1 Introduction

1.1 Complexité

— Si une fonction est la somme de plusieurs termes, si l'un d'eux croit plus vite que les autres, on garde uniquement celui-là et l'on ignore les autres.

— Si une fonction est le produit de plusieurs facteurs, on peut ignorer tout facteur constant.

— $o(g)$ strictement plus lentement

— $O(g)$ au plus aussi vite

— $\Omega(g)$ au moins aussi vite

— $\Theta(g)$ même ordre de grandeur

1.2 Structures de données

TDA : types de données abstrait. Type + ensemble des opérations (implémentation cachée). Il existe quatre opérations sur les TDA :

1. **constructeurs** : créent des instances initialisées de la structure
2. **modificateurs** : append, assign, clear, erase, insert, resize, push_back
3. **sélecteurs** : capacity, find, substr
4. **itérateurs** : begin, end, rbegin, rend

2 Récursivité

Table 1 : Complexités de quelques algorithmes

Algorithme	Complexité
factorielle récursif / itératif	$O(n)$
Fibonacci récursif / itératif	$O(\Theta^n)$ / $O(n)$
PGCD (Euclide)	$O(\log n)$
Tours de Hanoï récursif / itératif	$O(2^n)$
Permutations	$O(n!)$
Tic Tac Toe	9!
Puissance 4, exploration de d tours	$O(7^d)$
Minimax (negamax), m mouvements possibles, profondeur de d tours	$O(m^d)$
`std::upper_bound`, `std::equal_range`, `std::binary_search`	$O(\log n)$
`std::generate`, `std::nth_element`, find / search	$O(n)$
`std::stable_sort`	$O(n \log n)$
`std::partial_sort`	$O(n \log m)$ (m = middle)

3 Tris

Table 2 : Tris et compléxités

Tri	Meilleur cas	En moyenne	Pire cas	Stable
bubble	$O(n)$	$O(n^2)$	$O(n^2)$	Oui
insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Oui
sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Non
fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Oui
rapide	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Non
comptage	-	$O(n+b)$	-	Oui
par base	-	$O(d(n+b))$	-	Oui

Table 3 : Tris et propriétés

Tri	Indépendant ordre données en entrée	Tri par échange	En place
bubble	Non	Oui	Oui
insertion	Non	Non	Oui
sélection	Oui	Oui	Oui
fusion	Oui	Non	Non
rapide	Non	Oui	Oui
comptage	Oui	Non	Non
par base	Oui	Non	Non

3.1 Tris simples

3.1.1 Bubble sort

— $O(n^2)$ comparaisons

— Entre 0 et $O(n^2)$ échanges

Idée : permuter deux voisins si celui de droite est plus petit que celui de gauche. Stable si l'opérateur utilisé est $<$.

Pattern : derniers éléments triés

Algorithme 1 : Tri à bulles

```
Fonction bubbleSort(A, n) : // A tab. de n élém.
    pour i ← [1, n − 1] faire
        pour j ← [1, n − i] faire
            si A[j + 1] < A[j] alors
                échanger A[j] et A[j + 1]
```

3.1.2 Selection sort

— $O(n^2)$ comparaisons

— $O(n)$ échanges

Idée : sélectionner le plus petit élément et le placer au début du tableau.

Pattern : début trié, mais fin modifiée

Algorithme 2 : Tri par sélection

```
Fonction selectionSort(A, n) :
    pour i ← [1, n − 1] faire
        i_min ← i
        pour j ← [i + 1, n] faire
            si A[j] < A[i_min] alors
                i_min ← j
        échanger A[i] et A[i_min]
```

3.1.3 Insertion sort

Idée : Placer l'élément $i+1$ à la bonne position dans la partie du tableau déjà triée (jusqu'à i inclus).

Pattern : début trié et fin inchangée

Algorithme 3 : Tri par insertion

```
Fonction insertionSort(A, n) :
    pour i ← [2, n] faire
        tmp ← A[i]
        j ← i
        tant que j − 1 ≥ 1 et A[j − 1] > tmp
          faire
            A[j] ← A[j − 1]
            j ← j − 1
        A[j] ← tmp
```

3.2 Tri par fusion

Idée : deux sous-tableaux triés, on compare les deux plus petits éléments des deux tableaux, on déplace le plus petit des deux dans le tableau trié et on répète jusqu'à ce que les deux tableaux soient vides.

Pattern : Valeur coupée en deux et chaque partie est à peu près triée.

Complexité spatiale : Demande la création d'un tableau de même taille que le tableau initial (le tri ne se fait pas en place).

Algorithme 4 : Fusion de deux sous-tableaux

```
Fonction fusion(A, p, q, r) :
    L ← sous-liste A[p .. q]
    R ← sous-liste A[q + 1 .. r]
    i, j ← 1
    pour k ← [p, r] faire
        si L[i] ≤ R[j] alors
            A[k] ← L[i]
            i ← i + 1
        sinon
            A[k] ← R[j]
            j ← j + 1
```

Algorithme 5 : Tri par fusion

```
Fonction fusionSort(A, lo, hi) :
    si hi ≤ lo alors retourner
    mid ← lo + (hi − lo)/2
    fusionSort(A, lo, mid)
    fusionSort(A, mid + 1, hi)
    fusion(A, lo, mid, hi)
```

3.3 Tri rapide

Pattern : Début en partie trié et le pivot placé à la fin de la partie triée.

Algorithme 6 : Algorithme de partition

```
Fonction partition(A, lo, hi) :
    i ← lo − 1
    j ← hi
    répéter
        tant que A[i] < A[hi] faire
            i ← i + 1
        tant que j > lo et A[hi] < A[j] faire
            j ← j − 1
        si i ≥ j alors sortir boucle
        échanger A[i] et A[j]
    échanger A[i] et A[hi]
    retourner i
```

Algorithme 7 : Tri rapide

```
Fonction quickSort(A, lo, hi) :
    si lo < hi alors
        p ← choisir l'élément pivot
        échanger A[hi] et A[p]
        i ← partition(A, lo, i − 1)
        quickSort(A, lo, i − 1)
        quickSort(A, i + 1, hi)
```

Algorithme 8 : Tri rapide semi-récursif

```
Fonction quickSort(A, lo, hi) :
    si lo < hi alors
        p ← choisir l'élément pivot
        échanger A[hi] et A[p]
        i ← partition(A, lo, i − 1)
        si i < lo alors
            quickSort(A, lo, i − 1)
            lo ← i + 1
        sinon
            quickSort(A, i + 1, hi)
            hi ← i − 1
```

3.3.1 Sélection rapide

Complexité moyenne : $O(n)$. Complexité dans le pire cas : $O(n^2)$

Algorithme 9 : Sélection rapide

```
Fonction quickSelect(A, n, k) :
    lo ← 1
    hi ← n
    tant que hi > lo faire
        i ← partition(A, lo, hi)
        si i < k alors
            lo ← i + 1
        sinon si i > k alors
            hi ← i − 1
        sinon
            retourner A[k]
    retourner A[k]
```

3.3.2 Exemple

Figure 1 : Traitement de la partition

```
E X E M P L E D E T R I     i=0, j=12
E X E M P L E D E T R I     ++i = 1
E X E M P L E D E T R I     ++i = 2
E X E M P L E D E T R I     --j = 11
E X E M P L E D E T R I     --j = 10
E X E M P L E D E T R I     --j = 9
E E E M P L E D X T R I     echange(2,9)
```

Figure 2 : Traitement de la récursion

```
lo  i hi    1 2 3 4 5 6 7 8 9 10 11 12
            E X E M P L E D E T  R  I
 1  6 12    E E E D E I P M X T  R  L
 1  3  5    D E E E E I P M X T  R  L
 1  2  2    D E E E E I P M X T  R  L
 1     1    D E E E E I P M X T  R  L
```

3.4 Tri comptage

Algorithme 10 : Tri comptage

```
// A tableau de n éléments dont b éléments
   disctincts, key une fonction de
   catégorisation donnant la clé en
   fonction d'un ordre
Fonction countSort(A, n, b, key) :
    C ← tableau de b compteurs à 0
    pour tous e ∈ A faire
        C[key(e)] ← C[key(e)] + 1
    idx ← 1
    pour i ← [1, b] faire
        tmp ← C[i]
        C[i] ← idx
        idx ← idx + tmp
    B ← tableau de taille n
    pour tous e ∈ A faire
        B[C[key(e)]] ← e
        C[key(e)] ← C[key(e)] + 1
    retourner B
```

3.4.1 Tri par base

Algorithme 11 : Tri par base

```
Fonction radixSort(A, d) :
    pour i ← [d, 1] faire
        trier le tableau A avec un tri stable
          selon le i-ème chiffre
```

4 Structures linéaires

4.1 Tableaux

4.1.1 Tableaux de taille fixe (`std::array`)

— Pas d'insertion / suppression d'éléments.

— Permet un accès en O(1) en calculant l'offset via une taille (`sizeof`).

— Fourni nativement en C et C++.

— **Consommation mémoire** — structure vide : |éléments| · `sizeof(élément)` ; par élément : `sizeof(élément)`

4.1.2 Tableaux de capacité fixe (à la C)

— Il faut garder en mémoire la taille

— Insertion et suppression à la fin : O(1)

— Insertion et suppression à la position i : linéaire en O(taille - i)

— Insertion et suppression au début : linéaire en O(taille)

4.1.3 Buffer circulaire

— Permet de faire des files FIFO (first-in first-out, insère à la fin, supprime au début)

— Insertion, suppression et accès en O(1)

— Insertion libre linéaire en O(min(i, $taille - i$))

Algorithme 12 : Structure d'un buffer circulaire

```
Classe BufferCirculaire :
    capacité            // entier constant
    début, taille       // entiers variables
    data     // tableau de #capacité éléments
```

Algorithme 13 : Trouver l'index physique

```
Fonction iPhysique(iLogique) :
    retourner
      (début + iLogique + capacité) % capacité
```

Algorithme 14 : Insérer en fin

```
Fonction insérerEnFin(v) :
    si taille ≥ capacité alors alerter
    construire data en position
      iPhysique(taille) ← v
    taille ← taille + 1
```

Algorithme 15 : Insérer au début

```
Fonction insérerAuDébut(v) :
    si taille ≥ capacité alors alerter
    avant ← iPhysique(capacité − 1)
    construire data en position avant ← v
    début ← avant
    taille ← taille + 1
```

Algorithme 16 : Supprimer en fin

```
Fonction supprimerEnFin(v) :
    si taille = 0 alors alerter
    détruire data en position
      iPhysique(taille − 1)
    taille ← taille − 1
```

Algorithme 17 : Supprimer au début

```
Fonction supprimerAuDébut(v) :
    si taille = 0 alors alerter
    détruire data en position
      iPhysique(taille − 1)
    début ← iPhysique(1)
    taille ← taille − 1
```

4.1.4 Tableaux de capacité variable (`std::vector`)

— Capacité modifiée à l'insertion si nécessaire.

— Diminution sur demande, divise par deux si 4x supérieur à la taille. (`shrink_to_fit`)

— **Consommation mémoire** — structure vide : 3 pointeurs ; par élément : `sizeof(élément)`

Algorithme 18 : Structure d'un buffer circulaire

```
Classe TableauCapacitéVariable :
    capacité, taille      // entiers variables
    data*         // pointeur du 1er élément
```

Algorithme 19 : Déterminer la nouvelle taille

```
Fonction nouvelleTaille() :
    si capacité = 0 alors
        retourner 1
    sinon
        retourner 2 · capacité
```

Table 4 : Complexités des opérations de `std::vector`

Opération	en moyenne	au pire
Consultation	O(1)	O(1)
Insertion en fin	O(1)	O(n)
... au milieu	O(n-i)	O(n-i)
... au début	O(n)	O(n)
Suppression en fin	O(1)	O(1)
... au milieu	O(n-i)	O(n-i)
... au début	O(n)	O(n)
Taille	O(1)	O(1)
Suivant(s)	O(1)	O(1)
Distance	O(1)	O(1)

4.2 Listes

4.2.1 Liste simplement chaînées (`std::forward_list`)

— Stockés dans des emplacements mémoire non consécutifs ⇒ pas de relation suivant/-précédent implicite

— Efficace pour insérer à une position connue

— Peut être encapsulé dans une classe avec la référence vers le premier élément.

— **Consommation mémoire** — structure vide : 1 pointeur ; par élément : 1 pointeur + `sizeof(élément)`

Algorithme 20 : Structure d'un maillon de la chaîne

```
Structure Maillon :
    valeur    // valeur variable quelconque
    suivant*     // ptr sur Maillon suivant
```

Algorithme 21 : Parcours de la liste

```
Fonction afficherListe(premier) :
    courant* ← premier   // ptr de Maillon
    tant que courant ≠ ∅ faire
        afficher "la valeur de courant →"
        courant ← le maillon après courant
    afficher "∅"
```

Algorithme 22 : Insertion au début

```
Fonction insérerAuDébut(premier, v) :
    retourner nouveau Maillon(v, premier)
```

Algorithme 23 : Supprimer au début

```
Fonction retirerAuDébut(premier) :
    si premier = ∅ alors alerter
    temp ← le maillon après premier
    détruire premier ; retourner temp
```

Algorithme 24 : Insertion après

```
Fonction insérerAprès(m, v) :
    si m = ∅ alors alerter
    m.suivant ← nouveau
      Maillon(v, m.suivant)
```

Algorithme 25 : Supprimer après

```
Fonction retirerAprès(m) :
    si m ou m.suivant = ∅ alors alerter
    temp ← le maillon après m
    m.suivant ← temp.suivant
    détruire temp
```

Algorithme 26 : Ôter les répétitions

```
Fonction ôterRépétitions(premier) :
    m ← premier
    tant que m ≠ ∅ et m.suivant ≠ ∅ faire
        si m.valeur = m.suivant.valeur
          alors
            retirerAprès(m)
        sinon
            m ← m.suivant
```

Table 5 : Complexités des opérations de `std::forward_list`

Opération	en moyenne	au pire
Consultation	-	-
Insertion en fin	O(n)	O(n)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Suppression en fin	O(n)	O(n)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Taille	O(n)	O(n)
Suivant(s)	O(1)	O(n)
Distance	O(n)	O(n)

4.2.2 Listes doublement chaînées

Algorithme 27 : Structure d'un maillon de la chaîne

```
Structure Maillon :
    valeur      // valeur variable quelconque
    suivant*        // ptr sur Maillon suivant
    précédent*  // ptr sur Maillon précédent
```

Algorithme 28 : Insertion devant un maillon

```
Fonction insérerDevant(m, v) :
    p ← m.précédent
    n ← nouveau Maillon(v, m, p)
    p.suivant ← n
    m.précédent ← p
```

Algorithme 29 : Suppression

```
Fonction suppression(m) :
    m.précédent.suivant ← m.suivant
    m.suivant.précédent ← m.précédent
    détruire m
```

Algorithme 30 : Échange de maillons

```
Fonction échangeMaillons(m1, m2) :
    si m1 = m2 alors retourner
    m1p ← m1.précédent
    m1s ← m1.suivant
    m2p ← m2.précédent
    m2s ← m2.suivant
    échanger m1.précédent et m2.précédent
    échanger m1p.suivant et m2p.suivant
    échanger m1.suivant et m2.suivant
    échanger m1s.précédent et
      m2s.précédent
```

Algorithme 31 : Tri par sélection

```
Fonction triSélection(liste) :
    j ← premier maillon de liste
    tant que j ≠ liste.fin faire
        iMin ← j
        i ← j.suivant
        tant que i ≠ liste.fin faire
            si i.valeur < iMin.valeur alors
                iMin ← i
            i ← i.suivant
        échangeMaillons(imin, j)
        j ← iMin.suivant
```

Algorithme 32 : Tri par insertion

```
Fonction triInsertion(liste) :
    si liste est vide alors retourner
    k ← second maillon de liste
    tant que k ≠ liste.fin faire
        tmp ← k.valeur
        i ← k
        pi ← i.précédent
        tant que i ≠ liste.début et
          tmp < pi.valeur faire
            i.valeur ← pi.valeur
            i ← pi
            pi ← i.précédent
        i.valeur ← tmp
        k ← k.suivant
```

Algorithme 33 : Splice

```
Fonction splice(m1, d, f) :
    si d = f alors retourner
    m1p ← m1.précédent
    dp ← d.précédent
    fp ← f.précédent
    dp.suivant ← f
    f.précédent ← dp
    m1p.suivant ← d
    d.précédent ← m1p
    fp.suivant ← m1
    m1.précédent ← fp
```

Table 6 : Complexités des opérations de `std::list`

Opération	en moyenne	au pire
Insertion en fin	O(1)	O(1)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Suppression en fin	O(1)	O(1)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Taille	O(1)	O(1)
Suivant(s)	O(1)	O(n)
Distance	O(n)	O(n)

Consommation mémoire — structure vide : 2 pointeurs + 1 taille (`size_t`) ; par élément : 2 pointeurs + `sizeof(élément)`

`splice` est de complexité O(1).

4.2.3 Double ended queue (`std::deque`)

Consommation mémoire — structure vide : 6 pointeurs

Insertion au début :

- Calculer les indices `i_chunk` et `i_map` de la position précédente à celle d'insertion
- Si besoin, allouer un nouveau `chunk`
- Si besoin, augmenter la capacité de la `map`
- Écrire dans `map` à l'index `iMap(i)` et `iChunk(i)`
- Mettre à jour les champs `chunkDébut`, `mapDébut` et `taille`

Algorithme 34 : Fonctions d'indices

```
Fonction iChunk(i) :
    retourner (i + chunkDébut) % chunkCap

Fonction iMap(i) :
    j ← (i + chunkDébut)/chunkCap
    retourner (j + mapDébut) % mapCap
```

Table 7 : Complexités des opérations de `std::deque`

Opération	en moyenne	au pire
Consultation	O(1)	O(1)
Insertion en fin	O(1)	O(c+n/c)
... au milieu	O(min(i, n-i))	O(c+n/c)
... au début	O(1)	O(c+n/c)
Suppression en fin	O(1)	O(1)
... au milieu	O(min(i, n-i))	O(min(i, n-i))
... au début	O(1)	O(1)
Taille	O(1)	O(1)
Suivant(s)	-	-
Distance	-	-

4.3 Tas (`std::heap`)

- enfants ≤ parents
- Insertion : metrre à la fin et remonter
- Suppression (racine) : swap avec le dernier, descente de la nouvelle racine
- Créer tas : rétablir la condition depuis les feuilles jusqu'à la racine
- Tri par tas : suppression (racine) n fois
- `push_heap` place au bon endroit le dernier élément
- `pop_heap` fait une suppression

Algorithme 35 : Indices et parentés

```
Fonction parent(i) :
    retourner (i − 1)//2
Fonction e1(i) :
    retourner 2 * i + 1
Fonction e2(i) :
    retourner 2 * i + 2
```

Algorithme 36 : Remontée d'éléments

```
Fonction remonter(t, i) :
    tant que i > 0 et t[parent(i)] < t[i] faire
        échanger t[i], t[parent(i)]
        i ← parent(i)
```

Algorithme 37 : Descente d'éléments

```
Fonction pge(t, i) :
    si e2(i) < t.taille et t[e2(i)] > t[e1(i)]
      alors
        retourner e2(i)
    sinon
        retourner e1(i)
Fonction descendre(t, i) :
    tant que e1(i) < t.taille et
      t[pge(t, i)] > t[i] faire
        échanger t[pge(t, i)], t[i]
        i ← pge(t, i)
```

Algorithme 38 : Créer un tas

```
Fonction créerTas(t) :
    p ← parent(t.taille - 1)
    pour tous i ∈ [p, 0] faire      // i entier
        descendre(t, i)
```

Algorithme 39 : Tri par tas

```
Fonction triParTas(t) :
    n ← t.taille
    créerTas(t)
    pour tous k ∈ [n − 1, 1] faire // k entier
        échanger t[k], t[0]
        descendre(t, 0, k)
```

Table 8 : Complexités des opérations sur les tas

Opération	en moyenne	au pire
`make_heap`	O(n)	O(n)
`push_heap`	O(log(n))	O(log(n))
`pop_heap`	O(log(n))	O(log(n))
`sort_heap`	O(nlog(n)	O(nlog(n)

4.4 Pile et files

4.4.1 Pile (`std::stack`)

- Utile pour LIFO (last-in, first-out)
- Empiler (`push`), dépiler (`pop`), sommet (`top`)
- Utilise `std::deque` comme container

4.4.2 File (`std::queue`)

- Utile pour FIFO (last-in, first-out)
- Insérer (`push`), supprimer (`pop`), premier (`front`), derner (`back`)
- Utilise `std::deque` comme container

4.4.3 File de priorité (`std::priority_queue`)

- Utilisé avec un tas
- Insérer (`push`), plus important (`top`), moins important (`pop`)
- Utilise `std::vector` comme container

5 Arbres

5.1 Arbres génériques

Algorithme 40 : Structure d'un noeud d'un arbre

```
Structure Noeud :
    tiquette   // valeur variable quelconque
    ainé*                     // ptr sur Noeud
    puiné*                    // ptr sur Noeud
    parent*                   // ptr sur Noeud
```

5.1.1 Parcours

Le parcours en pré-ordre est utile pour copier un arbre. Le parcours en post-ordre est utile pour détruire un arbre.

Algorithme 41 : Parcours en profondeur (pré-ordre)

```
Fonction profondeur(racine, fn) :
    si racine = ∅ alors retourner
    fn(racine)
    pour tous e ∈ racine faire
        profondeur(e, fn)
```

Algorithme 42 : Parcours post-ordonné

```
Fonction postOrdre(racine, fn) :
    si racine = ∅ alors retourner
    pour tous e ∈ racine faire
        postOrdre(e, fn)
    fn(racine)
```

Algorithme 43 : Parcours en largeur

```
Fonction largeur(racine, fn) :
    q ← File FIFO         // p.ex: std::queue
    pousser racine dans q
    tant que q ≠ ∅ faire
        racine ← sommet de q
        fn(racine)
        pour tous e ∈ racine faire
            pousser e dans q
```

5.2 Arbres binaires

Arbres de degré 2, dont les enfants sont explicitement à gauche ou à droite, il n'a pas de liaison entre les noeuds d'un même niveau.

Un tas est un arbre binaire complet.

Algorithme 44 : Structure d'un noeud d'un arbre binaire

```
Structure Noeud :
    tiquette    // valeur variable quelconque
    gauche*                 // ptr sur Noeud
    droite*                 // ptr sur Noeud
    parent*       // ptr optionnel sur Noeud
```

5.2.1 Parcours

Algorithme 45 : Parcours en profondeur (pré-ordre)

```
Fonction profondeur(racine, fn) :
    si racine = ∅ alors retourner
    fn(racine)
    profondeur(racine.gauche, fn)
    profondeur(racine.droit, fn)
```

Algorithme 46 : Parcours post-ordonné

```
Fonction postOrdre(racine, fn) :
    si racine = ∅ alors retourner
    postOrdre(racine.gauche, fn)
    postOrdre(racine.droit, fn)
    fn(racine)
```

Algorithme 47 : Parcours symétrique

```
Fonction symétrique(racine, fn) :
    si racine = ∅ alors retourner
    symétrique(racine.gauche, fn)
    fn(racine)
    symétrique(racine.droit, fn)
```

5.2.2 Notations d'expressions

Algorithme 48 : Notation préfixe

```
Fonction préfixe(racine, fn) :
    si racine = ∅ alors retourner
    afficher racine.etiquette
    préfixe(racine.gauche, fn)
    préfixe(racine.droit, fn)
```

Algorithme 49 : Notation postfixe

```
Fonction postfixe(racine, fn) :
    si racine = ∅ alors retourner
    postfixe(racine.gauche, fn)
    postfixe(racine.droit, fn)
    afficher racine.etiquette
```

Algorithme 50 : Notation infixe

```
Fonction infixe(racine, fn) :
  si racine = ∅ alors retourner
  si racine est un noeud interne alors
    afficher "("
    infixe(racine.gauche, fn)
    afficher racine.etiquette
    infixe(racine.droit, fn)
    afficher ")"
  sinon
    afficher racine.etiquette
```

5.3 Arbres binaires de recherche

Clés uniques et $g.clé < r.clé < d.clé$.

Algorithme 51 : Structure d'un noeud d'un arbre binaire de recherche

```
Structure Noeud :
  clé        // valeur unique d'identification
  gauche*                     // ptr sur Noeud
  droite*                     // ptr sur Noeud
  valeur  // valeur quelconque optionnelle
  parent*         // ptr optionnel sur Noeud
  taille            // entier long optionnel
  hauteur          // entier court optionnel
  équilibre      // entier booléen optionnel
```

Algorithme 52 : Rechercher une valeur

```
Fonction chercher(racine, k) :
  si racine = ∅ alors
    retourner                  // introuvable
  sinon si k = racine.clé alors
    retourner racine // valeur demandée
  sinon si k < racine.clé alors
    chercher(racine.gauche, k)
  sinon
    chercher(racine.droit, k)
```

Algorithme 53 : Insérer une valeur

```
Fonction insérer(&racine, k) :
  si racine = ∅ alors
    r ← nouveau Noeud avec la clé k
  sinon si k = racine.clé alors
    retourner          // valeur présente
  sinon si k < racine.clé alors
    insérer(racine.gauche, k)
  sinon
    insérer(racine.droit, k)
```

Algorithme 54 : Parcours de recherche

```
Fonction croissant(racine, fn) :
  si racine = ∅ alors retourner
  croissant(racine.gauche, fn)
  fn(racine)
  croissant(racine.droit, fn)
Fonction décroissant(racine, fn) :
  si racine = ∅ alors retourner
  décroissant(racine.droit, fn)
  fn(racine)
  décroissant(racine.gauche, fn)
```

5.3.1 Suppression de valeurs

Algorithme 55 : Rechercher le minimum

```
Fonction min(racine) :
  si racine.gauche = ∅ alors
    retourner r
  sinon
    min(racine.gauche)
```

Algorithme 56 : Supprimer le minimum

```
Fonction supprimerMin(&racine) :
  si racine = ∅ alors alerter
  si racine.gauche ≠ ∅ alors
    supprimerMin(racine.gauche)
  sinon
    d ← racine.droit
    détruire racine
    racine ← d
```

Algorithme 57 : Supprimer la clé k (Hibbard)

```
Fonction sortirMin(&racine) :
  si racine.gauche ≠ ∅ alors
    retourner sortirMin(racine.gauche)
  sinon
    tmp ← r
    racine ← racine.droit
    retourner tmp
Fonction supprimer(&racine, k) :
  si racine = ∅ alors retourner
  sinon si k < racine.clé alors
    supprimer(r.gauche, k)
  sinon si k > racine.clé alors
    supprimer(r.droit, k)
  sinon
    tmp ← racine
    si racine.gauche = ∅ alors
      racine ← racine.droit
    sinon si racine.droit = ∅ alors
      racine ← racine.gauche
    sinon
      m ← sortirMin(racine.droit)
      m.droit ← racine.droit
      m.gauche ← racine.gauche
      racine ← m
    détruire tmp
```

5.3.2 Complexités

Table 9 : Complexités des arbres de taille n et hauteur h

Opération	en moyenne	au pire
Parcours	O(n)	O(n)
Itération sur tout l'arbre	O(n)	O(n)
...suivant	O(1)	O(h)
Recherche, insertion, suppression, min, max	O(p)	O(p)

5.3.3 Rang, hauteur

Rang : nombre d'éléments de clé inférieur à k.
Hauteur : nombre max d'enfants.

5.3.4 Équilibrage

Algorithme 58 : Calcul de l'équilibre (OK si entre -1 et 1)

```
Fonction équilibre(racine) :
  si racine = ∅ alors
    retourner 0
  sinon
    retourner hauteur(racine.gauche) -
      hauteur(r.droit)
```

Algorithme 59 : Équilibrage à la demande O(n)

```
Fonction linéariser(racine, &lien,
 &noeuds) :
  si racine = ∅ alors retourner
  linéariser(racine.droit, lien, noeuds)
  racine.droit ← lien
  lien ← racine
  n ← n + 1
  linéariser(racine.gauche, lien, noeuds)
  racine.gauche ← ∅
Fonction arboriser(&lien, noeuds) :
  si noeuds = 0 alors retourner
  rg ← arboriser(lien, (n − 1)//2)
  r ← lien
  r.gauche ← rg
  lien ← lien.droit
  r.droit ← arboriser(lien, n//2)
  retourner r
```

Algorithme 60 : Équilibrage à la demande O(n) (suite)

```
Fonction équilibre(racine) :
  lien ← ∅
  noeuds ← 0
  linéariser(racine, lien, noeuds)
  retourner arboriser(lien, noeuds)
```

5.4 Arbres AVL

Insertion : calcul de hauteur (`calculerHauteur`) après la création d'une nouvelle feuille.

5.4.1 Rotations

Algorithme 61 : Rotations

```
Fonction rotationGauche(&r) :
  t ← r.droit
  r.droit ← t.gauche
  t.gauche ← r
  r ← t
  calculerHauteur(r.gauche)
  calculerHauteur(r)
Fonction rotationDroite(&r) :
  t ← r.gauche
  r.gauche ← t.droit
  t.droit ← r
  r ← t
  calculerHauteur(r.droit)
  calculerHauteur(r)
```

5.4.2 Équilibrage

Technique : à partir du déséquilibre, il faut regarder si le signe est le même. Le cas échéant : double rotation. Il faut toujours commencer au déséquilibre le plus proche des feuilles.

Algorithme 62 : Rétablissement de l'équilibre

```
Fonction rétablirÉquilibre(&r) :
  si r = ∅ alors retourner
  si équilibre(r) < −1 alors         // drte
    si équilibre(r.droit) > 0 alors
      rotationDroite(r.droit)
    rotationGauche(r)
  sinon si équilibre(r) > 1 alors   // gche
    si équilibre(r.gauche) < 0 alors
      rotationGauche(r.gauche)
    rotationDroite(r)
  sinon
    calculerHauteur(r)
```

5.5 `std::set`

Consommation mémoire — structure vide : 2 pointeurs + 1 taille (`size_t`) ; par élément : 3 pointeurs + `sizeof(élément)`

Table 10 : Complexités des opérations de `std::set`

Opération	en moyenne	au pire
Consultation	-	-
Insertion en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Suppression en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Taille	O(1)	O(1)
Suivant(s)	O(n)	O(n)
Distance	-	-

5.6 `std::map`

Consommation mémoire — structure vide : 2 pointeurs + 1 taille (`size_t`) ; par élément : 3 pointeurs + `sizeof(clé)` + `sizeof(valeur)`

Table 11 : Complexités des opérations de `std::map`

Opération	en moyenne	au pire
Consultation	O(log(n))	O(log(n))
Insertion en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Suppression en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Taille	O(1)	O(1)
Suivant(s)	-	-
Distance	-	-

6 Graphes

6.1 Définitions

Graphe non orienté — $G = (V, E)$, V ensemble de sommets, E ensemble d'arêtes, une fonction d'incidence qui associe une paire de sommets à une arête.
Graphe orienté — $G = (V, E)$, V ensemble de sommets, E ensembles d'arcs directionnels, une fonction d'incidence qui relie les extrémités (initiales et finales) d'un arc.
Sommets adjacents — deux sommets sont adjacents s'ils sont reliés par une arête. Ils sont alors **incidents** à l'arête.
Graphe sous-jacent — on remplace tous les arcs d'un graphe orienté par des arêtes, ce qui donne un graphe non orienté.
Graphe simple — graphe sans **boucle** ni **arête/arc multiple**. Opposé : **multigraphe**.
Degré — Nombre d'arêtes ou arcs incidents au sommet. **Sommet pendant** → degré = 1.
Demi-degré entrant/sortant — Graphe orienté. Nombres d'arcs dont le sommet est l'extrémité initiale/finale, respectivement.
Chaine/Chemin — (non orienté/orienté). Suite de sommets reliés par des arêtes/arcs. **Élémentaire** si aucun sommet répété. **Simple** si aucune arête n'est répétée.
Cycle/Circuit — (non orienté/orienté). Suite de sommets reliés par des arêtes/arcs qui commencent et se terminent au même sommet. **Élémentaire** si aucun sommet répété. **Simple** si aucune arête n'est répétée. Graphe **acyclique** s'il est sans cycle simple.

6.2 Représentation

Matrice d'adjacence — Représentation des n éléments par une matrice $n \times n$. Chaque élément de la matrice indique le nombre d'arêtes/arcs reliant l'indice de la ligne à l'indice de la colonne. Efficace si le nombre d'arêtes est proche du carré du nombre de sommets.
Listes de sommets — Stockage des relations par liste de sommets et pour chaque sommet une liste d'indices d'autres sommets avec lesquels ils partagent une arête/arc. Dans le cas d'un graphe orienté, la liste stocke les successeurs.

6.3 Parcours

Pour parcourir un graphe, les sommets déjà atteints par le parcours doivent être marqués pour éviter de rester bloqué dans un cycle.
Le parcours depuis un sommet n'atteint pas nécessairement tous les sommets du graphe.

6.3.1 Profondeur

Depth first search (DFS) — méthode récursive qui s'appelle pour tous les sommets adjacents.
Technique : lors de l'appel des fonctions **pre** et **post**, ajouter une pastille (à gauche et à droite, respectivement) dans le graphe.

Algorithme 63 : Parcours en profondeur

```
// Précondition: sommets non marqués
Fonction profondeur(sommet, pre, post) :
  pre()
  marquer sommet                   // visité
  pour tous w adjacent à sommet faire
    si w n'est pas visité alors
      profondeur(w, pre, post)
  post()
```

Algorithme 64 : Parcours en profondeur global

```
Fonction profondeurGlobal(graphe) :
  marquer tous les sommets  // non visité
  pour tous s sommet de graphe faire
    si s n'est pas visité alors
      profondeur(w, (any), (any))
```

6.3.2 Largeur

Breadth first search (BFS) — utilisation d'une file FIFO qui parcours d'abord tous les sommets adjacents avant ceux à distance 2, puis 3, etc.

Technique : marquer d'une pastille à gauche les sommets visités lors de l'ajout dans la file (dans les deux cas), marquer d'une pastille en bas les sommets où l'action a été appelée, et marquer d'une pastille à droite les sommets une fois les sommets adjacents traités. Garder à côté une trace de la file.

Algorithme 65 : Parcours en largeur

```
Fonction largeur(sommet, action) :
  Q ← file FIFO
  pousser sommet dans Q
  marquer sommet                   // visité
  tant que Q n'est pas vide faire
    v ← sommet de Q
    action(v)
    pour tous w adjacent à v faire
      si w n'est pas marqué alors
        pousser w dans Q
        marquer w                  // visité
```

6.3.3 Complexités

Avec n sommets et m arêtes :

— Matrice d'adjacence : $O(n^2)$

— Listes d'adjacences : $O(n + m)$

6.3.4 Application des parcours

Parents sera un tableau indiquant pour chaque sommet le sommet parent en direction du sommet de départ.

Algorithme 66 : Récupération des parents en largeur

```
Fonction parentsEnLargeur(sommet) :
  Parents ← tableau initialisé à −1
  Q ← file FIFO
  pousser sommet dans Q
  Parents[sommet] ← sommet
  tant que Q n'est pas vide faire
    v ← sommet de Q
    pour tous w adjacent à v faire
      si Parents[w] = −1 alors
        pousser w dans Q
        Parents[w] ← v
  retourner Parents
```

Algorithme 67 : Recherche de chaîne

```
Fonction chaine(parents, sommet) :
    chaine ← chaine vide
    si parents[sommet] = -1 alors
        retourner chaine

    tant que parents[sommet] ≠sommet faire
        pousser sommet dans chaine
        sommet ← parents[sommet]
    retourner chaine
```

Algorithme 68 : Parents de plusieurs sommets

```
Fonction parentsEnLargeur(sommets) :
    Parents ← tableau initialisé à −1
    Q ← file FIFO
    pour tous v sommet de sommets faire
        pousser v dans Q
        Parents[v] ← v
    tant que Q n'est pas vide faire
        v ← sommet de Q
        pour tous w adjacent à v faire
            si Parents[w] = −1 alors
                pousser w dans Q
                Parents[w] ← v
    retourner Parents
```

L'algorithme 68 permet de récupérer pour chaque sommet du graphe le sommet le plus proche des trois sommets donnés. Il est ensuite possible d'utiliser l'algorithme 67 pour retrouver le chemin le plus rapide vers ce parent.

6.4 Composantes connexes

Méthode en $O(1)$ de déterminer si un chemin entre deux sommets est possible. Requiers un calcul au préalable, de préférence lors de l'édition du graphe.

Algorithme 69 : Calcul des composantes connexes

```
Fonction composantesConnexes(G(V, E)) :
    id ← 0
    CC ← tableau initialisé à −1
    pour tous sommet v de G faire
        si CC[v] = −1 alors
            profondeur(v, CC[w] ← id)
            id ← id + 1
    retourner CC
```

6.5 Dijkstra

Avec un parcours en largeur avec parents, on peut calculer la chaine / le chemin le plus court « au sens du nombre d'arêtes / arcs inclus ».

Quand le graphe est pondéré, la métrique pertinente est différente : on cherche le chemin le plus court « au sens de la somme des poids le long du chemin ».

6.5.1 Relâchement d'un arc

Chaque sommet reçoit deux valeurs :

— `distTo`(v) qui est la distance entre v et le sommet de départ v_0. Initialisé à ∞, sauf pour $v_0 = 0$.

— `edgeTo`(v) qui est le dernier arc du chemin le plus court allant de v_0 à v.

Algorithme 70 : Relâcher un arc

```
Fonction relâcherArc(v → w) :
    d ← distTo(v) + poids(v → w)
    si d < distTo(w) alors
        distTo(w) ← d
        edgeTo(w) ← (v → w)
```

6.5.2 Parcours de Dijkstra

L'algorithme de Dijkstra consiste à relaxer les arcs en parcourant le graphe par ordre de distance croissante au sommet de départ.
Pour cela, on remplace la file FIFO du parcours en largeur par une file de priorité où le sommet le plus prioritaire a la plus petite valeur `distTo`(v) parmi les sommets non traités.

Algorithme 71 : Parcours de Dijkstra

```
Fonction dijkstra(G(V, E), v0) :
    Q ← queue de priorité
    distTo ← tableau de |V| éléments
    edgeTo ← tableau de |V| éléments
    distTo(v0) ← 0
    pousser (v0, 0) dans Q
    pour tous sommet v dans G faire
        si v ≠ v0 alors
            distTo(v) ← ∞
            edgeTo(v) ← ?
    tant que Q n'est pas vide faire
        v ← sommet de Q
        pour tous arc v → w depuis v faire
            d ← distTo(v) + poids(v → w)
            si d < distTo(w) alors
                distTo(w) ← d
                edgeTo(w) ← (v → w)
                pousser (w, −distTo(w)) dans
                  Q  // édition si nécessaire
    retourner (distTo, edgeTo)
```

6.6 Tri topologique

Fonctionne uniquement sur les DAG (graphe orienté acyclique). Inverse du parcours post-ordre (DFS).
Ordre pour le DFS arbitraire, il reste préférable de prendre les valeurs dans un ordre croissant.

6.6.1 Détection de cycles

Principe : Parcours en profondeur où l'on garde trace des sommets présents dans la pile de récursion.
Si le parcours nous amène à atteindre un sommet présent dans cette pile, tous les éléments de la pile entre ce sommet (vertex) et le sommet (top) de la pile forment un circuit.

Algorithme 72 : Détection de circuit

```
// Préconditions : marqués et empilés des
   tableau de booléens initialement à faux,
   cycleExistant un booléen faux.
Fonction détectionCycle(sommet v) :
    marqués[v] ← vrai
    empilés[v] ← vrai
    pour tous w adjacent à v faire
        si cycleExistant alors retourner
        sinon si marqués[w] = faux alors
            détectionCycle(w)
        sinon si empilés[w] = vrai alors
            cycleExistant = vrai
    empilés[v] ← faux
```

6.7 Composantes fortement connexes

Les sommets v et w sont fortement connectés s'il existe un chemin (orienté) de v à w et de w à v.

6.7.1 Algorithme de Kosaraju-Sharir

1. Calculer le post ordre inverse du parcours en profondeur pour le graphe inverse de G
2. Calculer les composantes connexes par parcours en profondeur DFS sur G dans cet ordre

7 Annexes

Listing 1 : Tris avec Iterator

```
template <class Iterator>
inline void BubbleSort(Iterator begin, Iterator end) {
    for (Iterator i = begin; i != end; ++i)
        for (Iterator j = begin; j < i; ++j)
            if (*i < *j)
                std::iter_swap(i, j);
}

template <class Iterator>
void InsertionSort(Iterator begin, Iterator end) {
    std::iter_swap(begin, std::min_element(begin, end));
    for (Iterator b = begin; ++b < end; begin = b)
        for (Iterator c = b; *c < *begin; --c, --begin)
            std::iter_swap(begin, c);
}

template <class Iterator>
inline void SelectionSort(Iterator begin, Iterator end) {
    for (Iterator i = begin; i != end; ++i)
        std::iter_swap(i, std::min_element(i, end));
}

template <class Iterator>
inline void QuickSort(Iterator begin, Iterator end) {
    if (end <= begin) return;
    Iterator pivot = begin, middle = begin + 1;
    for (Iterator i = begin + 1; i < end; ++i) {
        if (*i < *pivot) {
            std::iter_swap(i, middle);
            ++middle;
        }
    }
    std::iter_swap(begin, middle - 1);
    QuickSort(begin, middle - 1);
    QuickSort(middle, end);
}

template <class Iterator>
inline void MergeSort(Iterator begin, Iterator end) {
    if (end <= begin + 1) return;
    Iterator middle = begin + (end - begin) / 2;
    MergeSort(begin, middle);
    MergeSort(middle, end);
    std::inplace_merge(begin, middle, end);
}

template <class Iterator>
inline void HeapSort(Iterator begin, Iterator end) {
    while (begin != end)
        std::pop_heap(begin, end--);
}
```

Listing 2 : Tri par base et tri comptage

```
/**
* Sort a given input range using the counting sort algorithm.
 * @tparam Iterator Iterator type of the input range.
 * @tparam Fn Function type to compare two elements.
 * @param first First element of the input range.
 * @param last Last element of the input range.
 * @param output_first First element of the output range.
 * @param index_fn Function to get the index of an element.
 * @param N Number of different categories
 */
template<typename Iterator, typename Fn>
void tri_comptage(Iterator first, Iterator last,
                  Iterator output_first, Fn index_fn, size_t N) {
    // T will be the type of the elements of the input sequence
    using T = typename Iterator::value_type;

    // Create the temporary count vector
    std::vector<unsigned long long> counters(N, 0);
    std::for_each(first, last, [&](const T &x) { ++counters[index_fn(x)]; });

    // Recreate count tab to set indexes rather than count
    size_t idx = 0;
    for (size_t i = 0; i < N; ++i) {
        size_t tmp = counters[i];
        counters[i] = (T) idx;
        idx += tmp;
    }

    // Build the output vector
    for (auto it = first; it != last; ++it) {
        auto &count = counters[index_fn(*it)];
        *std::next(output_first, (long long) count) = *it;
        ++count;
    }
}

/**
 * Sort a given input range using the radis sort algorithm.
 * @tparam Iterator Iterator type of the input range.
 * @tparam NBITS Number of bits used to represent the categories.
 * @param first First element of the input range.
 * @param last Last element of the input range.
 */
template<typename Iterator, size_t NBITS>
void tri_par_base(Iterator first, Iterator last) {
    using T = typename Iterator::value_type;
    static_assert(std::is_unsigned<T>::value);

    // Get the size of vector, assumed last > first
    auto size = (size_t) std::distance(first, last);
    // Temporary vector for countsort
    std::vector<T> countsortOutput(size, 0);
    // Max value from the vector to sort
    T maxValue = *std::max_element(first, last);

    size_t pos = 0;
    // number of categories for countsort
    const T keyNumber = maxValue + 1;

    // Apply the counting sort
    while (maxValue) {
        auto fn = SomeBits<unsigned long long>(NBITS, pos++);
        maxValue >>= NBITS;
        tri_comptage<>(first, last, countsortOutput.begin(), fn, keyNumber);
        std::swap_ranges(first, last, countsortOutput.begin());
    }
}
```