A]
    faire clé ← A[j]
        Insère A[j] dans la séquence triée A[1 . . j - 1].
        i ← j – 1
        tant que i > 0 et A[i] > clé
            faire A[i + 1] ← A[i]
                i ← i – 1
        A[i + 1] ← clé
```

2.3.1 METHODE DIVISER-POUR-REGNER

FUSION(A, p, q, r)

```
n1 ← q - p + 1
n2 ← r - q
créer tableaux L[1 . . n1 + 1] et R[1 . . n2 + 1]
pour i ← 1 à n1
    faire L[i] ← A[p + i - 1]
pour j ← 1 à n2
faire R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
pour k ← p à r
    faire si L[i] ≤R[j]
        alors A[k] ← L[i]
            i ← i + 1
        sinon A[k] ← R[j]
```

j ← j + 1

TRI-FUSION(A, p, r)

si p < r

alors q ← (p + r)/2

TRI-FUSION(A, p, q)

TRI-FUSION(A, q + 1, r)

FUSION(A, p, q, r)

CHAPITRE 3 : Croissance des fonctions

CHAPITRE 4 : Récurrences

CHAPITRE 5 : Analyse probabiliste et algorithmes randomisés

EMBAUCHE-SECRÉTAIRE(n)

meilleure ← 0 candidate 0 est une candidate fictive, moins qualifiée que quiconque

pour i ← 1 à n

faire interviewer candidate i

si candidate i supérieure à candidate meilleure

alors meilleure ← i

embaucher candidate i

EMBAUCHE-SECRÉTAIRE-RANDOMISÉ(n)

permuter aléatoirement la liste des candidates

meilleure ← 0 candidate 0 est une candidate fictive moins qualifiée que quiconque

```
pour i ← 1 à n
    faire interviewer candidate i
        si candidate i supérieure à candidate meilleure
            alors meilleure ← i
                embaucher candidate i
```

<u>PERMUTE-PAR-TRI(A)</u>

```
n ← longueur[A]
pour i ← 1 à n
    faire P[i] = RANDOM(1, n3)
trier A, avec P comme clés de tri
retourner A
```

<u>RANDOMISATION-DIRECTE(A)</u>

```
n ← longueur[A]
pour i ← 1 à n
    faire échanger A[i] ↔ A[RANDOM(i, n)]
```

MAXIMUM-EN-LIGNE(k, n)

```
meilleur score ← -∞
pour i ← 1 à k
    faire si score(i) > meilleur score
        alors meilleur score ← score(i)
pour i ← k + 1 à n
    faire si score(i) > meilleur score
        alors retourner i
retourner n
```

PARTIE 2 : TRI ET RANGS

CHAPITRE 6 : Tri par tas

ENTASSER-MAX(A, i)

l ← GAUCHE(i)

r ← DROITE(i)

si l ≤ taille[A] et A[l] > A[i]

alors max ← l

sinon max ← i

si r ≤ taille[A] et A[r] > A[max]

alors max ← r

si max ≠ i

alors échanger A[i] ↔ A[max]

ENTASSER-MAX(A, max)

CONSTRUIRE-TAS-MAX(A)

taille[A] ← longueur[A]

pour i ← longueur[A]/2 **jusqu'à** 1

faire ENTASSER-MAX(A, i)

TRI-PAR-TAS(A)

CONSTRUIRE-TAS-MAX(A)

pour i ← longueur[A] **jusqu'à** 2

faire échanger A[1] ↔ A[i]

taille[A] ← taille[A] – 1

ENTASSER-MAX(A, 1)

EXTRAIRE-MAX-TAS(A)

si taille[A] < 1

    alors erreur « limite inférieure dépassée »

max ← A[1]

A[1] ← A[taille[A]]

taille[A] ← taille[A] – 1

ENTASSER-MAX(A, 1)

retourner max

AUGMENTER-CLÉ-TAS(A, i, clé)

si clé < A[i]

    alors erreur « nouvelle clé plus petite que clé actuelle »

A[i] ← clé

tant que i > 1 et A[PARENT(i)] < A[i]

    faire permuter A[i] ↔ A[PARENT(i)]

        i ← PARENT(i)

CHAPITRE 7 : Tri rapide

TRI-RAPIDE(A, p, r)

si p < r

    alors q ← PARTITION(A, p, r)

        TRI-RAPIDE(A, p, q - 1)

        TRI-RAPIDE(A, q + 1, r)

PARTITION(A, p, r)

x ← A[r]

i ← p – 1

pour j ← p **à** r – 1

    faire si A[j] x

        alors i ← i + 1

            permuter A[i] ↔ A[j]

permuter A[i + 1] ↔ A[r]

retourner i + 1

PARTITION-RANDOMISE(A, p, r)

i ← RANDOM(p, r)

échanger A[r] ↔ A[i]

retourner PARTITION(A, p, r)

TRI-RAPIDE-RANDOMISE(A, p, r)

si p < r

    alors q ← PARTITION-RANDOMISE(A, p, r)

        TRI-RAPIDE-RANDOMISE(A, p, q - 1)

        TRI-RAPIDE-RANDOMISE(A, q + 1, r)

CHAPITRE 8 : Tri en temps linéaire

TRI-DÉNOMBREMENT(A, B, k)

pour i ← 0 **à** k

  faire C[i] ← 0

pour j ← 1 **à** longueur[A]

  faire C[A[j]] ← C[A[j]] + 1

C[i] contient maintenant le nombre d'éléments égaux à i.

pour i ← 1 **à** k

  faire C[i] ← C[i] + C[i - 1]

C[i] contient maintenant le nombre d'éléments inférieurs ou égaux à i.

pour j ← longueur[A] **jusqu'à** 1

  faire B[C[A[j]]] ← A[j]

    C[A[j]] ← C[A[j]] – 1

TRI-PAQUETS(A)

n ← longueur[A]

pour i ← 1 **à** n

faire insérer A[i] dans liste B[nA[i]]

pour i ← 0 **à** n – 1

faire trier liste B[i] via tri par insertion

concaténer les listes B[0], B[1], . . . , B[n - 1] dans l'ordre

CHAPITRE 9 : Médians et rangs

MINIMUM(A)

min ← A[1]

pour i ← 2 **à** longueur[A]

  faire si min > A[i]

    alors min ← A[i]

retourner min

SÉLECTION-RANDOMISÉE(A, p, r, i)

si p = r

    alors retourner A[p]

q ← PARTITION-RANDOMISÉE(A, p, r)

k ← q - p + 1

si i = k la valeur du pivot est la réponse

    alors retourner A[q]

sinon si i < k

    alors retourner SÉLECTION-RANDOMISÉE(A, p, q - 1, i)

sinon retourner SÉLECTION-RANDOMISÉE(A, q + 1, r, i - k)

PARTIE 3 STRUCTURES DE DONNÉES

CHAPITRE 10 : Structures de données élémentaires

PILE-VIDE(P)

si sommet[P] = 0

    alors retourner VRAI

    sinon retourner FAUX

EMPILER(P, x)

sommet[P] ← sommet[P] + 1

P[sommet[P]] ← x

DÉPILER(P)

```
si PILE-VIDE(P)
      alors erreur « débordement négatif »
      sinon sommet[P] ← sommet[P] – 1
            retourner P[sommet[P] + 1]
```

ENFILER(F, x)

```
F[queue[F]] ← x
si queue[F] = longueur[F]
      alors queue[F] ← 1
      sinon queue[F] ← queue[F] + 1
```

DÉFILER(F)

```
x ← F[tête[F]]
si tête[F] = longueur[F]
      alors tête[F] ← 1
      sinon tête[F] ← tête[F] + 1
retourner x
```

RECHERCHE-LISTE(L, k)

```
x ← tête[L]
tant que x ≠ NIL et clé[x] ≠ k
      faire x ← succ[x]
retourner x
```

<u>INSÉRER-LISTE(L, x)</u>

succ[x] ← tête[L]

si tête[L] ≠ NIL

    alors préd[tête[L]] ← x

tête[L] ← x

préd[x] ← NIL

<u>SUPPRIMER-LISTE(L, x)</u>

si préd[x] ≠ NIL

    alors succ[préd[x]] ← succ[x]

    sinon tête[L] ← succ[x]

si succ[x] ≠ NIL

    alors préd[succ[x]] ← préd[x]

<u>SUPPRIMER-LISTE(L, x)</u>

succ[préd[x]] ← succ[x]
préd[succ[x]] ← préd[x]

<u>RECHERCHE-LISTE(L, k)</u>

x ← succ[nil[L]]

tant que x ≠ nil[L] et clé[x] ≠ k

    faire x ← succ[x]

retourner x

<u>INSÉRER-LISTE(L, x)</u>

succ[x] ← succ[nil[L]]
préd[succ[nil[L]]] ← x
succ[nil[L]] ← x
préd[x] ← nil[L]

<u>ALLOUER-OBJET()</u>

```
si libre = NIL
      alors erreur « plus assez d'espace disponible »
      sinon x ← libre
            libre ← succ[x]
            retourner x
```

<u>CHAPITRE 11 :</u> Tables de hachage

<u>INSÉRER-HACHAGE(T, k)</u>

```
i ← 0
répeter j ← h(k, i)
      si T[j] = NIL
            alors T[j] ← k
            retourner j
      sinon i ← i + 1
      jusqu'à i = m
erreur « débordement de la table de hachage »
```

<u>RECHERCHER-HACHAGE(T, k)</u>

```
i ← 0
répéter j ← h(k, i)
      si T[j] = k
            alors retourner j
            i ← i + 1
      jusqu'à T[j] = NIL ou i = m
retourner NIL
```

CHAPITRE 12 : Arbres binaires de recherche

PARCOURS-INFIXE(x)

```
si x ≠ NIL
    alors PARCOURS-INFIXE(gauche[x])
          afficher clé[x]
          PARCOURS-INFIXE(droite[x])
```

ARBRE-RECHERCHER(x, k)

```
si x = NIL ou k = clé[x]
    alors retourner x
si k < clé[x]
    alors retourner ARBRE-RECHERCHER(gauche[x], k)
    sinon retourner ARBRE-RECHERCHER(droite[x], k)
```

ARBRE-RECHERCHER-ITÉRATIF(x, k)

```
tant que x ≠ NIL et k ≠ clé[x]
    faire si k < clé[x]
            alors x ← gauche[x]
            sinon x ← droite[x]
retourner x
```

ARBRE-MAXIMUM(x)

```
tant que droite[x] ≠ NIL
    faire x ← droite[x]
retourner x
```

ARBRE-SUCCESSEUR(x)

```
si droite[x] ≠ NIL
        alors retourner ARBRE-MINIMUM(droite[x])
y ← p[x]
tant que y ≠ NIL et x = droite[y]
        faire x ← y
              y ← p[y]
retourner y
```

ARBRE-INSÉRER(T, z)

```
y ← NIL
x ← racine[T]
tant que x ≠ NIL
        faire y ← x
              si clé[z] < clé[x]
                    alors x ← gauche[x]
                    sinon x ← droite[x]
p[z] ← y
si y = NIL
        alors racine[T] ← z            arbre T était vide
        sinon si clé[z] < clé[y]
              alors gauche[y] ← z
              sinon droite[y] ← z
```

ARBRE-SUPPRIMER(T, z)

si gauche[z] = NIL ou droite[z] = NIL

    alors y ← z

    sinon y ← ARBRE-SUCCESSEUR(z)

si gauche[y] ≠ NIL

    alors x ← gauche[y]

    sinon x ← droite[y]

si x ≠ NIL

    alors p[x] ← p[y]

si p[y] = NIL

    alors racine[T] ← x

    sinon si y = gauche[p[y]]

        alors gauche[p[y]] ← x

        sinon droite[p[y]] ← x

si y ≠ z

    alors clé[z] ← clé[y]

        copier données satellites de y dans z

retourner y

CHAPITRE 13 : Arbres rouge-noir

ROTATION-GAUCHE(T, x)

y ← droite[x]   initialise y.

droite[x] ← gauche[y]   sous-arbre gauche de y devient sous-arbre droit de x.

si gauche[y] ≠ nil[T]

    alors p[gauche[y]] ← x

        p[y] ← p[x] relie parent de x à y.

```
        si p[x] = nil[T]
            alors racine[T] ← y
            sinon si x = gauche[p[x]]
                alors gauche[p[x]] ← y
                sinon droite[p[x]] ← y
        gauche[y] ← x place x à gauche de y.
        p[x] ← y
```

<u>RN-INSÉRER(T, z)</u>

```
y ← nil[T]
x ← racine[T]
tant que x ≠ nil[T]
    faire y ← x
        si clé[z] < clé[x]
            alors x ← gauche[x]
            sinon x ← droite[x]
p[z] ← y
si y = nil[T]
    alors racine[T] ← z
    sinon si clé[z] < clé[y]
        alors gauche[y] ← z
        sinon droite[y] ← z
gauche[z] ← nil[T]
droite[z] ← nil[T]
couleur[z] ← ROUGE
RN-INSÉRER-CORRECTION(T, z)
```

RN-INSÉRER-CORRECTION(T, z)

```
tant que couleur[p[z]] = ROUGE
        faire si p[z] = gauche[p[p[z]]]
                 alors y ← droite[p[p[z]]]
                         si couleur[y] = ROUGE
                                 alors couleur[p[z]] ← NOIR                    Cas 1
                                         couleur[y] ← NOIR                      Cas 1
                                         couleur[p[p[z]]] ← ROUGE               Cas 1
                                         z ← p[p[z]]                            Cas 1
                                 sinon si z = droite[p[z]]
                                           alors z ← p[z]                       Cas 2
                                                   ROTATION-GAUCHE(T, z)        Cas 2
                                         couleur[p[z]] ← NOIR                   Cas 3
                                         couleur[p[p[z]]] ← ROUGE               Cas 3
                                         ROTATION-DROITE(T, p[p[z]])            Cas 3
                 sinon (idem clause alors avec permutation de « droite » et « gauche »)
couleur[racine[T]] ← NOIR
```

RN-SUPPRIMER(T, z)

```
si gauche[z] = nil[T] ou droite[z] = nil[T]
        alors y ← z
        sinon y ← ARBRE-SUCCESSEUR(z)
si gauche[y] ≠ nil[T]
        alors x ← gauche[y]
        sinon x ← droite[y]
p[x] ← p[y]
si p[y] = nil[T]
        alors racine[T] ← x
        sinon si y = gauche[p[y]]
                alors gauche[p[y]] ← x
                sinon droite[p[y]] ← x
si y ≠ z
        alors clé[z] ← clé[y]
                copier données satellite de y dans z
si couleur[y] = NOIR
        alors RN-SUPPRIMER-CORRECTION(T, x)
retourner y
```

RN-SUPPRIMER-CORRECTION(T, x)

```
tant que x ≠  racine[T] et couleur[x] = NOIR
    faire si x = gauche[p[x]]
        alors w ← droite[p[x]]
            si couleur[w] = ROUGE
                alors couleur[w] ← NOIR                                   Cas 1
                    couleur[p[x]] ← ROUGE                                 Cas 1
                    ROTATION-GAUCHE(T, p[x])                              Cas 1
                    w ← droite[p[x]]                                      Cas 1
            si couleur[gauche[w]] = NOIR et couleur[droite[w]] = NOIR
                alors couleur[w] ← ROUGE                                  Cas 2
                x ← p[x]                                                  Cas 2
                sinon si couleur[droite[w]] = NOI
                    alors couleur[gauche[w]] ← NOIR                       Cas 3
                    couleur[w] ← ROUGE                                    Cas 3
                    ROTATION-DROITE(T, w)                                 Cas 3
                    w ← droite[p[x]]                                      Cas 3
                 couleur[w] ← couleur[p[x]]                               Cas 4
                 couleur[p[x]] ← NOIR                                     Cas 4
                 couleur[droite[w]] ← NOIR                                Cas 4
                 ROTATION-GAUCHE(T, p[x])                                 Cas 4
                 x ← racine[T]                                            Cas 4
        sinon (idem clause alors avec « droite » et « gauche » échangés)
couleur[x] ← NOIR
```

CHAPITRE 14 : Extension d'une structure de données

RÉCUPÉRER-RANG(x, i)

```
r ← taille[gauche[x]]+1
si i = r
    alors retourner x
sinon si i < r
    alors retourner RÉCUPÉRER-RANG(gauche[x], i)
sinon retourner RÉCUPÉRER-RANG(droite[x], i - r)
```

DÉTERMINER-RANG(T, x)

```
r ← taille[gauche[x]] + 1
y ← x
tant que y fi racine[T]
        faire si y = droite[p[y]]
                    alors r ← r + taille[gauche[p[y]]] + 1
                  y ← p[y]
retourner r
```

PARTIE 4 : TECHNIQUES AVANCÉES DE CONCEPTION ET D'ANALYSE

CHAPITRE 15 : Programmation dynamique

PLUS-RAPIDE-CHEMIN(a, t, e, x, n)

```
f1[1] ← e1 + a1,1
f2[1] ← e2 + a2,1
pour j ← 2 à n
        faire si f1[j - 1] + a1,j  f2[j - 1] + t2,j-1 + a1,j
                  alors f1[j] ← f1[j - 1] + a1,j
                         l1[j] ← 1
                  sinon f1[j] ← f2[j - 1] + t2,j-1 + a1,j
                         l1[j] ← 2
               si f2[j - 1] + a2,j  f1[j - 1] + t1,j-1 + a2,j
                  alors f2[j] ← f2[j - 1] + a2,j
                         l2[j] ← 2
                  sinon f2[j] ← f1[j - 1] + t1,j-1 + a2,j
                         l2[j] ← 1
si f1[n] + x1  f2[n] + x2
        alors f ∗ = f1[n] + x1
               l∗ = 1
        sinon f ∗ = f2[n] + x2
               l∗ = 2
```

AFFICHER-POSTES(l, n)

```
i ← l*
afficher « chaîne » i « , poste » n
pour j ← n jusqu'à 2
    faire i ← li[j]
        afficher « chaîne » i « ,poste » j - 1
```

MULTIPLIER-MATRICES(A, B)

```
si colonnes[A] ≠ lignes[B]
    alors erreur « dimensions incompatibles »
    sinon pour i ← 1 à lignes[A]
        faire pour j ← 1 à colonnes[B]
            faire C[i, j] ← 0
                pour k ← 1 à colonnes[A]
                    faire C[i, j] ← C[i, j] + A[i, k]·B[k, j]
retourner C
```

ORDRE-CHAÎNE-MATRICES(p)

```
n ← longueur[p] – 1
pour i ← 1 à n
    faire m[i, i] ← 0
pour l ← 2 à n              l est la longueur de la chaîne.
    faire pour i ← 1 à n - l + 1
        faire j ← i + l – 1
            m[i, j] ← ∞
            pour k ← i à j – 1
                faire q ← m[i, k] + m[k + 1, j] + pi-1pkpj
                    si q < m[i, j]
                        alors m[i, j] ← q
                        s[i, j] ← k
retourner m et s
```

AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, j)

```
si i = j
        alors afficher« A »i
        sinon afficher« ( «
                AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, s[i, j])
                AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, s[i, j] + 1, j)
                print » ) »
```

CHAÎNE-MATRICES-RÉCURSIF(p, i, j)

```
si i = j
        alors retourner 0
m[i, j] ← ∞
pour k ← i à j – 1
        faire q ← CHAÎNE-MATRICES-RÉCURSIF(p, i, k)
                + CHAÎNE-MATRICES-RÉCURSIF(p, k + 1, j)
                + pi-1pkpj
          si q < m[i, j]
                alors m[i, j] ← q
retourner m[i, j]
```

MÉMORISATION-CHAÎNE-MATRICES(p)

```
n ← longueur[p] – 1
pour i ← 1 à n
        faire pour j ← i à n
                faire m[i, j] ← ∞
retourner RÉCUPÉRER-CHAÎNE(p, 1, n)
```

RÉCUPÉRER-CHAÎNE(p, i, j)

```
si m[i, j] < ∞
        alors retourner m[i, j]
si i = j
        alors m[i, j] ← 0
        sinon pour k ← i à j – 1
                faire q ← RÉCUPÉRER-CHAÎNE(p, i, k)
                        + RÉCUPÉRER-CHAÎNE(p, k + 1, j) + pi-1pkpj
```

```
            si q < m[i, j]
                  alors m[i, j] ← q
retourner m[i, j]
```

<u>LONGUEUR-PLSC(X, Y)</u>

```
m ← longueur[X]
n ← longueur[Y]
pour i ← 1 à m
      faire c[i, 0] ← 0
pour j ← 0 à n
      faire c[0, j] ← 0
pour i ← 1 à m
      faire pour j ← 1 à n
            faire si xi = yj
                  alors c[i, j] ← c[i - 1, j - 1] + 1
                        b[i, j] ← « ? »
                  sinon si c[i - 1, j]  c[i, j - 1]
                        alors c[i, j] ← c[i - 1, j]
                              b[i, j] ← « ↑ »
                        sinon c[i, j] ← c[i, j - 1]
                              b[i, j] ← « ← »
retourner c et b
```

<u>IMPRIMER-PLSC(b, X, i, j)</u>

```
si i = 0 ou j = 0
      alors retourner
si b[i, j] = « ? »
      alors IMPRIMER-PLSC(b, X, i - 1, j - 1)
            imprimer xi
sinon si b[i, j] = « ↑ »
      alors IMPRIMER-PLSC(b, X, i - 1, j)
```

sinon IMPRIMER-PLSC(b, X, i, j - 1)

<u>ABR-OPTIMAL(p, q, n)</u>

pour i ← 1 **à** n + 1

    faire e[i, i - 1] ← qi-1

        w[i, i - 1] ← qi-1

pour l ← 1 **à** n

    faire pour i ← 1 **à** n - l + 1

        faire j ← i + l – 1

            e[i, j] ← ∞

            w[i, j] ← w[i, j - 1] + pj + qj

            pour r ← i **à** j

                faire t ← e[i, r - 1] + e[r + 1, j] + w[i, j]

                    si t < e[i, j]

                        alors e[i, j] ← t

                            racine[i, j] ← r

retourner e et racine

<u>CHAPITRE 16 :</u> Algorithmes gloutons

<u>CHOIX-D’ACTIVITÉS-RÉCURSIF(s, f , i, n)</u>

m ← i + 1

tant que m≤ n et sm < fi   trouver première activité de Si,n+1.

    faire m ← m + 1

si m ≤n

    alors retourner {am} ∪ CHOIX-D’ACTIVITÉS-RÉCURSIF(s, f , m, n)

    sinon retourner Ø

<u>CHOIX-D'ACTIVITÉS-GLOUTON(s, f)</u>

n ← longueur[s]

A ← {a1}

i ← 1

pour m ← 2 **à** n

    faire si sm ≥ fi

        alors A ← A ∪ {am}

            i ← m

retourner A

<u>HUFFMAN(C)</u>

n ← |C|

Q ← C

pour i ← 1 **à** n – 1

    faire allouer un nouveau nœud z

        gauche[z] ← x ← EXTRAIRE-MIN(Q)

        droite[z] ← y ← EXTRAIRE-MIN(Q)

        f [z] ← f [x] + f [y]

        INSÉRER(Q, z)

retourner EXTRAIRE-MIN(Q)   Retourner la racine de l'arborescence.

<u>GLOUTON(M, w)</u>

F ← Ø

trier E[M] par ordre de poids décroissant w

pour chaque x ∈ S[M], pris par ordre de poids décroissant w(x)

    faire si F ∪ {x} ∈ I[M]

        alors F ← F ∪ {x}

retourner F

CHAPITRE 17 : Analyse amortie

MULTIDÉP(S, k)

```
tant que pas PILE-VIDE(S) et k ≠ 0
        faire DÉPILER(S)
              k ← k – 1
```

INCRÉMENTER(A)

```
i ← 0
tant que i < longueur[A] et A[i] = 1
        faire A[i] ← 0
              i ← i + 1
si i < longueur[A]
        alors A[i] ← 1
```

INSÉRER-TABLE(T, x)

```
si taille[T] = 0
        alors allouer table[T] avec 1 alvéole
              taille[T] ← 1
si num[T] = taille[T]
        alors allouer 2· taille[T] alvéoles pour nouvelle-table
              insérer tous les éléments de table[T] dans nouvelle-table
              libérer table[T]
              table[T] ← nouvelle-table
              taille[T] ← 2· taille[T]
insérer x dans table[T]
num[T] ← num[T] + 1
```

PARTIE 5 : STRUCTURES DE DONNÉES AVANCÉES

CHAPITRE 18 : B-arbres

RECHERCHER-B-ARBRE(x, k)

```
i ← 1
tant que i ≤ n[x] et k > cléi[x]
        faire i ← i + 1
si i ≤ n[x] et k = cléi[x]
        alors retourner (x, i)
si feuille[x]
        alors retourner NIL
        sinon LIRE-DISQUE(ci[x])
                retourner RECHERCHER-B-ARBRE(ci[x], k)
```

CRÉER-B-ARBRE(T)

```
x ← ALLOUER-NŒUD()
feuille[x] ← VRAI
n[x] ← 0
ÉCRIRE-DISQUE(x)
racine[T] ← x
```

PARTAGER-ENFANT-B-ARBRE(x, i, y)

```
z ← ALLOUER-NŒUD()
feuille[z] ← feuille[y]
n[z] ← t – 1
pour j ← 1 à t – 1
        faire cléj[z] ← cléj+t[y]
si non feuille[y]
        alors pour j ← 1 à t
```

```
            faire cj[z] ← cj+t[y]
n[y] ← t – 1
pour j ← n[x] + 1 jusqu'à i + 1
        faire cj+1[x] ← cj[x]
            ci+1[x] ← z
pour j ← n[x] jusqu'à i
        faire cléj+1[x] ← cléj[x]
cléi[x] ← clét[y]
n[x] ← n[x] + 1
ÉCRIRE-DISQUE(y)
ÉCRIRE-DISQUE(z)
ÉCRIRE-DISQUE(x)
```

<u>INSÉRER-B-ARBRE(T, k)</u>

```
r ← racine[T]
si n[r] = 2t – 1
        alors s ← ALLOUER-NŒUD()
            racine[T] ← s
            feuille[s] ← FAUX
            n[s] ← 0
            c1[s] ← r
            PARTAGER-ENFANT-B-ARBRE(s, 1, r)
            INSÉRER-B-ARBRE-INCOMPLET(s, k)
        sinon INSÉRER-B-ARBRE-INCOMPLET(r, k)
```

INSÉRER-B-ARBRE-INCOMPLET(x, k)

```
i ← n[x]
si feuille[x]
    alors tant que i ≥ 1 et k < cléi[x]
        faire cléi+1[x] ← cléi[x]
            i ← i – 1
      cléi+1[x] ← k
      n[x] ← n[x] + 1
      ÉCRIRE-DISQUE(x)
    sinon tant que i ≥ 1 et k < cléi[x]
        faire i ← i – 1
      i ← i + 1
      LIRE-DISQUE(ci[x])
      si n[ci[x]] = 2t – 1
        alors PARTAGER-ENFANT-B-ARBRE(x, i, ci[x])
            si k > cléi[x]
                alors i ← i + 1
      INSÉRER-B-ARBRE-INCOMPLET(ci[x], k)
```

CHAPITRE 19 : Tas binomiaux

MINIMUM-TAS-BINOMIAL(T)

```
y ← NIL
x ← tête[T]
min ← ∞
tant que x ≠ NIL
    faire si clé[x] < min
        alors min ← clé[x]
            y ← x
      x ← frère[x]
```

retourner y

LIEN-BINOMIAL(y, z)

```
p[y] ← z
frère[y] ← enfant[z]
enfant[z] ← y
degré[z] ← degré[z] + 1
```

FUSIONNER-TAS-BINOMIAUX(T1, T2)

```
T ← CRÉER-TAS-BINOMIAL()
tête[T] ← FUSIONNER-TAS-BINOMIAUX(T1, T2)
libère objets T1 et T2, mais pas les listes vers lesquelles ils pointent
si tête[T] = NIL
        alors retourner T
avant-x ← NIL
x ← tête[T]
après-x ← frère[x]
tant que après-x ≠ NIL
        faire si (degré[x] ≠ degré[après-x]) ou
                        (frère[après-x] ≠ NIL et degré[frère[après-x]] = degré[x])
                  alors avant-x ← x                                         Cas 1 et 2
                        x ← après-x                                         Cas 1 et 2
                  sinon si clé[x] ≤ clé[après-x]
                        alors frère[x] ← frère[après-x]                     Cas 3
                              LIEN-BINOMIAL(après-x, x)                     Cas 3
                        sinon si avant-x = NIL                              Cas 4
                              alors tête[T] ← après-x                       Cas 4
                              sinon frère[avant-x] ← après-x                Cas 4
                            LIEN-BINOMIAL(x, après-x)                       Cas 4
                            x ← après-x                                     Cas 4
                 après-x ← frère[x]
retourner T
```

TAS-BINOMIAL-INSÉRER(T, x)

```
T ← CRÉER-TAS-BINOMIAL()
p[x] ← NIL
enfant[x] ← NIL
frère[x] ← NIL
degré[x] ← 0
tête[T] ← x
T ← UNION-TAS-BINOMIAUX(T, T)
```

TAS-BINOMIAL-EXTRAIRE-MIN(T)

trouver la racine x de clé minimale dans la liste des racines de T, et supprimer x de la liste

T ← CRÉER-TAS-BINOMIAL()

inverser l'ordre de la liste chaînée des enfant de x, et faire pointer tête[T] sur la tête de la liste résultante

T ← UNION-TAS-BINOMIAUX(T, T)

retourner x

TAS-BINOMIAL-DIMINUER-CLÉ(T, x, k)

si k > clé[x]

  alors erreur « La nouvelle clé est plus grande que la clé courante »

clé[x] ← k

y ← x

z ← p[y]

tant que z ≠ NIL et clé[y] < clé[z]

  faire permuter clé[y] ↔ clé[z]

    Si y et z ont des champs satellites, on les permute aussi.

    y ← z

    z ← p[y]

CHAPITRE 20 : Tas de Fibonacci

INSÉRER-TAS-FIB(T, x)

degré[x] ← 0

p[x] ← NIL

enfant[x] ← NIL

gauche[x] ← x

droite[x] ← x

marqué[x] ← FAUX

concaténer liste de racines contenant x et liste de racines T

si min[T] = NIL ou clé[x] < clé[min[T]]

  alors min[T] ← x

n[T] ← n[T] + 1

UNION-TAS-FIB(T1, T2)

```
T ← CRÉER-TAS-FIB()
min[T] ← min[T1]
concaténer liste de racines de T2 à celle de T
si (min[T1] = NIL) ou (min[T2] ≠ NIL et clé[min[T2]] < clé[min[T1]])
        alors min[T] ← min[T2]
n[T] ← n[T1] + n[T2]
libérer objets T1 et T2
retourner T
```

EXTRAIRE-MIN-TAS-FIB(T)

```
z ← min[T]
si z ≠ NIL
        alors pour chaque enfant x de z
                faire ajouter x à la liste de racines de T
                        p[x] ← NIL
            supprimer z de la liste de racines de T
            si z = droite[z]
                alors min[T] ← NIL
                sinon min[T] ← droite[z]
                        CONSOLIDER(T)
            n[T] ← n[T] – 1
retourner z
```

```
CONSOLIDER(T)
 pour i ← 0 à D(n[T])
     faire A[i] ← NIL
 pour chaque nœud w de la liste des racines de T
     faire x ← w
         d ← degré[x
         tant que A[d] ≠ NIL
             faire y ← A[d]
                 si clé[x] > clé[y]
                     alors permuter x ↔ y
                 RELIER-TAS-FIB(T, y, x)
                 A[d] ← NIL
                 d ← d + 1
         A[d] ← x
 min[T] ← NIL
 pour i ← 0 à D(n[T])
     faire si A[i] ≠ NIL
         alors ajouter A[i] à la liste des racines de T
             si min[T] = NIL ou clé[A[i]] < clé[min[T]]
                 alors min[T] ← A[i]
```

<u>RELIER-TAS-FIB(T, y, x)</u>

```
 supprimer y de la liste des racines de T
 faire de y un enfant de x, incrémenter degré[x]
 marqué[y] ← FAUX
```

DIMINUER-CLÉ-TAS-FIB(T, x, k)

si k > clé[x]

  alors erreur « nouvelle clé plus grande que clé courante »

clé[x] ← k

y ← p[x]

si y ≠ NIL et clé[x] < clé[y]

  alors COUPER(T, x, y)

    COUPE-EN-CASCADE(T, y)

si clé[x] < clé[min[T]]

  alors min[T] ← x

COUPER(T, x, y)

supprimer x de liste des enfant de y, en décrémentant degré[y]

ajouter x à liste de racines de T

p[x] ← NIL

marqué[x] ← FAUX

COUPE-EN-CASCADE(T, y)

z ← p[y]

si z ≠ NIL

  alors si marqué[y] = FAUX

    alors marqué[y] ← VRAI

    sinon COUPER(T, y, z)

      COUPE-EN-CASCADE(T, z)

SUPPRIMER-TAS-FIB(T, x)

DIMINUER-CLÉ-TAS-FIB(T, x, -∞)

EXTRAIRE-MIN-TAS-FIB(T)

CHAPITRE 21 : Structures de données pour ensembles disjoints

```
COMPOSANTES-CONNEXES(G)
 pour chaque sommet v ∈ S[G]
     faire CRÉER-ENSEMBLE(v)
         pour chaque arête (u, v) ∈ A[G]
             faire si TROUVER-ENSEMBLE(u) fi TROUVER-ENSEMBLE(v)
                 alors UNION(u, v)
```

```
MÊME-COMPOSANTE(u, v)
 si TROUVER-ENSEMBLE(u) = TROUVER-ENSEMBLE(v)
     alors retourner VRAI
     sinon retourner FAUX
```

```
CRÉER-ENSEMBLE(x)
 p[x] ← x
 rang[x] ← 0
```

```
UNION(x, y)
 LIER(TROUVER-ENSEMBLE(x), TROUVER-ENSEMBLE(y))
```

```
LIER(x, y)
 si rang[x] > rang[y]
     alors p[y] ← x
     sinon p[x] ← y
         si rang[x] = rang[y]
             alors rang[y] ← rang[y] + 1
```

```
TROUVER-ENSEMBLE(x)
 si x ≠ p[x]
     alors p[x] ← TROUVER-ENSEMBLE(p[x])
         retourner p[x]
```

PARTIE 6 : ALGORITHMES POUR LES GRAPHES

CHAPITRE 22 : Algorithmes élémentaires pour les graphes

PARCOURS EN LARGEUR (PL)

```
PL(G, s)
 pour chaque sommet u ∈ S[G] - {s}
         faire couleur[u] ← BLANC
                  d[u] ← ∞
                  p[u] ← NIL
 couleur[s] ← GRIS
 d[s] ← 0
 p[s] ← NIL
 F ← {s}
 tant que F ≠ Ø
         faire u ← tête[F]
                  pour chaque v ∈ Adj[u]
                           faire si couleur[v] = BLANC
                                    alors couleur[v] ← GRIS
                                             d[v] ← d[u] + 1
                                             p[v] ← u
                                             ENFILE(F, v)
                                             DÉFILE(F)
 couleur[u] ← NOIR
```

IMPRIMER-CHEMIN(G, s, v)

```
si v = s
        alors imprimer s
        sinon si p[v] = NIL
                alors imprimer " il n'existe aucun chemin de " s " à " v
                sinon IMPRIMER-CHEMIN(G, s, p[v])
                        imprimer v
```

PARCOURS EN PROFONDEUR

PP(G)

```
pour chaque sommet u ∈ S[G]
        faire couleur[u] ← BLANC
                π[u] ← NIL
date ← 0
pour chaque sommet u ∈ S[G]
        faire si couleur[u] = BLANC
                alors VISITER-PP(u)
```

VISITER-PP(u)

```
couleur[u] ← GRIS sommet blanc u vient d'être découvert.
date ← date +1
d[u] ← date
pour chaque v ∈ Adj[u] Exploration de l'arc (u, v).
        faire si couleur[v] = BLANC
                alors p[v] ← u
                        VISITER-PP(v)
couleur[u] ← NOIR noircir u, car on en a fini avec lui.
f [u] ← date ← date +1
```

COMPOSANTES-FORTEMENT-CONNEXES(G)

appeler PP(G) pour calculer les dates de fin de traitement f [u] pour chaque sommet u

calculer ^{T}G

appeler PP(^{T}G), mais dans la boucle principale de PP, traiter les sommets par ordre de f [u] (calculés en ligne 1) décroissants

imprimer les sommets de chaque arborescence de la forêt obtenue en ligne 3 en tant que composante fortement connexe distincte

CHAPITRE 23 : Arbres couvrants de poids minimum

ACM-GÉNÉRIQUE(G, w)

E ← Ø

tant que E ne forme pas un arbre couvrant

faire trouver une arête (u, v) qui est sûre pour E

E ← E ∪ {(u, v)}

retourner E

ACM-KRUSKAL(G, w)

E ← Ø

pour chaque sommet v ∈ S[G]

faire CRÉER-ENSEMBLE(v)

trier les arêtes de A par ordre croissant de poids w

pour chaque arête (u, v) ∈ A pris par ordre de poids croissant

faire si TROUVER-ENSEMBLE(u) **fi** TROUVER-ENSEMBLE(v)

alors E ← E ∪ {(u, v)}

UNION(u, v)

retourner E

ACM-PRIM(G, w, r)

```
pour chaque u ∈ S[G]
        faire clé[u] ← ∞
                π[u] ← NIL
clé[r] ← 0
F ← S[G]
tant que F ≠ Ø
        faire u ← EXTRAIRE-MIN(F)
                pour chaque v ∈ Adj[u]
                        faire si v ∈ F et w(u, v) < clé[v]
                                alors p[v] ← u
                                        clé[v] ← w(u, v)
```

<u>ACM-RÉDUIRE(G, orig, c, T)</u>

```
pour chaque v ∈ S[G]
        faire marque[v] ← FAUX
                CRÉER-ENSEMBLE(v)
pour chaque u ∈ S[G]
        faire si marque[u] = FAUX
                alors choisir v ∈ Adj[u] tel que c[u, v] soit minimisé
                        UNION(u, v)
                        T ← T ∪ {orig[u, v]}
                        marque[u] ← marque[v] ← VRAI
S[G’] ← {TROUVER-ENSEMBLE(v) : v ∈ S[G]}
A[G’] ← Ø
pour chaque (x, y) ∈ A[G]
        faire u ← TROUVER-ENSEMBLE(x)
                v ← TROUVER-ENSEMBLE(y)
                si (u, v) ∉ A[G’]
                        alors A[G’] ← A[G’] ∪ {(u, v)}
```

```
                    orig'[u, v] ← orig[x, y]
                    c[u, v] ← c[x, y]
               sinon si c[x, y] < c'[u, v]
                    alors orig'[u, v] ← orig[x, y]
                         c'[u, v] ← c[x, y]
construire listes d'adjacences Adj pour G'
retourner G', orig', c' et T
```

23.4. Autres algorithmes d'arbre couvrant minimum

PEUT-ÊTRE-ACM-A(G, w)

```
trier les arêtes en ordre décroissant de poids d'arête w
T ← A
pour chaque arête a, prise par ordre décroissant de poids
     faire si T - {a} est un graphe connexe
          alors T ← T – a
retourner T
```

PEUT-ÊTRE-ACM-B(G, w)

```
T ← Ø
pour chaque arête a prise dans un ordre arbitraire
     faire si T ∪ {a} n'a pas de cycles
          alors T ← T ∪ a
retourner T
```

PEUT-ÊTRE-ACM-C(G, w)

```
T ← Ø
pour chaque arête a prise dans un ordre arbitraire
     faire T ← T ∪ {a}
          si T a un cycle c
               alors soit a une arête de poids maximal de c
                    T ← T - {a}
```

retourner T

CHAPITRE 24 : Plus courts chemins à origine unique

SOURCE-UNIQUE-INITIALISATION(G, s)

```
pour chaque sommet v ∈ S[G]
    faire d[v] ← ∞
        π[v] ← NIL
d[s] ← 0
```

RELÂCHER(u, v, w)

```
si d[v] > d[u] + w(u, v)
    alors d[v] ← d[u] + w(u, v)
π[v] ← u
```

BELLMAN-FORD(G, w, s)

```
SOURCE-UNIQUE-INITIALISATION(G, s)
pour i ← 1 à |S[G]| - 1
    faire pour chaque arc (u, v) ∈ A[G]
        faire RELÂCHER(u, v, w)
pour chaque arc (u, v) ∈ A[G]
    faire si d[v] > d[u] + w(u, v)
        alors retourner FAUX
retourner VRAI
```

PLUS-COURTS-CHEMINS-GSS(G, w, s)

```
trier topologiquement les sommets de G
SOURCE-UNIQUE-INITIALISATION(G, s)
pour chaque sommet u pris dans l'ordre topologique
    faire pour chaque sommet v ∈ Adj[u]
        faire RELÂCHER(u, v, w)
```

DIJKSTRA(G, w, s)

```
SOURCE-UNIQUE-INITIALISATION(G, s)
E ← Ø
F ← S[G]
tant que F ≠ Ø
    faire u ← EXTRAIRE-MIN(F)
        E ← E ∪ {u}
        pour chaque sommet v ∈ Adj[u]
            faire RELÂCHER(u, v, w)
```

CHAPITRE 25 : Plus courts chemins pour tout couple de sommets

IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES(Π, i, j)

```
si i = j
    alors imprimer i
    sinon si pij = NIL
        alors imprimer « il n'existe aucun chemin de » i « à » j
        sinon IMPRIMER-PLUS-COURT-CHEMIN-TOUS-COUPLES(P, i, pij)
            imprimer j
```

EXTENSION-PLUS-COURTS-CHEMINS(D, W)

```
n ← lignes[D]
soit D' = (d'ij ) une matrice n × n
pour i ← 1 à n
    faire pour j ← 1 à n
        faire d'ij ← ∞
            pour k ← 1 à n
                faire d'ij ← min(d'ij , dik + wkj)
retourner D'
```

MULTIPLIER-MATRICES(A, B)

```
n ← lignes[A]
soit C une matrice n × n
pour i ← 1 à n
        faire pour j ← 1 à n
        faire cij ← 0
                pour k ← 1 à n
                        faire cij ← cij + aik·bkj
retourner C
```

PLUS-COURT-CHEMIN-TOUS-COUPLES-RALENTI(W)

```
n ← lignes[W]
D(1) ← W
pour m ← 2 à n – 1
        faire D(m) ← EXTENSION-PLUS-COURTS-CHEMINS(D(m-1), W)
retourner D(n-1)
```

PLUS-COURT-CHEMIN-TOUS-COUPLES-ACCÉLÉRÉ(W)

```
n ← lignes[W]
D(1) ← W
m ← 1
tant que m < n – 1
        faire D(2m) ← EXTENSION-PLUS-COURTS-CHEMINS(D(m), D(m))
                m ← 2m
retourner D(m)
```

FLOYD-WARSHALL(W)

n ← lignes[W]

$D^{(0)}$ ← W

pour k ← 1 **à** n

faire pour i ← 1 **à** n

faire pour j ← 1 **à** n

faire $d^{(k)}ij \leftarrow \min(dij^{(k-1)}, dik^{(k-1)} + dkj^{(k-1)})$

retourner $D^{(n)}$

FERMETURE-TRANSITIVE(G)

n ← |S[G]|

pour i ← 1 **à** n

faire pour j ← 1 **à** n

faire si i = j or (i, j) ∈ A[G]

alors $t^{(0)}_{ij} \leftarrow 1$

sinon $t^{(0)}_{ij} \leftarrow 0$

pour k ← 1 **à** n

faire pour i ← 1 **à** n

faire pour j ← 1 **à** n

faire $t^{(k)}_{ij} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

retourner $T^{(n)}$

JOHNSON(G)

calculer G', où S[G'] = S[G] ∪ {s},
A[G'] = A[G] ∪ {(s, v) : v ∈ S[G]}, et
w(s, v) = 0 pour tout v ∈ S[G]

si BELLMAN-FORD(G', w, s) = FAUX
alors imprimer « le graphe contient un circuit de longueur strictement négative »
sinon pour chaque sommet v ∈ S[G']
faire affecter à h(v) la valeur de δ(s, v)
calculée par l'algorithme de Bellman-Ford
pour chaque arc (u, v) ∈ A[G']

```
        faire ŵ (u, v) ← w(u, v) + h(u) - h(v)
    pour chaque sommet u ∈ S[G]
        faire exécuter DIJKSTRA(G, ŵ, u) pour calculer δ̂ (u, v)
                                      pour tout v ∈ S[G]
            pour chaque sommet v ∈ S[G]
                faire d_uv ← δ̂ (u, v) + h(v) - h(u)
    retourner D
```

CHAPITRE 26 : Flot maximum

MÉTHODE-FORD-FULKERSON(G, s, t)

```
initialiser flot f à 0
tant que il existe un chemin améliorant p
    faire augmenter le flot f le long de p
retourner f
```

FORD-FULKERSON(G, s, t)

```
pour chaque arc (u, v) ∈ A[G]
    faire f [u, v] ← 0
     f [v, u] ← 0
tant que il existe un chemin p de s à t dans le réseau résiduel Gf
    faire cf (p) ← min {cf (u, v) : (u, v) is in p}
        pour chaque arc (u, v) de p
            faire f [u, v] ← f [u, v] + cf (p)
                f [v, u] ← -f [u, v]
```

POUSSER(u, v)

s'applique quand : u déborde, cf [u, v] > 0 et h[u] = h[v] + 1.

action : pousser df (u, v) = min(e[u], cf (u, v)) unités de flot de u vers v.

df (u, v) ← min(e[u], cf (u, v))

f [u, v] ← f [u, v] + df (u, v)

f [v, u] ← -f [u, v]

e[u] ← e[u] - df (u, v)

e[v] ← e[v] + df (u, v)

RÉÉTIQUETER(u)

s'applique quand : u est débordant et, pour tout v ∈ S tel que (u, v) ∈ Af ,

on a h[u] ≤ h[v].

action : accroît la hauteur de u.

h[u] ← 1 + min {h[v] : (u, v) ∈ A_f }

INITIALISER-PRÉFLOT(G, s)

pour chaque sommet u ∈ S[G]

faire h[u] ← 0

e[u] ← 0

pour chaque arc (u, v) ∈ A[G]

faire f [u, v] ← 0

f [v, u] ← 0

h[s] ← |S[G]|

pour chaque sommet u ∈ Adj[s]

faire f [s, u] ← c(s, u)

f [u, s] ← -c(s, u)

e[u] ← c(s, u)

e[s] ← e[s] - c(s, u)

PRÉFLOT-GÉNÉRIQUE(G)

```
INITIALISER-PRÉFLOT(G, s)
tant que il est possible d'appliquer un poussage ou un ré-étiquetage
    faire choisir un poussage ou ré-étiquetage applicable et l'exécuter
```

DÉCHARGER(u)

```
tant que e[u] > 0
    faire v ← courant[u]
        si v = NIL
            alors RÉÉTIQUETER(u)
                courant[u] ← tête[N[u]]
        sinon si cf (u, v) > 0 et h[u] = h[v] + 1
            alors POUSSER(u, v)
        sinon courant[u] ← voisin-suivant[v]
```

RÉÉTIQUETER-VERS-L'AVANT(G, s, t)

```
INITIALISER-PRÉFLOT(G, s)
L ← S[G] - {s, t}, dans un ordre quelconque
pour chaque sommet u ∈ S[G] - {s, t}
    faire courant[u] ← tête[N[u]]
u ← tête[L]
tant que u fi NIL
    faire ancienne-hauteur ← h[u]
        DÉCHARGER(u)
        si h[u] > ancienne-hauteur
            alors déplacer u vers le début de la liste L
        u ← suivant[u]
```

FLOT-MAX-ET-ECHELONNEMENT(G, s, t)

C ← max(u,v)∈ A c(u, v)

initialiser flot f à 0

K ← 2lg C

tant que K ≥ 1

faire tant que il existe un chemin améliorant p de capacité au moins K

faire augmenter flot f le long de p

K ← K/2

retourner f

HOPCROFT-KARP(G)

M ← Ø

Répéter

soit P ← {P1, P2, . . . , Pk} un ensemble maximum de plus courts chaînes améliorantes par rapport à M sans sommet commun

M ← M ⊕ (P1 ∪ P2 ∪ · · · ∪ Pk)

jusqu'à P = Ø

retourner M

PARTIE 7 : MORCEAUX CHOISIS

CHAPITRE 27 : Réseaux de tri

CHAPITRE 28 : Calcul matriciel

RÉSOLUTION-LUP(L, U, **p**, b)

n ← lignes[L]

pour i ← 1 **à** n

faire $y_i \leftarrow b_{\pi[\mathbf{i}]} - \sum_{j=1}^{i-1} l_{ij} y_j$

pour i ← n **jusqu'à** 1

faire $x_i \leftarrow (y_i - \sum_{j=i+1}^{n} u_{ij} x_j) / u_{ii}$

retourner x

DÉCOMPOSITION-LU(A)

n ← lignes[A]

pour k ← 1 **à** n

faire ukk ← akk

pour i ← k + 1 **à** n

faire lik ← aik/ukk lik contient vi

uki ← aki uki contient Twi

pour i ← k + 1 **à** n

faire pour j ← k + 1 **à** n

faire aij ← aij – likukj

retourner L et U

DÉCOMPOSITION-LUP(A)

n ← lignes[A]

pour i ← 1 **à** n

faire π[i] ← i

pour k ← 1 **à** n

faire p ← 0

pour i ← k **à** n

faire si |aik| > p

alors p ← |aik|

k' ← i

si p = 0

alors erreur « matrice singulière »

permuter **π** [k] ↔ **π** [k']

pour i ← 1 **à** n

faire permuter aki ↔ ak'i

pour i ← k + 1 **à** n

faire aik ← aik/akk

pour j ← k + 1 **à** n

faire aij ← aij - aikakj

<u>CHAPITRE 29 :</u> Programmation linéaire

<u>PIVOT(N, B, A, b, c, v, l, e)</u>

Calcule les coefficients de l'équation pour nouvelle variable de base xe.

b̂e ← bl/ale

pour tout j ∈ N - {e}

faire **â**ej ← alj/ale

âel ← 1/ale

Calcule les coefficients des contraintes restantes.

pour tout i ∈ B - {l}

faire **b̂**i ← bi – aie**b̂**e

pour tout j ∈ N - {e}

faire **â**ij ← aij – aie**â**ej

âil ← -aie**â**el

Calcule la fonction objectif.

v̂ ← v + ce**b̂**e

pour tout j ∈ N - {e}

faire **ĉ**j ← cj – ce**â**ej

ĉl ← -ce**â**el

Calcule nouveaux ensembles de variables de base/hors-base.

$\widehat{\mathbf{N}}$ = N - {e} ∪ {l}

$\widehat{\mathbf{B}}$ = B - {l} ∪ {e}

retourner ($\widehat{\mathbf{N}}$, $\widehat{\mathbf{B}}$, $\widehat{\mathbf{A}}$, $\hat{\mathbf{b}}$, $\hat{\mathbf{c}}$, $\hat{\mathbf{v}}$)

SIMPLEXE(A, b, c)

(N, B, A, b, c, v) ← INITIALISE-SIMPLEXE(A, b, c)

tant que qu'un indice j ∈ N vérifie cj > 0

faire choisir un indice e ∈ N pour lequel ce > 0

pour tout indice i ∈ B

faire si aie > 0

alors Δi ← bi/aie

sinon Δi ← ∞

choisir un indice l ∈ B qui minimise Δi

si Δl = ∞

alors retourner « non borné »

sinon (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, e)

pour i ← 1 **à** n

faire si i ∈ B

alors $\bar{\mathbf{x}}$i ← bi

sinon $\bar{\mathbf{x}}$i ← 0

retourner ($\bar{\mathbf{x}}$1, $\bar{\mathbf{x}}$2, . . . , $\bar{\mathbf{x}}$n)

INITIALISE-SIMPLEXE(A, b, c)

soit l l'indice du bi minimal

si bl $\geq$ 0 Est-ce que la solution de base initiale est réalisable ?

alors retourner ({1, 2, . . . , n} , {n + 1, n + 2, . . . , n + m} , A, b, c, 0)

former Laux en ajoutant -x0 au membre gauche de chaque équation et en choisissant la fonction objectif sur -x0

soit (N, B, A, b, c, v) la forme standard résultante pour Laux

Laux a n + 1 variables hors-base et m variables de base.

(N, B, A, b, c, v) $\leftarrow$ PIVOT(N, B, A, b, c, v, l, 0)

La solution de base est maintenant réalisable pour Laux.

itérer la boucle **tant que** des lignes 2–11 de SIMPLEXE jusqu'à obtention d'une solution optimale pour Laux

si la solution de base donne $\overline{x}0 = 0$

alors retourner la forme standard finale en supprimant x0 et en restaurant la fonction objectif d'origine

sinon retourner « irréalisable »

<u>CHAPITRE 30 :</u> Polynômes et transformée rapide de Fourier

<u>FFT-RÉCURSIVE(a)</u>

n $\leftarrow$ longueur[a] n est une puissance de 2.

si n = 1

alors retourner a

wn $\leftarrow$ $e^{2\mathbf{p}i/n}$

w $\leftarrow$ 1

$a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$

$a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$

$y^{[0]} \leftarrow$ FFT-RÉCURSIVE($a^{[0]}$)

$y^{[1]} \leftarrow$ FFT-RÉCURSIVE($a^{[1]}$)

pour k $\leftarrow$ 0 **à** n/2 – 1

faire $y_k \leftarrow y^{[0]}{}_k + \mathbf{w}\, y^{[1]}{}_k$

$y_{k+(n/2)} \leftarrow y^{[0]}{}_k - \mathbf{w}\, y^{[1]}{}_k$

$\mathbf{w} \leftarrow \mathbf{w}\, \mathbf{w}_n$

retourner y y est supposé être un vecteur colonne.

<u>FFT-ITÉRATIVE(a)</u>

COPIE-INVERSION-BITS(a, A)

n ← longueur[a] n est une puissance de 2.

pour s ← 1 **à** lg n

faire $m \leftarrow 2^s$

$\mathbf{w}_m \leftarrow e^{2\pi i/m}$

pour k ← 0 **à** n - 1 **avec un pas** m

faire w ← 1

pour j ← 0 **à** m/2 – 1

faire t ← **w**A[k + j + m/2]

u ← A[k + j]

A[k + j] ← u + t

A[k + j + m/2] ← u – t

$\mathbf{w} \leftarrow \mathbf{w}\, \mathbf{w}_m$

<u>COPIE-INVERSION-BITS(a, A)</u>

n ← longueur[a]

pour k ← 0 **à** n – 1

faire A[inv(k)] ← a_k

<u>CHAPITRE 31 :</u> Algorithmes de la théorie des nombres

<u>EUCLIDE(a, b)</u>

```
si b = 0
        alors retourner a
        sinon retourner EUCLIDE(b, a mod b)
```

<u>EUCLIDE-ETENDU(a, b)</u>

```
si b = 0
        alors retourner (a, 1, 0)
(d', x', y') ← EUCLIDE-ETENDU(b, a mod b)
(d, x, y) ← (d', y', x' – [a/b]y')
retourner (d, x, y)
```

<u>RÉSOLUTION-EQUATIONS-LINÉAIRES-MODULAIRES(a, b, n)</u>

```
(d, x', y') ← EUCLIDE-ETENDU(a, n)
si d | b
        alors x0 ← x'(b/d) mod n
                pour i ← 0 à d – 1
                        faire imprimer (x0 + i(n/d)) mod n
sinon imprimer « pas de solution »
```

<u>EXPONENTIATION-MODULAIRE(a, b, n)</u>

```
c ← 0
d ← 1
soit bk, bk-1, . . . , b0 la représentation binaire de b
pour i ← k decr jusqu'à 0
        faire c ← 2c
```

```
        d ← (d·d) mod n
        si bi = 1
            alors c ← c + 1
                d ← (d·a) mod n
retourner d
```

PSEUDO-PREMIER(n)

si EXPONENTIATION-MODULAIRE(2, n - 1, n) $\not\equiv 1 \pmod n$

alors retourner COMPOSÉ c'est sûr !

sinon retourner PREMIER Avec un peu de chance !

TÉMOIN(a, n)

soit $n - 1 = 2^t u$, avec $t \geq 1$ et u impair
x0 ← EXPONENTIATION-MODULAIRE(a, u, n)
pour i ← 1 **à** t
    faire xi ← x^2_{i-1} mod n
        si xi = 1 et xi-1 ≠ 1 et $x_{i-1} \neq n - 1$

            alors retourner VRAI

si xt ≠ 1

    alors retourner VRAI

retourner FAUX

MILLER-RABIN(n, s)

pour j ← 1 **à** s

    faire a ← RANDOM(1, n - 1)

        si TÉMOIN(a, n)

            alors retourner COMPOSÉ   c'est sûr !

retourner PREMIER   c'est à peu près sûr.

POLLARD-RHO(n)

```
i ← 1
x1 ← RANDOM(0, n - 1)
y ← x1
k ← 2
tant que VRAI
        faire i ← i + 1
                xi ← (x²i-1 - 1) mod n
                d ← pgcd(y - xi, n)
                si d ≠ 1 et d ≠ n
                        alors afficher d
                si i = k
                        alors y ← xi
                                k ← 2k
```

CHAPITRE 32 : Recherche de chaînes de caractères

RECHERCHE-NAÏVE(T, P)

```
n ← longueur[T]
m ← longueur[P]
pour s ← 0 à n – m
        faire si P[1 . . m] = T[s + 1 . . s + m]
                alors afficher « La chaîne apparaît avec le décalage » s
```

RABIN-KARP(T, P, d, q)

```
n ← longueur[T]
m ← longueur[P]
h ← d^(m-1) mod q
p ← 0
t0 ← 0
pour i ← 1 à m                                          pré traitement
        faire p ← (dp + P[i]) mod q
                t0 ← (dt0 + T[i]) mod q
pour s ← 0 à n - m                                      recherche de correspondance
```

faire si p = ts

alors si P[1 . . m] = T[s + 1 . . s + m]

alors afficher « la chaîne recherchée apparaît la position » s

si s < n – m

alors ts+1 ← (d(ts - T[s + 1]h) + T[s + m + 1]) mod q

<u>RECHERCHE-AUTOMATE-FINI(T, **d**, m)</u>

n ← longueur[T]

q ← 0

pour i ← 1 **à** n

faire q ← δ(q, T[i])

si q = m

alors s ← i – m

afficher « Le motif apparaît à la position » s

<u>CALCUL-FONCTION-TRANSITION(P, Σ)</u>

m ← longueur[P]

pour q ← 0 **à** m

faire pour chaque caractère a ∈ Σ

faire k ← min(m + 1, q + 2)

répéter k ← k – 1

jusqu'à Pk ⊐ P_qa

δ (q, a) ← k

retourner δ

CALCUL-FONCTION-PRÉFIXE(P)

```
m ← longueur[P]
π [1] ← 0
k ← 0
pour q ← 2 à m
        faire tant que k > 0 et P[k + 1] ≠ P[q]
                faire k ← π[k]
           si P[k + 1] = P[q]
                alors k ← k + 1
           π [q] ← k
retourner π
```

CHAPITRE 33 : Géométrie algorithmique

INTERSECTION-SEGMENTS(p1, p2, p3, p4)

```
d1 ← DIRECTION(p3, p4, p1)
d2 ← DIRECTION(p3, p4, p2)
d3 ← DIRECTION(p1, p2, p3)
d4 ← DIRECTION(p1, p2, p4)
si ((d1 > 0 et d2 < 0) ou (d1 < 0 et d2 > 0)) et
((d3 > 0 et d4 < 0) ou (d3 < 0 et d4 > 0))
        alors retourner VRAI
sinon si d1 = 0 et SUR-SEGMENT(p3, p4, p1)
        alors retourner VRAI
sinon si d2 = 0 et SUR-SEGMENT(p3, p4, p2)
        alors retourner VRAI
sinon si d3 = 0 et SUR-SEGMENT(p1, p2, p3)
        alors retourner VRAI
sinon si d4 = 0 et SUR-SEGMENT(p1, p2, p4)
        alors retourner VRAI
sinon retourner FAUX
```

DIRECTION(pi, pj, pk)

retourner (pk - pi) × (pj - pi)

SUR-SEGMENT(pi, pj, pk)

si min(xi, xj) xk max(xi, xj) et min(yi, yj) yk max(yi, yj)

alors retourner VRAI

sinon retourner FAUX

INTERSECTION-DEUX-SEGMENTS-QUELCONQUES(*S*)

T ← Ø
trier les extrémités des segments de *S* de la gauche vers la droite,
si égalité, placer les extrémités gauches avant les extrémités droites
puis, dans chacun de ces deux ensembles, trier dans l'ordre des ordonnées des points
pour tout point *p* de la liste triée des extrémités
faire si *p* est l'extrémité gauche d'un segment *s*
alors INSÉRER(*T, s*)
si (AU-DESSUS(*T, s*) existe et coupe *s*)
ou (AU-DESSOUS(*T, s*) existe et coupe *s*)
alors retourner VRAI
si *p* est l'extrémité droite d'un segment
alors si AU-DESSUS(*T, s*) et AU-DESSOUS(*T, s*) exist
et AU-DESSUS(*T, s*) coupe AU-DESSOUS(*T, s*)
alors retourner VRAI
SUPPRIMER(*T, s*)
retourner FAUX

BALAYAGE-GRAHAM(*Q*)

soit *p*0 le point de *Q* ayant l'ordonnée minimale, ou, en cas d'égalité, le point d'ordonnée
minimale qui est le plus à gauche
soient <*p*1, *p*2, . . . , *pm*> les autres points de *Q*, triés par angles polaires (mesurés relativement à
*p*0 dans l'ordre inverse des
aiguilles d'une montre) (si plusieurs points ont le même angle, on les supprime tous
sauf celui qui est le plus loin de *p*0)
EMPILER(*p*0, *S*)
EMPILER(*p*1, *S*)
EMPILER(*p*2, *S*)
pour *i* ← 3 **à** *m*
faire tant que l'angle formé par les points SOUS-SOMMET(*S*)
SOMMET(*S*) et *pi* fait un tour non à gauche
faire DÉPILER(*S*)

EMPILER(*pi*, *S*)

retourner *S*

CHAPITRE 34 : NP-complétude

CHAPITRE 35 : Algorithmes d'approximation

COUVERTURE-SOMMET-APPROCHÉE(*G*)

$C \leftarrow \emptyset$

$A' \leftarrow A[G]$

tant que $A' \neq \emptyset$

faire soit (u, v) une arête arbitraire de A'

$C \leftarrow C \cup \{u, v\}$

supprimer de A' toutes les arêtes incidentes à *u* ou à *v*

retourner *C*

TOURNÉE-VC-APPROCHÉE(*G, c*)

sélectionner un sommet $r \in V[G]$ pour faire office de « racine »

calculer un ACM *T* pour *G* depuis la racine *r* en utilisant ACM-PRIM(*G, c, r*)

soit *L* la liste des sommets visités dans un parcours préfixe de *T*

retourner le cycle hamiltonien *H* qui visite les sommets dans l'ordre *L*

COUVERTURE-ENSEMBLE-GLOUTON(*X, F*)

$U \leftarrow X$

$C \leftarrow \emptyset$

tant que $U \neq \emptyset$

faire choisir un $S \in F$ qui maximise $|S \cap U|$

U ← *U* − *S*

C ← *C* ∪ {*S*}

retourner *C*

<u>COUVERTURE-SOMMET-PONDÉRÉ-APPROCHÉE(*G, w*)</u>

C ← ∅

calculer $\bar{x}$, solution optimale du programme linéaire des lignes (35.15)–(35.18)

pour tout *v* ∈ *V*

faire si $\bar{x}$(*v*) ≥ ½

alors *C* ← *C* ∪ {*v*}

retourner *C*

<u>SOMME-EXACTE-SOUS-ENSEMBLE(*S, t*)</u>

n ← |*S*|

*L*0 ← <0>

pour *i* ← 1 **à** *n*

faire *Li* ← FUSIONNER-LISTES(*Li*-1*, Li*-1 + *xi*)

supprimer de *Li* tout élément supérieur à *t*

retourner le plus grand élément de *Ln*

<u>ÉPURER(L, δ)</u>

m ← |L|

L ← <y1>

dernier ← y1

pour i ← 2 **à** m

faire si yi > dernier ·(1 + δ) yi ≥ dernier car L est triée

alors ajouter yi à la fin de L'

dernier ← yi

retourner L'

SOMME-SOUS-ENSEMBLE-APPROCHÉE(S, t, ε)

n ← |S|

L0 ← <0>

pour i ← 1 **à** n

faire Li ← FUSIONNER-LISTES(Li-1, Li-1 + xi)

Li ← ÉPURER(Li, ε /2n)

supprimer de Li chaque élément qui est supérieur à t

soit z* la valeur maximale de Ln

retourner z*

PARTIE 8 : ANNEXES : ÉLÉMENTS DE MATHÉMATIQUES