

1 Introduction aux Algorithmes

Definitions :

Algorithme : Suite finie d'operations pour resoudre un probleme

Structure de donnees : Organisation des donnees pour un acces/stockage efficace

Complexite : Mesure des ressources (temps/espace) en fonction de l'entree

2 Analyse de Complexite

Notations Asymptotiques :

$\mathcal{O}(f(n))$: Borne superieure (pire cas)

$\Omega(f(n))$: Borne inferieure (meilleur cas)

$\Theta(f(n))$: Borne serree (cas moyen)

$o(f(n))$: Borne superieure stricte

$\omega(f(n))$: Borne inferieure stricte

Complexites Courantes :

$\mathcal{O}(1)$: Acces tableau, operations elementaires

$\mathcal{O}(\log n)$: Recherche binaire, operations ABR/AVL

$\mathcal{O}(n)$: Parcours sequentiel, recherche liste non trie

$\mathcal{O}(n \log n)$: Meilleurs tris par comparaison

$\mathcal{O}(n^2)$: Tris quadratiques, parcours matrice

$\mathcal{O}(2^n)$: Problemes exponentiels (sous-ensembles)

$\mathcal{O}(n!)$: Problemes factoriels (permutations)

3 Analyse Recursive

Theoreme Maitre :

Pour $T(n) = aT(n/b) + f(n)$ ou $a \geq 1, b > 1$:

Si $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$: $T(n) = \Theta(n^{\log_b a})$

Si $f(n) = \Theta(n^{\log_b a} \log^k n)$: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ et $af(n/b) \leq cf(n)$: $T(n) = \Theta(f(n))$

Exemples :

Recherche binaire : $T(n) = T(n/2) + \mathcal{O}(1) = \mathcal{O}(\log n)$

Tri fusion : $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$

Tri rapide (moyen) : $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$

Tri rapide (pire cas) : $T(n) = T(n-1) + \mathcal{O}(n) = \mathcal{O}(n^2)$

4 Structures de Base

Tableau : Acces $\mathcal{O}(1)$, Insertion/Suppression $\mathcal{O}(n)$

Liste chaine : Acces $\mathcal{O}(n)$, Insertion/Suppression $\mathcal{O}(1)$ (si pointeur)

Pile : LIFO, operations $\mathcal{O}(1)$

File : FIFO, operations $\mathcal{O}(1)$ (avec deux piles en $\mathcal{O}(1)$ amorti)

Tas : Insertion/Extraction en $\mathcal{O}(\log n)$

5 Conseils d'Analyse

Toujours identifier le pire/meilleur cas moyen

Verifier les boucles imbriquees (multiplication des complexites)

Attention aux operations non elementaires (ex : tri dans une boucle)

Pour les recursions, ecrire l'equation de recurrence

Penser a l'initialisation et aux cas de base

Attention aux operations couteuses dans les boucles

6 Recursive

Definition : Algorithme qui s'appelle lui-meme avec des entrees plus petites.

Elements cles :

Cas de base (condition d'arret)

Appel recursif (parametres convergents)

Convergence vers le cas de base

Exemple - Factorielle :

```
1 def factorielle(n):
2     if n <= 1: return 1 % Cas de base
3     return n * factorielle(n-1) % Appel recursif
```

Exemple - Fibonacci :

```
1 def fib(n):
2     if n <= 1: return n % Cas de base
3     return fib(n-1) + fib(n-2) % Double appel recursif
```

Complexites :

Factorielle : $\mathcal{O}(n)$ temps, $\mathcal{O}(n)$ pile

Fibonacci naif : $\mathcal{O}(2^n)$ temps, $\mathcal{O}(n)$ pile

Fibonacci iteratif : $\mathcal{O}(n)$ temps, $\mathcal{O}(1)$ espace

7 Diviser pour Regner

Principe en 3 etapes :

- **Diviser** : Decouper en sous-problemes

- **Resoudre** : Appels recursifs

- **Combiner** : Fusionner les solutions

Relation de recurrence :

$$T(n) = aT(n/b) + f(n)$$

a : nb sous-problèmes

n/b : taille des sous-problèmes

$f(n)$: coût division + combinaison

8 Algorithmes Classiques

8.1 Tri Fusion

```
1 def tri_fusion(tab):
2     if len(tab) <= 1: return tab
3     m = len(tab) // 2
4     return fusion(tri_fusion(tab[:m]), tri_fusion(tab[m:]))
5
6 def fusion(g, d):
7     res, i, j = [], 0, 0
8     while i < len(g) and j < len(d):
9         if g[i] <= d[j]:
10            res.append(g[i])
11            i += 1
12        else:
13            res.append(d[j])
14            j += 1
15    return res + g[i:] + d[j:]
```

Complexite :

Temps : $\mathcal{O}(n \log n)$ (tous cas)

Espace : $\mathcal{O}(n)$ (non en place)

Stable : Oui

8.2 Identification des Algorithmes de Tri

Caracteristiques cles :

- **Bubble** : Echanges locaux, bulles remontent
- **Insertion** : Construction incrementale trie
- **Selection** : Recherche du minimum a chaque iteration
- **Quick** : Partition autour d'un pivot
- **Merge** : Division puis fusion ordonnee
- **Radix** : Tri par chiffres/caracteres

8.3 Recherche Dichotomique

```
1 def recherche_dichotomique(tab, cible):
2     g, d = 0, len(tab) - 1
3     while g <= d:
4         m = (g + d) // 2
5         if tab[m] == cible: return m
6         if tab[m] < cible: g = m + 1
7         else: d = m - 1
8     return -1
```

Complexite : $\mathcal{O}(\log n)$ temps, $\mathcal{O}(1)$ espace

8.4 Tours de Hanoi

```
1 def hanoi(n, source, cible, aux):
2     if n > 0:
3         hanoi(n-1, source, aux, cible) % Deplacer n-1 disques vers l'auxiliaire
4         print(f"{source} -> {cible}") % Deplacer le disque restant
5         hanoi(n-1, aux, cible, source) % Deplacer les n-1 disques vers la cible
```

Complexite :

Temps : $\mathcal{O}(2^n)$

Espace : $\mathcal{O}(n)$ (pile d'appels)

Nombre de coups : $2^n - 1$

8.5 Minimax avec elagage alpha-beta

```
1 def alphabeta(etat, profondeur, alpha, beta, est_maximisant):
2     if est_terminal(etat) or profondeur == 0:
3         return evaluation(etat)
4
5     if est_maximisant:
6         max_eval = -infini
7         for coup in coups_possibles(etat):
8             eval = alphabeta(jouer_coup(etat, coup), profondeur-1, alpha, beta, False)
9             max_eval = max(max_eval, eval)
10            alpha = max(alpha, eval)
11            if beta <= alpha:
12                break % Elagage
13        return max_eval
14    else:
15        min_eval = +infini
16        for coup in coups_possibles(etat):
17            eval = alphabeta(jouer_coup(etat, coup), profondeur-1, alpha, beta, True)
18            min_eval = min(min_eval, eval)
19            beta = min(beta, eval)
20            if beta <= alpha:
21                break % Elagage
22        return min_eval
```

9 Algorithmes STL - Tri et Recherche

9.1 STL Sorting Algorithms

Algorithm	Complexite	Stable	Usage
std::sort	$\mathcal{O}(n \log n)$	Non	Tri general rapide
std::stable_sort	$\mathcal{O}(n \log n)$	Oui	Preserve ordre elements egaux
std::partial_sort	$\mathcal{O}(n \log k)$	Non	k plus petits elements
std::nth_element	$\mathcal{O}(n)$ moyen	Non	k-ieme element

```
1 vector<int> v = {64, 34, 25, 12, 22, 11, 90};
2
3 // Tri standard (rapide mais non stable)
4 sort(v.begin(), v.end());
5
6 // Tri stable (preserve ordre des elements egaux)
7 stable_sort(v.begin(), v.end());
8
9 // Tri partiel : 3 plus petits elements en debut
10 partial_sort(v.begin(), v.begin() + 3, v.end());
11
12 // k-ieme element (ici 3eme plus petit)
13 nth_element(v.begin(), v.begin() + 2, v.end());
```

9.2 STL Binary Search

```
1 vector<int> v = {1, 2, 4, 4, 4, 6, 8}; // DOIT etre trie !
2
3 // lower_bound : premier element >= target
4 auto it1 = lower_bound(v.begin(), v.end(), 4); // pointe vers 1er 4
5
6 // upper_bound : premier element > target
7 auto it2 = upper_bound(v.begin(), v.end(), 4); // pointe apres dernier 4
8
9 // equal_range : [lower_bound, upper_bound)
10 auto range = equal_range(v.begin(), v.end(), 4);
11
12 // Compter occurrences
13 int count = range.second - range.first; // ou distance(range.first, range.second)
14
15 // Binary search : true/false si element existe
16 bool exists = binary_search(v.begin(), v.end(), 4);
```

Complexites : $\mathcal{O}(\log n)$ pour tous ces algorithmes sur conteneur trie

9.3 Custom Comparators

```
1 struct Person { string name; int age; };
```

```
2 // 1. Fonction comparateur
3
4 bool byAge(const Person& a, const Person& b) {
5     return a.age < b.age;
6 }
7
8 // 2. Functor (classe avec operator())
9
10 struct ByName {
11     bool operator()(const Person& a, const Person& b) const {
12         return a.name < b.name;
13     };
14 };
15
16 // 3. Lambda (C++11+)
17 auto byAgeDesc = [](const Person& a, const Person& b) {
18     return a.age > b.age; // Ordre decroissant
19 };
20
21 vector<Person> people = {{ "Alice", 25}, {"Bob", 20}, {"Charlie", 30}};
22
23 // Utilisation des comparateurs
24 sort(people.begin(), people.end(), byAge); // Fonction
25 sort(people.begin(), people.end(), ByName()); // Functor
26 sort(people.begin(), people.end(), byAgeDesc); // Lambda
27
28 // Avec lower bound aussi
29 auto it = lower_bound(people.begin(), people.end(),
30                       Person("", 25), byAge);
```

10 Conseils Pratiques

- Toujours definir un cas de base clair
- Verifier que les appels convergent vers le cas de base
- Attention aux appels recursifs multiples (ex : Fibonacci naif)
- Privilégier l'itératif quand c'est possible (meilleure complexite spatiale)
- Penser a la memoisation pour eviter les calculs redondants

11 Comparaison des Tris

Algorithme	Meilleur	Pire	Stable	Espace
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Non	$\mathcal{O}(1)$
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Oui	$\mathcal{O}(1)$
Rapide	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	Non	$\mathcal{O}(\log n)$
Fusion	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Oui	$\mathcal{O}(n)$
Tas	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Non	$\mathcal{O}(1)$
Denombrement	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	Oui	$\mathcal{O}(n + k)$
Base	$\mathcal{O}(d(n + k))$	$\mathcal{O}(d(n + k))$	Oui*	$\mathcal{O}(n + k)$

12 Tris Comparaison

12.1 Par Selection

```
1 def tri_selection(tab):
2     for i in range(len(tab)-1):
3         min_idx = i
4         for j in range(i+1, len(tab)):
5             if tab[j] < tab[min_idx]:
6                 min_idx = j
7         tab[i], tab[min_idx] = tab[min_idx], tab[i]
```

Caracteristiques :

$\mathcal{O}(n^2)$ operations

$\mathcal{O}(1)$ espace

Non stable

Peu efficace en pratique

12.2 Par Insertion

```
1 def tri_insertion(tab):
2     for i in range(1, len(tab)):
3         cle, j = tab[i], i-1
4         while j >= 0 and cle < tab[j]:
5             tab[j+1] = tab[j]
6             j -= 1
7         tab[j+1] = cle
```

Caracteristiques :

$\mathcal{O}(n)$ (deja trie) a $\mathcal{O}(n^2)$

Stable et en place

Efficace pour petits tableaux ($n \leq 50$)

12.3 Tri Rapide (QuickSort)

```
1 from random import randint
2
3 def partition(tab, g, d):
4     pivot = tab[randint(g, d)]
5     i, j = g, d
6     while i <= j:
7         while tab[i] < pivot: i += 1
8         while tab[j] > pivot: j -= 1
9         if i <= j:
```

```
10         tab[i], tab[j] = tab[j], tab[i]
11         i += 1; j -= 1
12     return i
13
14 def tri_rapide(tab, g=0, d=None):
15     if d is None: d = len(tab)-1
16     if g < d:
17         p = partition(tab, g, d)
18         tri_rapide(tab, g, p-1)
19         tri_rapide(tab, p, d)
```

Caracteristiques :

$\mathcal{O}(n \log n)$ en moyenne

$\mathcal{O}(n^2)$ pire cas (rare)

$\mathcal{O}(\log n)$ pile (recursif)

Non stable

Optimisations :

Pivot aleatoire ou mediane de 3

Tri insertion pour $n < 20$

Itératif pour eviter la pile

13 Tris Avances

13.1 Tri Fusion

```
1 def fusion(g, d):
2     res, i, j = [], 0, 0
3     while i < len(g) and j < len(d):
4         if g[i] <= d[j]:
5             res.append(g[i]); i += 1
6         else:
7             res.append(d[j]); j += 1
8     return res + g[i:] + d[j:]
9
10 def tri_fusion(tab):
11     if len(tab) <= 1: return tab
12     m = len(tab) // 2
13     return fusion(tri_fusion(tab[:m]), tri_fusion(tab[m:]))
```

Caracteristiques :

• $\mathcal{O}(n \log n)$ dans tous les cas

• $\mathcal{O}(n)$ espace supplementaire

• Stable

• Ideal pour :

- Listes chainees
- Donnees externes
- Besoin de stabilite

13.2 Tri par Tas

```
1 def entasser(tab, n, i):
2     g, d, max_i = 2*i+1, 2*i+2, i
3     if g < n and tab[g] > tab[max_i]: max_i = g
4     if d < n and tab[d] > tab[max_i]: max_i = d
5     if max_i != i:
6         tab[i], tab[max_i] = tab[max_i], tab[i]
7         entasser(tab, n, max_i)
8
9 def tri_tas(tab):
10     n = len(tab)
11     for i in range(n//2-1, -1, -1): Constructionentasser(tab, n, i)for i in range(n-1, 0, -1):
12         Extraction
13         tab[0], tab[i] = tab[i], tab[0]
14         entasser(tab, i, 0)
```

Caracteristiques :

• $\mathcal{O}(n \log n)$ dans tous les cas

• $\mathcal{O}(1)$ espace (en place)

• Non stable

• Utilisations :

- Files de priorite
- Tri partiel (k elements)
- Contraintes memoire

14 Tris Lineaires

14.1 Tri par Denombrement

```
1 def tri_denombrement(tab, k):
2     compte = [0] * (k + 1)
3     for x in tab: compte[x] += 1
4     resultat, valeur = [], 0
5     for nb in compte:
6         resultat.extend([valeur] * nb)
7         valeur += 1
8     return resultat
```

Caracteristiques :

• $\mathcal{O}(n + k)$ temps/espace

• Stable

• Efficace si $k = \mathcal{O}(n)$

• Exemple : tri de lettres (k=26)

14.2 Tri par Base (Radix)

```
1 def tri_base(tab, base=10):
2     max_val = max(tab) if tab else 0
3     exp = 1
4     while max_val // exp > 0:
5         tab = tri_denombrement(
6             [(x // exp) % base for x in tab],
7             base-1)
8         exp *= base
9     return tab
```

Caracteristiques :

• $\mathcal{O}(d(n + k))$ (d = chiffres)

• Stable si tri stable utilise

• Ideal pour :

- Grands entiers
- Nombres avec peu de chiffres
- Donnees de meme longueur

15 Complexites des Operations Courantes

Structures de Donnees :

• Acces tableau : $\mathcal{O}(1)$

• Recherche liste : $\mathcal{O}(n)$

• Pile/File : $\mathcal{O}(1)$ (operations de base)

• Table de hachage : $\mathcal{O}(1)$ (moyen)

Operations sur Tableaux :

• Recherche sequentielle : $\mathcal{O}(n)$

• Recherche dichotomique : $\mathcal{O}(\log n)$

• Tri : $\mathcal{O}(n \log n)$ (optimal pour comparaisons)

Graphes :

• Parcours (BFS/DFS) : $\mathcal{O}(V + E)$

• Dijkstra : $\mathcal{O}(E + V \log V)$

• Tri topologique : $\mathcal{O}(V + E)$

Autres :

• Exponentiation rapide : $\mathcal{O}(\log n)$

• Fibonacci : $\mathcal{O}(n)$ (iteratif)

• Fibonacci avec memoisation : $\mathcal{O}(n)$

16 Quel Algorithme Choisir ?

Par taille de donnees :

• $n \leq 50$: Tri insertion

• $50 < n \leq 10^6$: Tri rapide

• $n > 10^6$: Tri fusion/Radix

Par caracteristiques :

• Presque trie : Tri insertion

• Donnees externes : Tri fusion

• Contraintes memoire : Tri par tas

• Plage limitee : Tri denombrement

Par besoins :

• Stable : Fusion/Insertion

• En place : Rapide/Tas

• Temps garanti : Fusion/Tas

• Cache-friendly : Rapide

En pratique :

• Python : Timsort (fusion + insertion)

• Java : Dual-pivot quicksort

• C++ : Introsort (rapide + tas)

17 Structures Lineaires

Structures de Donnees Lineaires

Organisation sequentielle des elements

Acces sequentiel ou direct selon l'implmentation

Operations de base : insertion, suppression, acces

17.1 Pile (LIFO)

Operations : push (empiler), pop (depiler), peek (sommet), isEmpty (estVide)

Complexite : $\mathcal{O}(1)$ pour toutes les operations

Utilisations :

Appels de fonction (pile d'execution)

Backtracking (parcours en profondeur)

Evaluation d'expressions (notation polonaise inverse)



```
1 class Pile:
2     def __init__(self):
3         self.contenu = []
4
5     def empiler(self, x):
6         """Ajoute un element au sommet de la pile"""
7         self.contenu.append(x)
8
9     def depiler(self):
10        """Retire et retourne l'element au sommet"""
11        return self.contenu.pop() if self.contenu else None
12
13    def sommet(self):
14        """Retourne l'element au sommet sans le retirer"""
15        return self.contenu[-1] if self.contenu else None
16
17    def est_vide(self):
18        """Verifie si la pile est vide"""
19        return not bool(self.contenu)
```

17.2 File (FIFO)

Operations :

enqueue (enfiler) : ajouter a la fin

dequeue (defiler) : retirer du debut

peek (premier) : consulter le premier

isEmpty (estVide) : verifier si vide

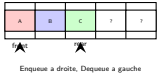
Complexite : $\mathcal{O}(1)$ avec implementation optimisee (deque)

Utilisations :

Parcours en largeur (BFS)

Gestion de taches (ordonnancement)

Buffers (tampons)



```
1 from collections import deque
2
3 class File:
4     def __init__(self):
5         """Initialise une file vide utilisant une deque"""
6         self.file = deque()
7
8     def enfiler(self, x):
9         """Ajoute un element a la fin de la file"""
10        self.file.appendleft(x)
11
12    def defiler(self):
13        """Retire et retourne l'element en tete de file"""
14        return self.file.pop() if self.file else None
15
16    def premier(self):
17        """Retourne l'element en tete sans le retirer"""
18        return self.file[-1] if self.file else None
19
20    def est_vide(self):
21        """Verifie si la file est vide"""
22        return not bool(self.file)
```

17.3 Listes Chainees

• Types :

• Simple : un seul lien vers le suivant

• Double : liens vers le precedent et le suivant

• Circulaire : le dernier element pointe vers le premier

• Complexite :

• Insertion/suppression en tete : $\mathcal{O}(1)$

• Recherche/acces par position : $\mathcal{O}(n)$

• Insertion/suppression en queue : $\mathcal{O}(1)$ (avec pointeur fin) ou $\mathcal{O}(n)$

• Acces au i-eme element : $\mathcal{O}(n)$

• Avantages :

• Taille dynamique (pas de redimensionnement)

• Insertion/suppression efficaces en $\mathcal{O}(1)$ si position connue

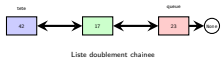
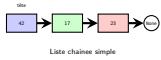
• Pas de gaspillage de memoire (allocation dynamique)

Inconvenients :

Acces sequentiel (pas d'acces direct aux elements)

Surcharge memoire pour les pointeurs

Moins efficace en termes de cache CPU



```
1 class Noeud:
2     """Classe representant un noeud d'une liste chainee simple"""
3     def __init__(self, val, suiv=None):
4         self.valeur = val          Valeur stockee dans le noeud
5         self.suivant = suiv        Reference vers le noeud suivant
6
7 class ListeChainee:
8     """Implementation d'une liste chainee simple"""
9     def __init__(self):
10        """Initialise une liste chainee vide"""
11        self.tete = None
12
13    def est_vide(self):
14        """Verifie si la liste est vide"""
15        return self.tete is None
16
17    def inserer_debut(self, val):
18        """Insere un element au debut de la liste"""
19        self.tete = Noeud(val, self.tete)
20
21    def inserer_fin(self, val):
22        """Insere un element a la fin de la liste"""
23        if not self.tete:
24            self.tete = Noeud(val)
25            return
26
27        courant = self.tete
28        while courant.suivant:
29            courant = courant.suivant
30        courant.suivant = Noeud(val)
31
32    def supprimer(self, val):
33        """Supprime la premiere occurrence de la valeur dans la liste"""
34        if not self.tete:
35            return Liste Vide Cas special : suppression de la tete
36        if self.tete.valeur == val:
37            self.tete = self.tete.suivant
38            return
39
40        Recherche du noeud a supprimer
41        precedent = self.tete
42        courant = self.tete.suivant
43        while courant:
44            if courant.valeur == val:
45                precedent.suivant = courant.suivant
46                del courant
47                return
48            precedent = courant
49            courant = courant.suivant
49        courant:elements.append(str(courant.valeur))
49        courant = courant.suivant
49        print("-> ".join(elements) if elements else "Liste vide")
```

18 Arbres Generaux

18.1 Traversees d'Arbres n-aires

18.1.1 BFS (Breadth-First Search)

Complexite : $\mathcal{O}(n)$ temps, $\mathcal{O}(\text{width})$ memoire (largeur max de l'arbre)

```
1 // Pseudo-code BFS pour arbres n-aires
2 BFS(root):
3     Q <- queue()
4     Q.push(root)
5     while Q not empty:
6         u <- Q.pop()
7         visit(u)
8         for v in u.children:
```

```
9 Q.push(v)
10
11 // Implementation C++ pour arbre n-aire
12 struct TreeNode {
13     int val;
14     vector<TreeNode*> children;
15     TreeNode(int x) : val(x) {}
16 };
17
18 void bfs_nary_tree(TreeNode* root) {
19     if (!root) return;
20
21     queue<TreeNode*> q;
22     q.push(root);
23
24     while (!q.empty()) {
25         TreeNode* node = q.front();
26         q.pop();
27
28         // Traiter le noeud
29         cout << node->val << " ";
30
31         // Ajouter tous les enfants
32         for (TreeNode* child : node->children) {
33             if (child) {
34                 q.push(child);
35             }
36         }
37     }
38 }
```

18.1.2 Rappel Pre-ordre et Post-ordre

```
1 // Pre-ordre : traiter noeud AVANT les enfants
2 void preorder_nary(TreeNode* root) {
3     if (!root) return;
4
5     cout << root->val << " "; // Traiter d'abord
6
7     for (TreeNode* child : root->children) {
8         preorder_nary(child);
9     }
10 }
11
12 // Post-ordre : traiter noeud APRES les enfants
13 void postorder_nary(TreeNode* root) {
14     if (!root) return;
15
16     for (TreeNode* child : root->children) {
17         postorder_nary(child);
18     }
19
20     cout << root->val << " "; // Traiter a la fin
21 }
```

19 Arbres Binaires de Recherche (BST)

Definition :

Arbre binaire ou chaque noeud a au plus 2 enfants

Propriete BST : left < noeud < right

Complexite moyenne : $\mathcal{O}(\log n)$ (equilibre)

Pire cas : $\mathcal{O}(n)$ (degenere en liste chainee)

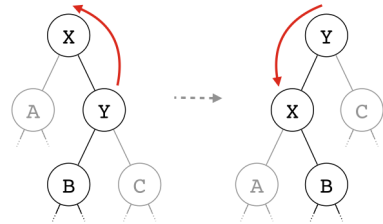
Operations :

Recherche : $\mathcal{O}(h)$

Insertion : $\mathcal{O}(h)$

Suppression : $\mathcal{O}(h)$

(h = hauteur de l'arbre)



Insertion dans un BST :

1. Parcourir l'arbre depuis la racine
2. Comparer avec le noeud courant

3. Aller a gauche si plus petit, a droite si plus grand

4. Inserer a la position trouvee

Suppression dans un BST :

Feuille : supprimer directement

Un seul enfant : le remplacer par son enfant

Deux enfants : remplacer par le successeur (ou predecesseur) et supprimer

ce dernier

19.1 Reconstruction préordre + inorder

```
1 def build(pre, ino):
2     if not pre:
3         return None
4     r = Node(pre[0])
5     i = ino.index(pre[0])
6     r.left = build(pre[1:i+1], ino[:i])
7     r.right = build(pre[i+1:], ino[i+1:])
8     return r
```

20 Tas (Heap)

Définition : Arbre binaire quasi-complet respectant la propriete de tas.

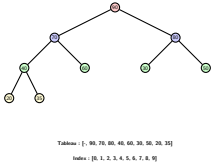
Propriete de tas :

- **Max-heap :** $parent \geq enfants$

- **Min-heap :** $parent \leq enfants$

20.1 Structure et Representation

Représentation en tableau : Indexation commence a 1



Relations parent-enfant :

- Parent de l'index i : $\lfloor i/2 \rfloor$
- Enfant gauche de i : $2i$
- Enfant droit de i : $2i + 1$

20.2 Algorithme Heapify (Percoler vers le bas)

But : Restaurer la propriete de tas vers le bas depuis un noeud.

```
1 void heapify_down(vector<int>& heap, int i, int heap_size) {
2     int largest = i;
3     int left = 2 * i;
4     int right = 2 * i + 1;
5
6     // Comparer avec enfant gauche
7     if (left <= heap_size && heap[left] > heap[largest]) {
8         largest = left;
9     }
10
11    // Comparer avec enfant droit
12    if (right <= heap_size && heap[right] > heap[largest]) {
13        largest = right;
14    }
15
16    // Si le plus grand n'est pas le parent
17    if (largest != i) {
18        swap(heap[i], heap[largest]);
19        heapify_down(heap, largest, heap_size); // Recursion
20    }
21 }
```

Complexite : $O(\log n)$ - hauteur de l'arbre

20.3 Construction d'un Heap (Build-Heap)

Methode optimale : Heapify de bas en haut

```
1 void build_max_heap(vector<int>& arr) {
2     int n = arr.size() - 1; // Index 0 non utilise
3
4     // Commencer du dernier parent : floor(n/2)
5     for (int i = n/2; i >= 1; i--) {
6         heapify_down(arr, i, n);
7     }
8 }
9
10 // Exemple d'utilisation
11 vector<int> data = {0, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
12 build_max_heap(data);
13 // Resultat: [0, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

Complexite : $O(n)$ - Non pas $O(n \log n)$!

Pourquoi $O(n)$? :

- Niveau h : $\leq 2^{n-h-1}$ noeuds
- Cout par noeud : $O(h)$
- Total : $\sum_{h=0}^{\log n} 2^{n-h-1} \cdot O(h) = O(n)$

20.4 Operations sur Heap

Insertion : Ajouter a la fin + percoler vers le haut

```
1 void heap_insert(vector<int>& heap, int value) {
2     heap.push_back(value);
3     int i = heap.size() - 1;
4
5     // Percoler vers le haut
6     while (i > 1 && heap[i/2] < heap[i]) {
7         swap(heap[i], heap[i/2]);
8         i = i/2;
9     }
10 }
```

Extraction du maximum :

```
1 int extract_max(vector<int>& heap) {
2     if (heap.size() <= 1) return -1; // Heap vide
3
4     int max_val = heap[1];
5     heap[1] = heap.back(); // Dernier element a la racine
6     heap.pop_back();
7
8     heapify_down(heap, 1, heap.size() - 1);
9     return max_val;
10 }
```

Complexites :

- Insertion : $O(\log n)$
- Extraction : $O(\log n)$
- Peek (max/min) : $O(1)$
- Construction : $O(n)$

20.5 Définition et Propriétés

Structure d'arbre binaire complet

Propriété de tas : chaque noeud est supérieur (max-heap) ou inférieur

(min-heap) à ses fils

Hauteur d'un tas avec n éléments : $\lfloor \log_2 n \rfloor + 1$

20.6 Opérations de Base

Insertion (push_heap) :

Ajout en feuille puis remontée (sift-up)

Complexité : $O(\log n)$

Suppression du max/min (pop_heap) :

Échange racine-feuille, suppression, puis descente (sift-down)

Complexité : $O(\log n)$

Construction (make_heap) :

Construction en $O(n)$ par sift-down depuis les feuilles

20.7 Tri par Tas (Heapsort)

1. Construction du tas max : $O(n)$
2. n extractions du maximum : $O(n \log n)$
3. Tri en place, non stable
4. Complexité : $O(n \log n)$ dans tous les cas

20.8 STL - Fonctions sur les Tas

#include <algorithm>

#include <vector>

#include <queue> // Pour std::priority_queue

std::vector<int> v = {3, 1, 4, 1, 5, 9};

// Construire un tas max

std::make_heap(v.begin(), v.end()); // {9,5,4,1,1,3}

// Ajouter un élément

v.push_back(6);

std::push_heap(v.begin(), v.end()); // {9,5,6,1,1,3,4}

// Supprimer le plus grand élément

std::pop_heap(v.begin(), v.end()); // déplace le max en fin de vecteur

v.pop_back(); // supprime l'élément

// Trier avec heapsort

std::sort_heap(v.begin(), v.end()); // tri croissant

20.9 File de Priorité

Implémentée par défaut avec un tas max

Opérations :

push : $O(\log n)$

pop : $O(\log n)$

top : $O(1)$

Exemple :

#include <queue>

std::priority_queue<int> pq; // max-heap par défaut

// Pour un min-heap :

// std::priority_queue<int, std::vector<int>, std::greater<int>> pq;

pq.push(3);

pq.push(1);

int top = pq.top(); // 3

pq.pop(); // supprime le 3

20.10 Comparaison : Fonctions Binary Heap vs std : :priority_queue

Opération	Binary Heap (STL)	std : :priority_queue
Construction	make_heap(v.begin(), v.end()) $O(n)$	Constructeur direct $O(n \log n)$ si séquentiel
Ajout élément	push_heap() après push_back() $O(\log n)$	push() $O(\log n)$
Suppression max	pop_heap() puis pop_back() $O(\log n)$	pop() $O(\log n)$
Accès max	v.front() $O(1)$	top() $O(1)$
Taille	v.size() $O(1)$	size() $O(1)$
Vide ?	v.empty() $O(1)$	empty() $O(1)$
Tri complet	sort_heap() $O(n \log n)$	Non disponible (destructif)
Accès conteneur	Direct via vector Permet itération	Aucun accès Encapsulation

Recommandations :

- std::priority_queue : interface plus propre, sécurisée

- Binary heap manuel : contrôle total, tri possible, accès direct

21 Heaps & Priority Queues - Synthèse

21.1 Table de Complexites

Operation	binary_heap	std : :priority_queue
make	$O(n)$	$O(n)$
push	$O(\log n)$	$O(\log n)$
pop	$O(\log n)$	$O(\log n)$
top	$O(1)$	$O(1)$

21.2 Applications Classiques

- **Heapsort** : tri en $O(n \log n)$ garanti
- **k-largest** : trouver les k plus grands elements
- **Dijkstra** : plus courts chemins avec file de priorite
- **Huffman** : codage optimal avec tas min
- **Simulation** : gestion d'evenements temporels

21.3 Heap vs Autres Structures

Structure	Insert	Find Min	Delete Min
Array trie	$O(n)$	$O(1)$	$O(n)$
Array non trie	$O(1)$	$O(n)$	$O(n)$
Binary Heap	$O(\log n)$	$O(1)$	$O(\log n)$
BST equilibre	$O(\log n)$	$O(\log n)$	$O(\log n)$

Avantage du heap : Find Min en $O(1)$ + Insert/Delete en $O(\log n)$

21.4 Applications

Tri partiel (trouver les k plus grands éléments)

Ordonnancement de tâches

Algorithme de Dijkstra

Tri d'une liste presque triée

21.5 Exercices Types

- 1. Implémenter un tas min à partir de zéro
- 2. Trouver les k plus grands éléments dans un tableau
- 3. Fusionner k listes triées
- 4. Implémenter une file de priorité avec un tas

22 Tables de Hachage

22.1 Concepts de Base

Fonction de hachage : transforme une cle en indice de tableau

Objectifs :

- Distribution uniforme des cle
- Faible probabilite de collisions
- Calcul rapide

Complexite moyenne : $O(1)$ pour recherche/insertion/suppression

22.2 Rappels rapides

- Rehash si $\alpha \gtrsim 0.75$ - Opérations $O(1)$ amorties, $O(n)$ pire cas si toutes les clés tombent dans un seau - Chaînage : insertion en tête via push_front sans réordonner

22.3 Gestion des Collisions

Chainage (Chaining) :

- Stockage des elements en collision dans une liste chaine
- Complexite : $O(1 + \alpha)$ ou α est le facteur de charge

Adressage Ouvert (Open Addressing) :

- Stockage direct dans le tableau avec recherche lineaire/quadratique/double hachage

Complexite : $O(1/(1 - \alpha))$ pour la recherche

22.4 Fonctions de Hachage

Pour les entiers : division ($h(k) = k \% m$), multiplication

Pour les chaines :

```
1 size_t hash_string(const string& s, size_t m) {
2     size_t h = 0;
3     for (char c : s)
4         h = (h * 31 + c) % m;
5     return h;
6 }
```

Pour les objets composites : combinaison des hachages des champs

22.5 STL - unordered_map et unordered_set

```
#include <unordered_map>
#include <unordered_set>
#include <string>
```

```
1 // Declaration
2 std::unordered_map<std::string, int> ages;
3 std::unordered_set<std::string> noms;
4
5 // Operations de base
6 ages["Alice"] = 30; // Insertion
7 if (ages.count("Alice")) { // Recherche
8     int age = ages["Alice"]; // Acces
9 }
10 ages.erase("Alice"); // Suppression
11
12 // Iteration
13 for (const auto& [nom, age] : ages) {
14     // Utiliser nom et age
15 }
16
17 // Pour les types personnalisés
18 struct Point { int x, y; };
19 struct PointHash {
20     size_t operator()(const Point& p) const {
21         return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
22     }
23 };
24 std::unordered_set<Point, PointHash> points;
```

22.6 Redimensionnement et Facteur de Charge

Facteur de charge (α) : nombre d'elements / taille du tableau

Seuil de redimensionnement typique : $\alpha_{max} = 0.7$

Operation de rehachage :

- 1. Creer un nouveau tableau plus grand (generalement x2)

- 2. Recalculer le hachage de tous les elements

- 3. Insérer dans le nouveau tableau

Complexite amortie : $O(1)$ par insertion

22.7 Algorithmme d'Insertion avec Gestion du Facteur de Charge

```
1 template<typename K, typename V>
2 class HashTable {
3 private:
4     vector<list<pair<K,V>>> table;
5     size_t num_elements;
6     static constexpr double MAX_LOAD_FACTOR = 0.75;
7
8     void resize() {
9         auto old_table = move(table);
10        table.clear();
11        table.resize(old_table.size() * 2);
12        num_elements = 0;
13
14        // Reinsert all elements
15        for (const auto& bucket : old_table) {
16            for (const auto& [key, value] : bucket) {
17                insert_internal(key, value);
18            }
19        }
20    }
21
22    void insert_internal(const K& key, const V& value) {
23        size_t index = hash(key) % table.size();
24        table[index].emplace_back(key, value);
25        num_elements++;
26    }
27
28 public:
29    void insert(const K& key, const V& value) {
30        // Verify load factor BEFORE insertion
31        if (load_factor() >= MAX_LOAD_FACTOR) {
32            resize();
33        }
34        insert_internal(key, value);
35    }
36
37    double load_factor() const {
38        return static_cast<double>(num_elements) / table.size();
39    }
40 };
```

Points clés insertion :

- Vérification facteur de charge AVANT insertion
- Redimensionnement x2 pour complexité amortie $O(1)$
- Rehachage complet nécessaire (indices changent)
- STL : redimensionnement automatique transparent

23 Hachage Dynamique & Load Factor

23.1 Load Factor et Redimensionnement

Load factor : $\lambda = \frac{n}{\text{buckets}}$ (taux de remplissage)

Regles de redimensionnement :

- Doubler le tableau si $\lambda \geq 0.5$ (insertion)
- Diviser par 2 si $\lambda \leq 0.125$ (suppression)

```
1 class DynamicHashTable {
2 private:
3     vector<vector<int>>> table;
4     int size; // nombre d'elements
5     int capacity; // nombre de buckets
6
7     double load_factor() const {
8         return static_cast<double>(size) / capacity;
9     }
10
11    void resize(int new_capacity) {
12        vector<vector<int>>> old_table = move(table);
13        table.assign(new_capacity, vector<int>());
14        capacity = new_capacity;
15        size = 0;
16
17        // Rehachage de tous les elements
18        for (const auto& bucket : old_table) {
19            for (int key : bucket) {
20                insert_no_resize(key);
21            }
22        }
23    }
24
25 public:
26    void insert(int key) {
27        // Verifier si redimensionnement necessaire
28        if (load_factor() >= 0.5) {
29            resize(capacity * 2);
30        }
31        insert_no_resize(key);
32    }
33
34    void remove(int key) {
35        // Implementation suppression...
36        // Apres suppression :
37        if (load_factor() <= 0.125 && capacity > 4) {
```

```
38         resize(capacity / 2);
39     }
40 }
41 };
```

23.2 Insertion par Chainage - Correction Prof

Important : L'insertion se fait par push_front mais ne reordonne jamais la liste existante!

Erreur Ex. 7.2 : Le 2 reste avant le 4 lors de l'insertion du 9.

```
1 void insert_chaining(int key) {
2     int bucket = hash(key) % capacity;
3
4     // 1. Parcourir la liste pour detecter doublons
5     for (int existing : table[bucket]) {
6         if (existing == key) {
7             return; // Doublet, ne pas inserer
8         }
9     }
10
11    // 2. Insertion en tete (push front)
12    table[bucket].insert(table[bucket].begin(), key);
13    size++;
14
15    // Note: Le reste de la liste n'est PAS modifie !
16 }
```

23.3 Complexites Hachage

Operation	Cas moyen	Pire cas
Insert	$O(1)$ amorti	$O(n)$
Search	$O(1)$ amorti	$O(n)$
Delete	$O(1)$ amorti	$O(n)$
Resize	$O(n)$	$O(n)$

Rappels

- Rehash typiquement si $\alpha \gtrsim 0.75$.
- Opérations en $O(1)$ amorti mais $O(n)$ dans le pire cas.
- Chaînage : insérer en tête sans réordonner.

23.4 Applications

Dictionnaires (cle-valeur)

Dédoublonnage d'ensembles

Cache (mémoire associative)

Comptage de fréquences

Tables de symboles dans les compilateurs

23.5 Exercices Types

- 1. Implémenter une table de hachage avec chaînage
- 2. Détecter les doublons dans un tableau
- 3. Compter les occurrences de chaque mot dans un texte
- 4. Trouver deux nombres dans un tableau dont la somme est égale à une valeur donnée
- 5. Implémenter un cache LRU (Least Recently Used)

24 Graphes - Concepts de Base

24.1 Définitions

Graphe $G = (V, E)$ où V est l'ensemble des sommets et E des arêtes

Types : orienté/non-orienté, pondéré/non-pondéré, cyclique/acyclique

Degré d'un sommet : nombre d'arêtes incidentes

Chemin : séquence de sommets reliés par des arêtes

24.2 Représentations

Matrice d'adjacence :

Matrice $|V| \times |V|$

$A[i][j] = 1$ si arête de i à j , 0 sinon

Complexité mémoire : $O(|V|^2)$

Liste d'adjacence :

Tableau de listes chaînées

Chaque élément pointe vers ses voisins

Complexité mémoire : $O(|V| + |E|)$

Comparaison Matrice vs Liste

Opération	Matrice	Liste
Espace	$O(V^2)$	$O(V + E)$
Test d'adjacence	$O(1)$	$O(\deg(v))$
Énumération voisins	$O(V)$	$O(\deg(v))$

25 Parcours de Graphes

25.1 Profondeur d'abord (DFS)

Exploration en profondeur

Utilise une pile (récursion ou itératif)

Applications : détection de cycles, tri topologique

Complexité : $O(|V| + |E|)$

```
1 void dfs(int v, vector<bool>& visited, const vector<vector<int>>& adj) {
2     visited[v] = true;
3     for (int u : adj[v]) {
4         if (!visited[u]) {
5             dfs(u, visited, adj);
6         }
7     }
8 }
```

25.2 Largeur d'abord (BFS)

Exploration par niveaux

Utilise une file

Applications : plus court chemin (poids unitaires), composantes connexes

Complexité : $O(|V| + |E|)$

```
1 void bfs(int start, const vector<vector<int>>& adj) {
2     vector<bool> visited(adj.size(), false);
3     queue<int> q;
4     q.push(start);
5     visited[start] = true;
6
7     while (!q.empty()) {
8         int v = q.front();
9         q.pop();
10        for (int u : adj[v]) {
11            if (!visited[u]) {
12                visited[u] = true;
13                q.push(u);
14            }
15        }
16    }
17 }
```

Points clés BFS :

- File FIFO, marquage au *push*
- Complexités : $O(V + E)$ temps, $O(V)$ mémoire
- Trouve plus courts chemins (poids unitaires)

26 Tri Topologique

26.1 Algorithme de Kahn

Utilise un tableau des degrés entrants

Complexité : $O(V + E)$

```
1 vector<int> kahn_topological_sort(const vector<vector<int>>& adj) {
2     int n = adj.size();
3     vector<int> inDeg(n, 0);
4
5     // Calculer les degr'és entrants
6     for (int u = 0; u < n; u++) {
7         for (int v : adj[u]) {
8             inDeg[v]++;
9         }
10    }
11
12    queue<int> q;
13    for (int i = 0; i < n; i++) {
14        if (inDeg[i] == 0) q.push(i);
15    }
16
17    vector<int> result;
18    while (!q.empty()) {
19        int u = q.front(); q.pop();
20        result.push_back(u);
21
22        for (int v : adj[u]) {
23            inDeg[v]--;
24            if (inDeg[v] == 0) q.push(v);
25        }
26    }
27
28    return result.size() == n ? result : vector<int>{};
29 }
```

26.2 Multi-DFS (Post-ordre)

Série de DFS post-ordre puis inversion finale

Ordre correct exemple : [4 5 3 2 6 1 7 0]

```
1 void dfs_topological(int v, vector<bool>& visited,
2                     const vector<vector<int>>& adj, stack<int>& topoStack) {
3     visited[v] = true;
4     for (int u : adj[v]) {
5         if (!visited[u]) {
6             dfs_topological(u, visited, adj, topoStack);
```

```
7         }
8     }
9     topoStack.push(v); // Post-ordre
10 }
11
12 vector<int> dfs_topological_sort(const vector<vector<int>>& adj) {
13     int n = adj.size();
14     vector<bool> visited(n, false);
15     stack<int> topoStack;
16
17     for (int i = 0; i < n; i++) {
18         if (!visited[i]) {
19             dfs_topological(i, visited, adj, topoStack);
20         }
21     }
22
23     vector<int> result;
24     while (!topoStack.empty()) {
25         result.push_back(topoStack.top());
26         topoStack.pop();
27     }
28     return result;
29 }
```

27 Algorithmes de Plus Courts Chemins

27.1 Dijkstra (Graphes pondérés positifs)

Utilise une file de priorité (min-heap)

Complexité : $O((|V| + |E|) \log |V|)$

Ne fonctionne pas avec des poids négatifs

```
1 vector<int> dijkstra(int start, const vector<vector<pair<int, int>>& adj) {
2     int n = adj.size();
3     vector<int> dist(n, INT_MAX);
4     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
5
6     dist[start] = 0;
7     pq.push({0, start});
8
9     while (!pq.empty()) {
10        auto [d, u] = pq.top();
11        pq.pop();
12        if (d > dist[u]) continue;
13
14        for (auto [v, w] : adj[u]) {
15            if (dist[u] + w < dist[v]) {
16                dist[v] = dist[u] + w;
17                pq.push({dist[v], v});
18            }
19        }
20    }
21    return dist;
22 }
```

Canevas Dist/Parent

Etape	Sommet	dist	parent

27.2 Bellman-Ford (Graphes avec poids négatifs)

Détecte les cycles de poids négatifs

Complexité : $O(|V| \cdot |E|)$

Relâchement des arêtes $|V| - 1$ fois

```
1 bool bellman_ford(int start, const vector<vector<pair<int, int>>& adj, vector<int>& dist) {
2     int n = adj.size();
3     dist.assign(n, INT_MAX);
4     dist[start] = 0;
5
6     // Relaxation des aretes |V| - 1 fois
7     for (int i = 0; i < n - 1; i++) {
8         for (int u = 0; u < n; u++) {
9             if (dist[u] != INT_MAX) {
10                for (auto [v, w] : adj[u]) {
11                    if (dist[u] + w < dist[v]) {
12                        dist[v] = dist[u] + w;
13                    }
14                }
15            }
16        }
17    }
18
19    // Verification des cycles negatifs
20    for (int u = 0; u < n; u++) {
21        if (dist[u] != INT_MAX) {
22            for (auto [v, w] : adj[u]) {
23                if (dist[u] + w < dist[v]) {
24                    return false; // Cycle negatif detecte
25                }
26            }
27        }
28    }
29    return true;
30 }
```

Canevas Dist/Parent

Etape	Sommet	dist	parent

28 Arbres Couvrants Minimaux (MST)

28.1 Algorithme de Prim

Construction gloutonne du MST

Similaire à Dijkstra mais avec des distances différentes

Complexité : $O((|V| + |E|) \log |V|)$ avec une file de priorité

```
1 struct Edge {
2     int u, v, weight;
3     bool operator<(const Edge& other) const {
4         return weight > other.weight; // Min-heap
5     }
6 };
7
8 int prim_mst(const vector<vector<pair<int, int>>& adj) {
9     int n = adj.size();
10    vector<bool> inMST(n, false);
11    priority_queue<Edge> pq;
12    int mstWeight = 0;
13
14    // Commencer avec le sommet 0
15    inMST[0] = true;
16    for (auto [v, w] : adj[0]) {
17        pq.push({0, v, w});
18    }
19
20    while (!pq.empty()) {
21        Edge e = pq.top();
22        pq.pop();
23
24        if (inMST[e.v]) continue;
25
26        // Ajouter l'arete au MST
27        inMST[e.v] = true;
28        mstWeight += e.weight;
29
30        // Ajouter les nouvelles aretes
31        for (auto [u, w] : adj[e.v]) {
32            if (!inMST[u]) {
33                pq.push({e.v, u, w});
34            }
35        }
36    }
37
38    return mstWeight;
39 }
```

28.2 Algorithme de Kruskal

Trie les arêtes par poids croissant

Utilise une structure Union-Find

Complexité : $O(|E| \log |E|)$

```
1 class UnionFind {
2 public:
3     vector<int> parent, rank;
4
5     UnionFind(int n) : parent(n), rank(n, 0) {
6         iota(parent.begin(), parent.end(), 0);
7     }
8
9     int find(int x) {
10        if (parent[x] != x) {
11            parent[x] = find(parent[x]); // Compression de chemin
12        }
13        return parent[x];
14    }
15
16    bool unite(int x, int y) {
17        int px = find(x), py = find(y);
18        if (px == py) return false;
19
20        if (rank[px] < rank[py]) swap(px, py);
21        parent[py] = px;
22        if (rank[px] == rank[py]) rank[px]++;
23        return true;
24    }
25 };
26
27 int kruskal_mst(int n, vector<Edge>& edges) {
28     sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
29         return a.weight < b.weight;
30     });
31
32     UnionFind uf(n);
33     int mstWeight = 0;
34     int edgesUsed = 0;
35
36     for (const Edge& e : edges) {
37         if (uf.unite(e.u, e.v)) {
38             mstWeight += e.weight;
39             edgesUsed++;
40             if (edgesUsed == n - 1) break;
41         }
42     }
```

```
43     return mstWeight;
44 }
45 }
```

Comparaison MST :

Algorithme	Complexité	Meilleur pour
Prim	$O((V + E) \log V)$	Graphes denses
Kruskal	$O(E \log E)$	Graphes clairsemés

29 Composantes Fortement Connexes

29.1 Algorithme de Kosaraju-Sharir

Complexité : $O(V + E)$

Étapes :

1. Graphe inversé
2. DFS post-ordre multi-pass (ordre croissant)
3. Inversion liste
4. DFS sur G, nouvel ID par appel

```
1 vector<vector<int>> kosaraju(const vector<vector<int>>& adj) {
2     int n = adj.size();
3
4     // 1. Construire le graphe inverse
5     vector<vector<int>> radj(n);
6     for (int u = 0; u < n; u++) {
7         for (int v : adj[u]) {
8             radj[v].push_back(u);
9         }
10    }
11
12    // 2. Premier DFS sur graphe inverse
13    vector<bool> visited(n, false);
14    stack<int> finishOrder;
15
16    function<void(int)> dfs1 = [&](int v) {
17        visited[v] = true;
18        for (int u : radj[v]) {
19            if (!visited[u]) dfs1(u);
20        }
21        finishOrder.push(v);
22    };
23
24    for (int i = 0; i < n; i++) {
25        if (!visited[i]) dfs1(i);
26    }
27
28    // 3. Deuxième DFS sur graphe original
29    fill(visited.begin(), visited.end(), false);
30    vector<vector<int>> sccs;
31
32    function<void(int, vector<int>&)> dfs2 = [&](int v, vector<int>& comp) {
33        visited[v] = true;
34        comp.push_back(v);
35
36        for (int u : adj[v]) {
37            if (!visited[u]) dfs2(u, comp);
38        }
39    };
40
41    while (!finishOrder.empty()) {
42        int v = finishOrder.top(); finishOrder.pop();
43        if (!visited[v]) {
44            vector<int> component;
45            dfs2(v, component);
46            sccs.push_back(component);
47        }
48    }
49
50    return sccs;
51 }
```

30 Applications Avancées

30.1 Composantes Fortement Connexes (Kosaraju)

1. Premier parcours DFS pour l'ordre de fin
2. Graphe inverse
3. Second parcours DFS dans l'ordre inverse
4. Complexité : $O(|V| + |E|)$

30.2 Tri Topologique

Pour les graphes acycliques orientés (DAG)

Utilise un parcours DFS avec tri par temps de fin

Applications : ordonnancement de tâches

31 Exercices Types

1. Compter le nombre de composantes connexes
2. Détecter un cycle dans un graphe
3. Trouver le plus court chemin avec contraintes

4. Trouver les points d'articulation

5. Calculer le diamètre d'un arbre

32 STL et Bibliothèques Utiles

```
1 include <vector>include <queue>
2 include <set>include <stack>
3
4 // Représentation avec liste d'adjacence
5 vector<vector<int>> adj; // non pondere
6 vector<vector<pair<int, int>>> wadj; // pondere (voisin, poids)
7
8
9 // Pour Dijkstra
10 priority_queue<pair<int, int>> pq; // max-heap par défaut
11 // Pour min-heap :
12 // priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
13
14 // Pour les ensembles ordonnés
15 set<int> s; // éléments uniques tries
16 multiset<int> ms; // éléments multiples tries
```

33 Graphes

33.1 Représentations de Graphes

Critère	Matrice d'adjacence	Liste d'adjacence
Espace	$O(V^2)$	$O(V + E)$
Test d'adjacence	$O(1)$	$O(\deg(v))$
Énumération voisins	$O(V)$	$O(\deg(v))$

33.2 Parcours de Graphes

33.2.1 BFS (Breadth-First Search)

Principe : File FIFO, marquage au push

Complexité : $O(V + E)$ temps, $O(V)$ mémoire

```
1 void bfs(int start, const vector<vector<int>>& adj) {
2     vector<bool> visited(adj.size(), false);
3     queue<int> q;
4
5     q.push(start);
6     visited[start] = true; // Marquage au push
7
8     while (!q.empty()) {
9         int u = q.front();
10        q.pop();
11
12        // Traiter le sommet u
13        cout << u << " ";
14
15        for (int v : adj[u]) {
16            if (!visited[v]) {
17                visited[v] = true; // Marquage au push
18                q.push(v);
19            }
20        }
21    }
22 }
```

33.2.2 DFS (Depth-First Search)

Itératif (pile explicite) et récursif

Modes : pre-ordre, post-ordre

```
1 // DFS Itératif
2 void dfs_iterative(int start, const vector<vector<int>>& adj) {
3     vector<bool> visited(adj.size(), false);
4     stack<int> s;
5
6     s.push(start);
7
8     while (!s.empty()) {
9         int u = s.top();
10        s.pop();
11
12        if (!visited[u]) {
13            visited[u] = true;
14            cout << u << " "; // Pre-ordre
15
16            // Ajouter voisins dans l'ordre inverse pour parcours correct
17            for (auto it = adj[u].rbegin(); it != adj[u].rend(); ++it) {
18                if (!visited[*it]) {
19                    s.push(*it);
20                }
21            }
22        }
23    }
24 }
25
26 // DFS Récursif avec post-ordre
27 void dfs_recursive_postorder(int u, const vector<vector<int>>& adj,
28                             vector<bool>& visited, vector<int>& post_order) {
29     visited[u] = true;
30
31     for (int v : adj[u]) {
```

```
32         if (!visited[v]) {
33             dfs_recursive_postorder(v, adj, visited, post_order);
34         }
35     }
36     post_order.push_back(u); // Post-ordre
37 }
38 }
```

33.3 Tri Topologique

33.3.1 Algorithme de Kahn

Tableau inDeg, queue, détection cycle ($|L| < V$)

```
1 vector<int> topological_sort_kahn(const vector<vector<int>>& adj) {
2     int n = adj.size();
3     vector<int> inDegree(n, 0);
4     vector<int> result;
5     queue<int> q;
6
7     // Calculer les degrés entrants
8     for (int u = 0; u < n; u++) {
9         for (int v : adj[u]) {
10            inDegree[v]++;
11        }
12    }
13
14    // Ajouter tous les sommets de degré 0
15    for (int i = 0; i < n; i++) {
16        if (inDegree[i] == 0) {
17            q.push(i);
18        }
19    }
20
21    while (!q.empty()) {
22        int u = q.front();
23        q.pop();
24        result.push_back(u);
25
26        for (int v : adj[u]) {
27            inDegree[v]--;
28            if (inDegree[v] == 0) {
29                q.push(v);
30            }
31        }
32    }
33
34    // Détection de cycle
35    if (result.size() != n) {
36        return {}; // Cycle détecté
37    }
38
39    return result;
40 }
```

33.3.2 Multi-DFS (méthode du prof)

Série de DFS post-ordre puis inversion finale

Correction Ex. 6.3.b : [4, 5, 3, 2, 6, 1, 7, 0] (pas [4, 3, 5, 2, 6, 1, 7, 0])

```
1 vector<int> topological_sort_dfs(const vector<vector<int>>& adj) {
2     int n = adj.size();
3     vector<bool> visited(n, false);
4     vector<int> post_order;
5
6     // Série de DFS post-ordre (ordre croissant 0,1,2,...)
7     for (int i = 0; i < n; i++) {
8         if (!visited[i]) {
9             dfs_recursive_postorder(i, adj, visited, post_order);
10        }
11    }
12
13    // Inversion du résultat final
14    reverse(post_order.begin(), post_order.end());
15    return post_order;
16 }
```

33.4 Composantes Fortement Connexes (Kosaraju-Sharir)

Algorithme (clarification du prof) :

1. Inversion du graphe (chaque arc change de direction)
2. Série d'appels DFS post-ordre sur graphe inverse (ordre croissant 0,1,2,...)
3. Inversion de la liste post-ordre obtenue
4. DFS sur graphe original avec nouvel ID par appel, dans l'ordre de l'étape 3

```
1 vector<vector<int>> strongly_connected_components(const vector<vector<int>>& adj) {
2     int n = adj.size();
3
4     // 1. Inverser le graphe
5     vector<vector<int>> rev_adj(n);
6     for (int u = 0; u < n; u++) {
7         for (int v : adj[u]) {
8             rev_adj[v].push_back(u);
9         }
10    }
11
12    // 2. Série de DFS post-ordre sur graphe inverse
13    vector<bool> visited(n, false);
14    vector<int> post_order;
```

```
15
16 for (int i = 0; i < n; i++) {
17     if (!visited[i]) {
18         dfs_recursive_postorder(i, rev_adj, visited, post_order);
19     }
20 }
21
22 // 3. Inversion du post-ordre
23 reverse(post_order.begin(), post_order.end());
24
25 // 4. DFS sur graphe original avec nouvel ID par appel
26 fill(visited.begin(), visited.end(), false);
27 vector<vector<int>> sccs;
28
29 for (int u : post_order) {
30     if (!visited[u]) {
31         vector<int> component;
32         dfs_component(u, adj, visited, component);
33         sccs.push_back(component);
34     }
35 }
36
37 return sccs;
38 }
39
40 void dfs_component(int u, const vector<vector<int>>& adj,
41                  vector<bool>& visited, vector<int>& component) {
42     visited[u] = true;
43     component.push_back(u);
44
45     for (int v : adj[u]) {
46         if (!visited[v]) {
47             dfs_component(v, adj, visited, component);
48         }
49     }
50 }
```

33.5 Plus Courts Chemins

33.5.1 Dijkstra (resume)

File de priorite, relax arcs

Complexite : $O((V + E) \log V)$

```
1 vector<int> dijkstra(int start, const vector<vector<pair<int, int>>& adj) {
2     int n = adj.size();
3     vector<int> dist(n, INT_MAX);
4     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
5
6     dist[start] = 0;
7     pq.push({0, start}); // {distance, vertex}
8
9     while (!pq.empty()) {
10         auto [d, u] = pq.top();
11         pq.pop();
12
13         if (d > dist[u]) continue; // Arc obsolete
14
15         for (auto [v, w] : adj[u]) {
16             if (dist[u] + w < dist[v]) {
17                 dist[v] = dist[u] + w;
18                 pq.push({dist[v], v});
19             }
20         }
21     }
22
23     return dist;
24 }
```

34 Minimum Spanning Tree (MST)

Definition : Arbre couvrant de poids minimum dans un graphe connexe pondere.

Proprietes :

- Contient tous les sommets
- $n - 1$ aretes pour n sommets
- Aucun cycle
- Poids total minimal

34.1 Algorithme de Prim

Principe : Construction incrementale depuis un sommet initial.

```
1 include <queue>include <vector>
2 using namespace std;
3
4 struct Edge {
5     int to, weight;
6     bool operator<(const Edge& other) const {
7         return weight > other.weight; // Min-heap
8     }
9 };
10
11 vector<Edge> prim(const vector<vector<Edge>>& graph) {
12     int n = graph.size();
13     vector<bool> visited(n, false);
14     vector<Edge> mst;
15     priority_queue<Edge> pq;
16
17     // Commencer du sommet 0
```

```
15 visited[0] = true;
16 for (const auto& edge : graph[0]) {
17     pq.push(edge);
18 }
19
20 while (!pq.empty() && mst.size() < n - 1) {
21     Edge current = pq.top();
22     pq.pop();
23
24     if (visited[current.to]) continue;
25
26     // Ajouter l'arete au MST
27     mst.push_back(current);
28     visited[current.to] = true;
29
30     // Ajouter nouvelles aretes adjacentes
31     for (const auto& edge : graph[current.to]) {
32         if (!visited[edge.to]) {
33             pq.push(edge);
34         }
35     }
36 }
37
38 return mst;
39 }
```

Complexite : $O(E \log V)$ avec priority_queue

34.2 Union-Find (Disjoint Set)

Structure necessaire pour Kruskal :

```
1 class UnionFind {
2 private:
3     vector<int> parent, rank;
4
5 public:
6     UnionFind(int n) : parent(n), rank(n, 0) {
7         for (int i = 0; i < n; i++) {
8             parent[i] = i; // Chaque element est son propre parent
9         }
10    }
11
12    // Find avec compression de chemin
13    int find(int x) {
14        if (parent[x] != x) {
15            parent[x] = find(parent[x]); // Compression
16        }
17        return parent[x];
18    }
19
20    // Union par rang
21    bool unite(int x, int y) {
22        int rootX = find(x);
23        int rootY = find(y);
24
25        if (rootX == rootY) return false; // Deja connectes
26
27        // Union par rang pour optimiser
28        if (rank[rootX] < rank[rootY]) {
29            parent[rootX] = rootY;
30        } else if (rank[rootX] > rank[rootY]) {
31            parent[rootY] = rootX;
32        } else {
33            parent[rootY] = rootX;
34            rank[rootX]++;
35        }
36
37        return true;
38    }
39
40    bool connected(int x, int y) {
41        return find(x) == find(y);
42    }
43};
```

Complexite Union-Find : $O(\alpha(n))$ amortized (quasi-constant)

34.3 Algorithme de Kruskal

Principe : Trier toutes les aretes, ajouter si pas de cycle.

```
1 struct EdgeKruskal {
2     int from, to, weight;
3     bool operator<(const EdgeKruskal& other) const {
4         return weight < other.weight;
5     }
6 };
7
8 vector<EdgeKruskal> kruskal(vector<EdgeKruskal>& edges, int n) {
9     vector<EdgeKruskal> mst;
10    UnionFind uf(n);
11
12    // Trier les aretes par poids croissant
13    sort(edges.begin(), edges.end());
14
15    for (const auto& edge : edges) {
16        // Si les sommets ne sont pas connectes
17        if (uf.unite(edge.from, edge.to)) {
18            mst.push_back(edge);
19        }
20
21        // MST complet quand n-1 aretes
22        if (mst.size() == n - 1) break;
23    }
```

```
23 }
24
25     return mst;
26 }
```

Complexite : $O(E \log E)$ (tri dominant)

34.4 Comparaison Prim vs Kruskal

Critere	Prim	Kruskal
Complexite	$O(E \log V)$	$O(E \log E)$
Graphe dense	Meilleur	-
Graphe sparse	-	Meilleur
Implementation	Priority queue	Tri + Union-Find
Memoire	$O(V)$	$O(E)$

35 Plus Courts Chemins avec Poids Negatifs

35.1 Algorithme de Bellman-Ford

Usage : Detecte cycles negatifs, gere poids negatifs.

```
1 struct EdgeBF { int from, to, weight; };
2
3 pair<vector<int>, bool> bellmanFord(const vector<EdgeBF>& edges,
4                                  int n, int source) {
5
6     vector<int> dist(n, INT_MAX);
7     dist[source] = 0;
8
9     // Relaxation : n-1 iterations
10    for (int i = 0; i < n - 1; i++) {
11        for (const auto& edge : edges) {
12            if (dist[edge.from] != INT_MAX &&
13                dist[edge.from] + edge.weight < dist[edge.to]) {
14                dist[edge.to] = dist[edge.from] + edge.weight;
15            }
16        }
17    }
18
19    // Detection cycle negatif
20    for (const auto& edge : edges) {
21        if (dist[edge.from] != INT_MAX &&
22            dist[edge.from] + edge.weight < dist[edge.to]) {
23            return {dist, false}; // Cycle negatif detecte
24        }
25    }
26
27    return {dist, true}; // Pas de cycle negatif
28 }
```

Complexite : $O(VE)$

Avantages vs Dijkstra :

- Gere les poids negatifs
- Detecte les cycles negatifs
- Plus simple a implementer

Inconvenients :

- Plus lent que Dijkstra
- Ne fonctionne pas avec cycles negatifs

36 Arbres Équilibrés - AVL

36.1 Définition et Propriétés

Un arbre AVL est un arbre binaire de recherche équilibré en hauteur. Pour chaque nœud, la hauteur des sous-arbres gauche et droit diffère d'au plus 1.

Hauteur maximale : $\approx 1.44 \log_2(n + 2) - 0.328$

Complexité garantie : $O(\log n)$ pour recherche/insertion/suppression

Rappel : $bf(n) = h(gauche) - h(droit) \in \{-1, 0, 1\}$. Si $|bf| = 2$: LL, LR, RR ou RL selon les signes.

36.2 Facteur d'Équilibre

$\text{équilibre}(n) = \text{hauteur}(\text{droit}) - \text{hauteur}(\text{gauche})$

Un nœud est équilibré si $|\text{équilibre}(n)| \leq 1$; sinon rotation

Valeurs possibles : $-1, 0, +1$

Mise à jour après insertion/suppression le long du chemin d'accès

36.3 Rotations

Rotation simple à droite :

Cas : sous-arbre gauche du fils gauche trop grand

Complexité : $O(1)$

Rotation simple à gauche :

Cas : sous-arbre droit du fils droit trop grand

Complexité : $O(1)$

Double rotation droite-gauche :

Cas : sous-arbre gauche du fils droit trop grand

= Rotation droite du fils droit + rotation gauche

Double rotation gauche-droite :

Cas : sous-arbre droit du fils gauche trop grand

= Rotation gauche du fils gauche + rotation droite

36.4 Table des Rotations AVL

Équilibre nœud	Équilibre enfant	Type rotation	Complexité
< −1	< 0	Simple droite	$O(1)$
< −1	> 0	Double gauche-droite	$O(1)$
> 1	> 0	Simple gauche	$O(1)$
> 1	< 0	Double droite-gauche	$O(1)$

Détails des rotations :

- Simple droite : $y \leftarrow x$, où x = fils gauche de y
- Simple gauche : $x \leftarrow y$, où y = fils droit de x
- Double : combinaison de 2 rotations simples
- Mise à jour : recalculer hauteurs après chaque rotation

36.5 Insertion

1. Insertion standard dans un ABR
2. Remontée vers la racine en mettant à jour les hauteurs
3. Si facteur d'équilibre = ± 2 , effectuer la rotation appropriée
4. Au plus une rotation simple ou double suffit à rééquilibrer

36.6 Suppression

1. Suppression standard dans un ABR
2. Remontée vers la racine en mettant à jour les hauteurs
3. Si facteur d'équilibre = ± 2 , effectuer la rotation appropriée
4. Peut nécessiter jusqu'à $O(\log n)$ rotations

36.7 Implémentation en C++

```
1 struct Noeud {
2     int cle;
3     Noeud *gauche, *droit;
4     int hauteur;
5
6     Noeud(int k) : cle(k), gauche(nullptr), droit(nullptr), hauteur(1) {}
7 };
8
9 int hauteur(Noeud* n) {
10     return n ? n->hauteur : 0;
11 }
12
13 int equilibre(Noeud* n) {
14     return n ? hauteur(n->droit) - hauteur(n->gauche) : 0;
15 }
16
17 void majHauteur(Noeud* n) {
18     if (n) {
19         n->hauteur = 1 + std::max(hauteur(n->gauche), hauteur(n->droit));
20     }
21 }
22
23 Noeud* rotationDroite(Noeud* y) {
24     Noeud* x = y->gauche;
25     Noeud* T2 = x->droit;
26
27     x->droit = y;
28     y->gauche = T2;
29
30     majHauteur(y);
31     majHauteur(x);
32
33     return x;
34 }
35
36 Noeud* rotationGauche(Noeud* x) {
37     Noeud* y = x->droit;
38     Noeud* T2 = y->gauche;
39
40     y->gauche = x;
41     x->droit = T2;
42
43     majHauteur(x);
44     majHauteur(y);
45
46     return y;
47 }
48
49 Noeud* inserer(Noeud* n, int cle) {
50     if (!n) return new Noeud(cle);
51
52     if (cle < n->cle)
53         n->gauche = inserer(n->gauche, cle);
54     else if (cle > n->cle)
55         n->droit = inserer(n->droit, cle);
56     else
57         return n; // Clé es l'égale non autoris'ees
58
59     majHauteur(n);
60
61     int eq = equilibre(n);
```

```
62 // Cas Gauche Gauche
63 if (eq < -1 && cle < n->gauche->cle)
64     return rotationDroite(n);
65
66 // Cas Droit Droit
67 if (eq > 1 && cle > n->droit->cle)
68     return rotationGauche(n);
69
70 // Cas Gauche Droit
71 if (eq < -1 && cle > n->gauche->cle) {
72     n->gauche = rotationGauche(n->gauche);
73     return rotationDroite(n);
74 }
75
76 // Cas Droit Gauche
77 if (eq > 1 && cle < n->droit->cle) {
78     n->droit = rotationDroite(n->droit);
79     return rotationGauche(n);
80 }
81
82 return n;
83 }
84 }
```

37 AVL - Rotations & Suppressions

37.1 Table Rotations AVL

Δ balance	Configuration	Rotation
−2	Gauche-Gauche	Rotation Droite (R)
−2 (+1 sous-gauche)	Gauche-Droite	Double : Gauche-Droite (LR)
+2	Droite-Droite	Rotation Gauche (L)
+2 (-1 sous-droite)	Droite-Gauche	Double : Droite-Gauche (RL)

Rappel : $balance(n)$ = hauteur(droite) − hauteur(gauche)

37.2 Suppression AVL - Algorithmes Complets

Complexité : jusqu'à $O(\log n)$ rotations possibles

```
1 AVLNode* delete_avl(AVLNode* root, int key) {
2     if (!root) return root;
3
4     // 1. Suppression BST standard
5     if (key < root->key) {
6         root->left = delete_avl(root->left, key);
7     } else if (key > root->key) {
8         root->right = delete_avl(root->right, key);
9     } else {
10        // Noeud a supprimer trouve
11        if (!root->left || !root->right) {
12            // Cas 0 ou 1 enfant
13            AVLNode* temp = root->left ? root->left : root->right;
14            if (!temp) {
15                temp = root;
16                root = nullptr;
17            } else {
18                *root = *temp; // Copier le contenu
19            }
20            delete temp;
21        } else {
22            // Cas 2 enfants : successeur in-ordre
23            AVLNode* temp = find_min(root->right);
24            root->key = temp->key;
25            root->right = delete_avl(root->right, temp->key);
26        }
27    }
28
29    if (!root) return root;
30
31    // 2. Mise à jour hauteur
32    update_height(root);
33
34    // 3. Verification equilibre et rotations
35    int balance = get_balance(root);
36
37    // Cas Gauche-Gauche
38    if (balance < -1 && get_balance(root->left) <= 0) {
39        return rotate_right(root);
40    }
41
42    // Cas Gauche-Droite
43    if (balance < -1 && get_balance(root->left) > 0) {
44        root->left = rotate_left(root->left);
45        return rotate_right(root);
46    }
47
48    // Cas Droite-Droite
49    if (balance > 1 && get_balance(root->right) >= 0) {
50        return rotate_left(root);
51    }
52
53    // Cas Droite-Gauche
54    if (balance > 1 && get_balance(root->right) < 0) {
55        root->right = rotate_right(root->right);
56        return rotate_left(root);
57    }
58
59    return root;
60 }
61
62 AVLNode* find_min(AVLNode* node) {
63     while (node->left) node = node->left;
```

```
64     return node;
65 }
```

37.3 Propriétés AVL

Hauteur maximale : $\approx 1.44 \log_2(n + 2) - 0.328$

Complexités garanties : $O(\log n)$ pour recherche/insertion/suppression

37.4 Comparaison avec d'autres structures

- AVL vs ABR standard : Meilleure complexité garantie mais surcoût mémoire
- AVL vs Arbre Rouge-Noir : Plus d'équilibre mais rotations plus fréquentes
- AVL vs Table de hachage : Garantie $O(\log n)$ vs $O(1)$ moyen mais avec ordre total

37.5 Exercices Types

1. Insérer une séquence de clés dans un AVL et montrer les rotations
2. Supprimer un nœud d'un AVL et rééquilibrer
3. Vérifier si un arbre binaire est un AVL
4. Implémenter la suppression dans un AVL
5. Calculer la hauteur minimale/maximale d'un AVL avec n nœuds

38 Programmation Dynamique

38.1 Principe Fondamental

Diviser un problème complexe en sous-problèmes plus simples

Stocker les solutions des sous-problèmes pour éviter les calculs redondants

Reconstruire la solution finale à partir des sous-solutions

Deux approches : descendante (top-down avec mémo"isation) et montante (bottom-up)

38.2 Étapes de Résolution

1. Caractériser la structure d'une solution optimale
2. Définir récursivement la valeur d'une solution optimale
3. Calculer la valeur d'une solution optimale (de manière ascendante ou descendante)
4. Construire la solution optimale à partir des informations calculées

38.3 Exemple Classique : Suite de Fibonacci

```
1 // Version r'ecursive naive
2 int fib(int n) {
3     if (n <= 1) return n;
4     return fib(n-1) + fib(n-2); // 0(2^n)
5 }
6
7 // Version programmation dynamique (top-down avec m'emo"isation)
8 int fib_memo(int n, vector<int>& memo) {
9     if (memo[n] != -1) return memo[n];
10    if (n <= 1) return n;
11    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo);
12    return memo[n]; // 0(n) temps et espace
13 }
14
15 // Version it'erative (bottom-up)
16 int fib_dp(int n) {
17     if (n <= 1) return n;
18     vector<int> dp(n+1);
19     dp[0] = 0; dp[1] = 1;
20     for (int i = 2; i <= n; i++)
21         dp[i] = dp[i-1] + dp[i-2]; // 0(n) temps, 0(n) espace
22     return dp[n];
23 }
24
25 // Version optimis'ee en espace O(1)
26 int fib_opt(int n) {
27     if (n <= 1) return n;
28     int a = 0, b = 1;
29     for (int i = 2; i <= n; i++) {
30         int c = a + b;
31         a = b;
32         b = c;
33     }
34     return b; // 0(n) temps, O(1) espace
35 }
```

39 Problèmes Classiques

39.1 Problème du Sac à Dos (0-1 Knapsack)

Entrée : Objets avec poids et valeurs, capacité maximale W

Objectif : Maximiser la valeur totale sans dépasser W

Sous-structure optimale :

Si on prend l'objet i : valeur[i] + meilleure valeur avec les i-1 premiers objets et capacité W-poids[i]

Si on ne prend pas : meilleure valeur avec les i-1 premiers objets et capacité W

Complexité : $O(nW)$ où n est le nombre d'objets

```
1 int knapsack(int W, const vector<int>& poids, const vector<int>& valeur) {
2     int n = poids.size();
3     vector<vector<int>>> dp(n+1, vector<int>(W+1, 0));
4
5     for (int i = 1; i <= n; i++) {
6         for (int w = 1; w <= W; w++) {
7             if (poids[i-1] <= w)
8                 dp[i][w] = max(valeur[i-1] + dp[i-1][w-poids[i-1]], dp[i-1][w]);
9             else
10                 dp[i][w] = dp[i-1][w];
11         }
12     }
13     return dp[n][W];
14 }
```

39.2 Plus Longue Sous-séquence Commune (LCS)

Entrée : Deux séquences X et Y

Objectif : Trouver la plus longue sous-séquence commune

Relation de récurrence :

Si $X[i] == Y[j]$: $LCS(i,j) = 1 + LCS(i-1,j-1)$

Sinon : $LCS(i,j) = \max(LCS(i-1,j), LCS(i,j-1))$

Complexité : $O(mn)$ où m et n sont les longueurs des séquences

```
1 int lcs(const string& X, const string& Y) {
2     int m = X.length(), n = Y.length();
3     vector<vector<int>>> dp(m+1, vector<int>(n+1, 0));
4
5     for (int i = 1; i <= m; i++) {
6         for (int j = 1; j <= n; j++) {
7             if (X[i-1] == Y[j-1])
8                 dp[i][j] = 1 + dp[i-1][j-1];
9             else
10                 dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
11         }
12     }
13     return dp[m][n];
14 }
```

40 Approche Top-down vs Bottom-up

- Top-down (mémo"isation) :
- Approche récursive
 - Stocke les résultats des sous-problèmes déjà résolus
 - Plus intuitive mais peut avoir une pile d'appels importante
- Bottom-up (remplissage de tableau) :
- Approche itérative
 - Résout les sous-problèmes du plus petit au plus grand
 - Meilleure utilisation du cache, évite la récursion

41 Exercices Types

- Pièces de monnaie : Trouver le nombre minimum de pièces pour faire une somme donnée
- Plus longue sous-séquence croissante (LIS)
- Problème du rendu de monnaie
- Multiplication de matrices enchaînées
- Distance d'édition (Levenshtein)
- Partition égale de somme (subset sum)
- Problème du voyageur de commerce (TSP) avec programmation dynamique

42 Optimisations d'Espace

- Réduire la dimension des tableaux de mémorisation
- Utiliser des tableaux 1D au lieu de 2D quand possible
- Exemple avec le sac à dos 0-1 en $O(W)$ espace :

```
1 int knapsack_opt(int W, const vector<int>& poids, const vector<int>& valeur) {
2     int n = poids.size();
3     vector<int> dp(W+1, 0);
4
5     for (int i = 0; i < n; i++) {
6         // Parcours inverse pour 'éviter la sur-'écriture pr'ématur\`ee
7         for (int w = W; w >= poids[i]; w--) {
8             dp[w] = max(dp[w], valeur[i] + dp[w - poids[i]]);
9         }
10    }
11    return dp[W];
12 }
```

```
12 }
```

43 Arbres Aine/Puine/Parent

Représentation n-aire avec pointeurs : chaque nœud possède un pointeur vers son **ainé** (premier enfant), son **puiné** (frère suivant) et son **parent**.



Navigation :

- ainé : premier enfant
- puiné : frère suivant
- parent : remonter

```
1 struct Node { Node* aine; Node* puine; Node* parent; };
2
3 void pre(Node* n){
4     if(!n) return;
5     visit(n);
6     for(Node* c = n->aine; c; c = c->puine)
7         pre(c);
8 }
9
10 void post(Node* n){
11     for(Node* c = n->aine; c; c = c->puine)
12         post(c);
13     if(n) visit(n);
14 }
```

44 Sémantique C++

Action	Appel
T(const T&)	Constructeur de copie
T(T&&)	Constructeur de déplacement
operator=(const T&)	Affectation copie
operator=(T&&)	Affectation déplacement
T()	Destructeur

vector :

- reserve(n) pré-alloue, pas de mouvement si $n \leq$ capacité.
- Réallocation copie ou déplace tous les éléments.
- push_back(x) copie/déplace un objet.
- emplace_back(args...) construit en place.

operator new vs placement new :

```
1 T* p = new T; // allocation + construction
2 void* buf = operator new(sizeof(T));
3 T* q = new(buf) T; // placement new
```

45 STL Big-O Synthèse

45.1 Table de Complexites STL (Tiree des Exos 4.3/4.18)

Containeur	push_back	insert(it)	erase(it)	lookup
vector	amorti $O(1)$	$O(n)$	$O(n)$	$O(1)$
deque	amorti $O(1)$	$O(n)$	$O(n)$	$O(1)$
list	$O(1)$ iter.	$O(1)$	$O(1)$	$O(n)$
set/map	—	$O(\log n)$	$O(\log n)$	$O(\log n)$
unordered_map	—	amorti $O(1)$	amorti $O(1)$	amorti $O(1)$

45.2 Ordre, itérateurs et coût amorti

Conteneur	Ordre	Itérateurs valides	Coût amorti
vector	insertion	invalidés si réallocation/erase	push_back $O(1)$
deque	insertion	reallocation invalide tout	push_front/back $O(1)$
list	insertion	stables sauf effacé	insert/erase $O(1)$
set/map	trié	stables sauf effacé	insert/erase $O(\log n)$
unordered_map	aucun	rehash invalide tout	insert $O(1)$ amorti
unordered_set	aucun	rehash invalide tout	insert $O(1)$ amorti
priority queue	tas max	n/a	push/pop $O(\log n)$

45.3 Détails Critiques

- vector : Réallocation complète lors du dépassement de capacité
- deque : Efficace aux deux extremités mais pas au milieu
- list : Pas d'accès aleatoire, iteration sequentielle uniquement
- set/map : Base sur arbre équilibré (Red-Black Tree)
- unordered_map : Table de hachage, performance degradée si mauvaise fonction de hash

45.4 std : :priority_queue

Tas max par défaut, min-heap via `std::greater<T>`

push/pop : $O(\log n)$, top : $O(1)$

46 Memo Taille-Memoire

46.1 Calcul Memoire Structures

```
1 // Exemple calcul memoire
2 vector<array<size_t, N>> v(M);
3 // = M * N * sizeof(size_t) + overhead vector (~24 B sur 64-bit)
4
5 // Attention: aligement & allocateur peuvent arrondir vers le haut !
```

Points clés :

- sizeof(size_t) = 8 bytes sur architecture 64-bit
- Overhead des containers STL (pointeurs, taille, capacité)
- Aligement memoire peut ajouter du padding

46.2 Tailles Memoire STL (64-bit)

Containeur	Taille (octets)
std::vector<size_t>	24
std::array<size_t, 0>	1
std::list<size_t>	24
std::forward_list<size_t>	8
std::deque<size_t>	8
std::set<size_t>	48
std::map<size_t>	48

47 Pieges d'Iterateurs & Splice

47.1 Invalidation d'Iterateurs

Boucles infinies courantes :

```
1 // ERREUR : oublier it = lst.erase(it)
2 for (auto it = lst.begin(); it != lst.end(); ++it) {
3     if (condition) {
4         lst.erase(it); // DANGER: it invalide !
5         // Solution: it = lst.erase(it); puis continue;
6     }
7 }
8
9 // CORRECT :
10 for (auto it = lst.begin(); it != lst.end(); ) {
11     if (condition) {
12         it = lst.erase(it); // erase retourne l'iterateur suivant
13     } else {
14         ++it;
15     }
16 }
```

47.2 Splice Operations

```
1 // list.splice_after(pos, src) : deplace apres pos
2 // Invalide seulement les pointeurs vers elements extraits
3 // PAS les iterateurs vers le reste de la liste
4
5 list<int> l1 = {1, 2, 3, 4};
6 list<int> l2 = {10, 20};
7 auto it = l1.begin(); ++it; // pointe vers 2
8
9 l1.splice(it, l2); // l1 = {1, 10, 20, 2, 3, 4}
10 // it pointe toujours vers 2 (valide !)
```

47.3 Regles d'Invalidation

Containeur	insert()	erase()
vector	Tous si reallocation	Tous apres position
deque	Tous	Tous
list	Aucun	Seulement element efface
set/map	Aucun	Seulement element efface

48 Exemples Pratiques STL

48.1 std : :vector - Tableau Dynamique

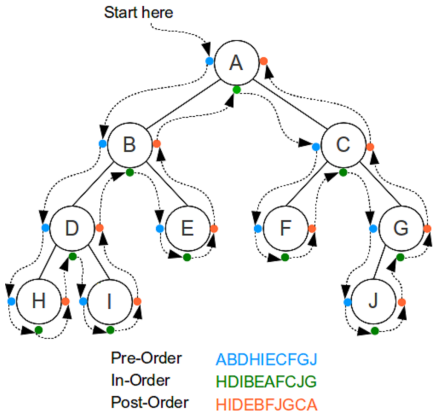
```
1 include <vector>include <algorithm>
2
3 vector<int> v = {1, 2, 3};
4 v.push_back(4); // [1,2,3,4] - 0(1) amortized
5 v.insert(v.begin()+1, 5); // [1,5,2,3,4] - 0(n)
6 v.erase(v.begin()+2); // [1,5,3,4] - 0(n)
7
8 // Acces rapide
9 int val = v[2]; // 0(1) - pas de verification
10 int safe = v.at(2); // 0(1) - avec verification bounds
11
12 // Tri et recherche
13 sort(v.begin(), v.end()); // [1,3,4,5]
14 auto it = lower_bound(v.begin(), v.end(), 4); // 0(log n)
15
16 // Capacité et reserve
17 v.reserve(100); // Pre-alloue pour 100 elements
18 cout << v.size() << " " << v.capacity(); // Taille vs capacité
```

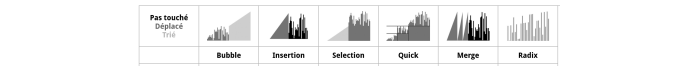
Recherche dichotomique

```
1 auto it1 = std::lower_bound(v.begin(), v.end(), x); // 1er >= x
2 auto it2 = std::upper_bound(v.begin(), v.end(), x); // 1er > x
```

Complexité : $O(\log n)$.

48.2 std : :deque - File Double





Pour calculer la taille mémoire utilisée par une structure, l'approche peut être résumée en la formule suivante :

$$\text{mem_par_struct} + \text{nombre_elements} * (\text{mem_par_elem} + \text{taille_dun_elem})$$

où les valeurs de `mem_par_struct` et `mem_par_elem` sont les suivantes (comme nous l'avons vu en cours) :

	mem_par_struct	mem_par_elem
vector	3p	0
list	3p	2p
forward_list	1p	1p
array	0	0

Pour ce qui est de ce `taille_dun_elem`, il pourra être calculé de la même manière, avec la même formule, s'il s'agit d'une autre structure imbriquée, ou bien il correspondra simplement à la taille mémoire de la valeur stockée (par exemple 4 octets si c'est un `int32_t`).

Donc par exemple, pour la structure suivante :

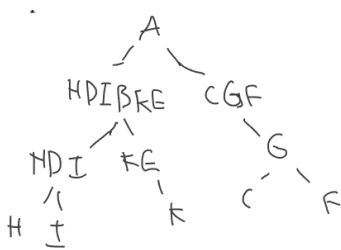
```
Plain Text
// un vecteur de M listes, chacune de N uint32_t (4 octets)
vector<list<uint32>> v(M);
for (unsigned int i = 0; i < M; ++i) {
    std::forward_list<size_t> l;
    l.assign(N, i);
    c4.push_back(l);
}
```

Donc par exemple, pour la structure suivante :

```
Plain Text
// un vecteur de M listes, chacune de N uint32_t (4 octets)
vector<list<uint32>> v(M);
for (unsigned int i = 0; i < M; ++i) {
    std::forward_list<size_t> l;
    l.assign(N, i);
    c4.push_back(l);
}
```

La taille mémoire pourra être calculée comme suit :

```
Plain Text
mem_par_s + nombre_el * (mem_par_el + taille_el) <-- taille du vector
3p + M * (0 + taille_el)
=
(mem_par_s + nombre_el * (mem_par_el + taille_el)) <-- taille de l
( 3p + N * (2p + taille_el))
=
4 <-- taille de uint32_t
p == sizeof(void*) == sizeof(size_t) = 8 octets sur système 64bit
```



Résumé de l'approche de Dijkstra (Shunting-Yard) pour évaluer une expression et la convertir en notation polonaise inverse (RPN)

1. Principe général

Dijkstra propose d'utiliser deux structures :

- Une **pile de valeurs** (opérandes),
- Une **pile d'opérateurs** (symboles arithmétiques et parenthèses).

L'idée est de parcourir l'expression token par token (nombre, opérateur, parenthèse) :

- Les **nombre**s (ou variables) sont immédiatement poussés sur la **pile de valeurs**.
- Les **opérateurs** (+, -, *, /) sont gérés selon leur **priorité** et l'état de la **pile d'opérateurs**.
- Les **parenthèses ouvrantes** (sont toujours poussées dans la pile d'opérateurs.
- Les **parenthèses fermantes**) déclenchent le **dépilage** des opérateurs jusqu'à la parenthèse ouvrante correspondante, en transférant chaque opérateur dépilé soit sur la pile de valeurs (pour calculer un résultat) soit dans la sortie RPN.

Table de priorité/associativité

Opérateur	Priorité	Associativité
()	max	—
* /	2	gauche
+ -	1	gauche

2. Étapes détaillées

Initialisation

- Pile valeurs = vide,
- Pile opérateurs = vide,
- Sortie RPN (file ou liste) = vide.

Parcours des tokens (gauche → droite)

- Si le **token est un nombre** :
 - On l'**ajoute directement** à la pile de valeurs (et/ou à la sortie RPN si on ne calcule pas immédiatement).
- Si le **token est un opérateur O** (par ex. +, *) :
 - Tant que la pile opérateurs n'est pas vide et que le **sommet** de cette pile est un opérateur de **priorité** $\geq$ **celle de O** (et n'est pas ()) :
 - On **dépile** cet opérateur du sommet.
 - On le **sort** (on l'envoie en RPN) ou on l'**applique** immédiatement aux deux dernières valeurs de la pile de valeurs.
 - Puis on **pousse O** dans la pile opérateurs.
- Si le **token est une parenthèse ouvrante** (:
 - On la **pousse** immédiatement dans la pile opérateurs (ça marque un point d'arrêt).
- Si le **token est une parenthèse fermante**) :
 - Tant que la pile opérateurs n'est pas vide et que le sommet n'est pas (:
 - On **dépile** l'opérateur du sommet.
 - On le **transfère** (place en sortie RPN ou l'applique aux valeurs).
 - Enfin, on **dépile la parenthèse ouvrante** ((sans la conserver).

Fin de la lecture

- Une fois tous les tokens parcourus, la pile opérateurs peut encore contenir des opérateurs.
- Tant que la pile opérateurs n'est pas vide :
 - On **dépile** l'opérateur du sommet.
 - On le **transfère** en sortie RPN (ou on l'applique aux valeurs).

3. Sortie RPN et calcul

- Notation polonaise inverse (RPN)** : à chaque fois qu'un opérateur est dépilé, on l'ajoute à la **sortie RPN** (après avoir listé les deux opérandes correspondants dans l'ordre où ils ont été empilés).
- Évaluation directe** : si on souhaite calculer la valeur numérique finale, on dépile l'opérateur, on dépile ses deux opérandes de la pile valeurs, on effectue le calcul (ex. x * y), puis on **pousse** le résultat sur la pile valeurs.
- À la fin, la **pile valeurs** contient le résultat unique, et la **sortie RPN** contient l'expression sous forme postfixée.

4. Exemple de suivi de piles (schéma simplifié)

Pour l'expression :

$$(1 + (2 * (7 - ((3 * 4) / 2))))$$

- On lit (→ on le met sur la pile opérateurs.
- On lit 1 → on le met sur la pile valeurs.
- On lit + → pile opérateurs = [(, +] (car (a priorité minimale)
- On lit (→ on empile (.
- On lit 2 → on empile 2 dans **valeurs**.
- On lit * → pile opérateurs = [(, +, (, *)
- On lit (→ empiler (.
- On lit 7 → empiler 7 dans **valeurs**.
- On lit - → pile opérateurs = [(, +, (, *, (, -]
- On lit (→ empiler (.
- On lit (→ empiler (.
- On lit 3 → empiler 3.
- On lit * → pile opérateurs = [(, +, (, *, (, -, (, (, *)
- On lit 4 → empiler 4.
- On lit) → on dépile * (car avant le () → en RPN on écrit 3 4 *; on dépile (.
- On lit / → pile opérateurs = [(, +, (, *, (, -, (, /]
- On lit 2 → empiler 2.
- On lit) → dépiler / → RPN 3 4 * 2 /; dépiler (.
- On lit) → dépiler - → RPN 3 4 * 2 / 7 -; dépiler (.
- On lit) → dépiler * → RPN 2 3 4 * 2 / 7 - *; dépiler (.

- On lit) → dépiler + → RPN 1 2 3 4 * 2 / 7 - * +; dépiler (.
- Fin → vider la pile opérateurs (vide), la RPN finale est :

$$1 \ 2 \ 3 \ 4 \ * \ 2 \ / \ 7 \ - \ * \ +$$

5. À retenir pour votre résumé

- Deux piles** :
 - Valeurs** (nombres/variables),
 - Opérateurs** (+, -, *, /, (,),).
- Règles de gestion des opérateurs** :
 - Quand on lit un opérateur, on **dépile** les opérateurs du sommet de la pile opérateurs qui ont **priorité** $\geq$ celle du nouveau, avant d'**empiler** le nouveau.
 - Les (et) gèrent le blocage : (se place sur la pile,) déclenche le dépilage jusqu'à rencontrer (.
- Construction de la RPN** : à chaque dépilage d'un opérateur, on l'**ajoute** à la sortie en postfixe, ou on calcule immédiatement en appliquant l'opérateur à deux valeurs dépilées.
- Fin de traitement** : après avoir parcouru tous les tokens, on **vide** la pile opérateurs en continuant à alimenter la RPN (ou à calculer).

4. HEAP MACHIN

- Construction du tas (make_heap)**
 - Parcourir les indices $i = \lfloor (n - 2) / 2 \rfloor \downarrow 0$.
 - À chaque i, appliquer "sift-down" : comparer la feuille racine A[i] avec ses deux enfants A[2i+1] et A[2i+2], échanger avec le plus grand s'il est plus petit, et répéter jusqu'à ce que i soit feuille ou plus grand que ses enfants.
 - Après chaque descente, le sous-tas ayant pour racine i est un max-heap.
 - Au final, on obtient un max-heap global de toute la table.
- Tri par tas (sort_heap)**
 - Tant que la taille du tas heap_size > 1 :
 - Swap** (échanger) A[0] ↔ A[heap_size-1].
 - heap_size ← heap_size - 1.
 - Appliquer un **sift-down** sur A[0] pour maintenir la propriété max-heap sur les heap_size premiers éléments.
 - Le segment [heap_size ...n-1] se remplit progressivement en ordre croissant.
 - Le résultat final est la table triée intégralement.
- Exemple résumé (tableau de 8 éléments)**
 - Tableau initial : [1, 8, 3, 4, 5, 6, 7, 9]
 - make_heap** aboutit à : [9, 8, 7, 4, 5, 6, 3, 1].
 - sort_heap** (itérations k = 8→2) :

k=8 : swap 9<->1 -> [1,8,7,4,5,6,3,9], sift-down -> [8,4,7,1,5,6,3,9]

k=7 : swap 8<->3 -> [3,4,7,1,5,6,8,9], sift-down -> [7,4,6,1,5,3,8,9]

k=6 : swap 7<->3 -> [3,4,6,1,5,7,8,9], sift-down -> [6,4,5,1,3,7,8,9]

k=5 : swap 6<->3 -> [3,4,5,1,6,7,8,9], sift-down -> [5,4,3,1,6,7,8,9]

k=4 : swap 5<->1 -> [1,4,3,5,6,7,8,9], sift-down -> [4,1,3,5,6,7,8,9]

k=3 : swap 4<->3 -> [3,1,4,5,6,7,8,9], sift-down -> [3,1,4,5,6,7,8,9]

k=2 : swap 3<->1 -> [1,3,4,5,6,7,8,9], sift-down -> [1,3,4,5,6,7,8,9]

- Tableau trié** : [1, 3, 4, 5, 6, 7, 8, 9].

H.1 Une Autre Section

Contenu pour cette autre section.

Algorithm 1 Parcours Pré-ordre (Root, Left, Right)

- ```
1: fonction PREORDER(n)
2: si n ≠ NIL alors
3: visit(n)
4: PREORDER(n.gauche)
5: PREORDER(n.droite)
6: fin si
7: fin fonction
```

### Algorithm 2 Parcours In-ordre (Symétrique — Left, Root, Right)

- ```
1: fonction INORDER(n)
2:   si n ≠ NIL alors
3:     INORDER(n.gauche)
4:     visit(n)
5:     INORDER(n.droite)
6:   fin si
7: fin fonction
```

Algorithm 3 Parcours Post-ordre (Left, Right, Root)

```
1: fonction POSTORDER(n)
2:   si n ≠ NIL alors
3:     POSTORDER(n.gauche)
4:     POSTORDER(n.droite)
5:   visit(n)
6:   fin si
7: fin fonction
```

I Algorithmes Complémentaires

```
1 // Priorites des operateurs
2 int precedence(char op) {
3   if (op == '+' || op == '-' || op == '*' || op == '/') return 1;
4   if (op == '^') return 2;
5   return 0;
6 }
7
8 vector<string> shuntingYard(const string& expression) {
9   stack<char> operators;
10  vector<string> output;
11
12  for (char token : expression) {
13    if (isdigit(token)) {
14      output.push_back(string(1, token));
15    }
16    else if (token == '(') {
17      operators.push(token);
18    }
19    else if (token == ')') {
20      while (!operators.empty() && operators.top() != '(') {
21        output.push_back(string(1, operators.top()));
22        operators.pop();
23      }
24      operators.pop(); // Supprimer '('
25    }
26    else if (token == '+' || token == '-' ||
27             token == '*' || token == '/') {
28      while (!operators.empty() &&
29            operators.top() != '(' &&
30            precedence(operators.top()) >= precedence(token)) {
31        output.push_back(string(1, operators.top()));
32        operators.pop();
33      }
34      operators.push(token);
35    }
36  }
37
38  // Vider pile operateurs
39  while (!operators.empty()) {
40    output.push_back(string(1, operators.top()));
41    operators.pop();
42  }
43
44  return output;
45 }
```

Listing 1 – Shunting-Yard Algorithm

Complexite : $O(n)$ temps, $O(n)$ espace

1.1 Exemple de Trace

Expression : $(1 + (2 * (7 - ((3 * 4) / 2)))$

Token	Pile Op.	Pile Val.	Sortie RPN
(	[(]	[]	[]
1	[(1]	[1]	[1]
+	[(1, +]	[1]	[1]
(	[((, +, []]	[1]	[1]
2	[((, +, [2]]	[1, 2]	[1, 2]
*	[((, +, (, *]	[1, 2]	[1, 2]
...	...	...	...

Resultat final : 1 2 3 4 * 2 / 7 - * +

```
1 void insertionSort(vector<int>& arr) {
2   int n = arr.size();
3
4   for (int i = 1; i < n; i++) {
5     int key = arr[i];
6     int j = i - 1;
7
8     // Decaler elements plus grands que key
9     while (j >= 0 && arr[j] > key) {
10      arr[j + 1] = arr[j];
11      j--;
12    }
13
14    arr[j + 1] = key;
15  }
16 }
```

Listing 2 – Insertion Sort

1.2 Tri par Selection

```
1 void selectionSort(vector<int>& arr) {
2   int n = arr.size();
3
4   for (int i = 0; i < n - 1; i++) {
5     int minIndex = i;
6
7     // Trouver minimum dans partie non trie
8     for (int j = i + 1; j < n; j++) {
9       if (arr[j] < arr[minIndex]) {
10        minIndex = j;
11      }
12    }
13
14    // Echanger avec position courante
15    swap(arr[i], arr[minIndex]);
16  }
17 }
```

Listing 3 – Selection Sort

1.3 Tri à Bulles

```
1 void bubbleSort(vector<int>& arr) {
2   int n = arr.size();
3
4   for (int i = 0; i < n - 1; i++) {
5     bool swapped = false;
6
7     for (int j = 0; j < n - i - 1; j++) {
8       if (arr[j] > arr[j + 1]) {
9         swap(arr[j], arr[j + 1]);
10        swapped = true;
11      }
12    }
13
14    // Optimisation : si pas d'echange, liste trie
15    if (!swapped) break;
16  }
17 }
```

Listing 4 – Bubble Sort

```
1 void selectionSort(vector<int>& arr) {
2   int n = arr.size();
3
4   for (int i = 0; i < n - 1; i++) {
5     int minIndex = i;
6
7     // Trouver minimum dans partie non trie
8     for (int j = i + 1; j < n; j++) {
9       if (arr[j] < arr[minIndex]) {
10        minIndex = j;
11      }
12    }
13
14    // Echanger avec position courante
15    swap(arr[i], arr[minIndex]);
16  }
17 }
```

Listing 5 – Selection Sort

```
1 void bubbleSort(vector<int>& arr) {
2   int n = arr.size();
3
4   for (int i = 0; i < n - 1; i++) {
5     bool swapped = false;
6
7     for (int j = 0; j < n - i - 1; j++) {
8       if (arr[j] > arr[j + 1]) {
9         swap(arr[j], arr[j + 1]);
10        swapped = true;
11      }
12    }
13
14    // Optimisation : si pas d'echange, liste trie
15    if (!swapped) break;
16  }
17 }
```

Listing 6 – Bubble Sort

```
1 void hanoi(int n, char from, char to, char aux) {
2   if (n == 1) {
3     cout << "Deplacer disque de " << from << " vers " << to << endl;
4     return;
5   }
6
7   // Etape 1 : deplacer n-1 disques vers auxiliaire
8   hanoi(n - 1, from, aux, to);
9
10  // Etape 2 : deplacer le plus gros disque
11  cout << "Deplacer disque de " << from << " vers " << to << endl;
12
13  // Etape 3 : deplacer n-1 disques vers destination
14  hanoi(n - 1, aux, to, from);
15 }
```

Listing 7 – Towers of Hanoi

```
1 bool search(TreeNode* root, int key) {
2   if (root == nullptr) return false;
3
4   if (key == root->data) return true;
5
6   if (key < root->data) {
7     return search(root->left, key);
8   } else {
9     return search(root->right, key);
10  }
11 }
```

Listing 8 – BST Search

```
1 TreeNode* insert(TreeNode* root, int key) {
2   if (root == nullptr) {
3     return new TreeNode(key);
4   }
5
6   if (key < root->data) {
7     root->left = insert(root->left, key);
8   } else if (key > root->data) {
9     root->right = insert(root->right, key);
10  }
11  // key == root->data : ignorer doublons
12
13  return root;
14 }
```

Listing 9 – BST Insertion

```
1 TreeNode* deleteMin(TreeNode* root) {
2   if (root == nullptr) return root;
3
4   if (root->left == nullptr) {
5     TreeNode* temp = root->right;
6     delete root;
7     return temp;
8   }
9
10  root->left = deleteMin(root->left);
11  return root;
12 }
```

Listing 10 – BST Delete Min

```
1 TreeNode* deleteNode(TreeNode* root, int key) {
2   if (root == nullptr) return root;
3
4   if (key < root->data) {
5     root->left = deleteNode(root->left, key);
6   } else if (key > root->data) {
7     root->right = deleteNode(root->right, key);
8   } else {
9     // Noeud a supprimer trouve
10    if (root->left == nullptr) {
11      TreeNode* temp = root->right;
12      delete root;
13      return temp;
14    } else if (root->right == nullptr) {
15      TreeNode* temp = root->left;
16      delete root;
17      return temp;
18    }
19
20    // Noeud avec 2 enfants : trouver successeur
21    TreeNode* temp = findMin(root->right);
22    root->data = temp->data;
23    root->right = deleteNode(root->right, temp->data);
24  }
25
26  return root;
27 }
```

Listing 11 – BST Delete

```
1 AVLNode* insertAVL(AVLNode* root, int key) {
2   // 1. Insertion BST normale
3   if (root == nullptr) {
4     return new AVLNode(key);
5   }
6
7   if (key < root->data) {
8     root->left = insertAVL(root->left, key);
9   } else if (key > root->data) {
10    root->right = insertAVL(root->right, key);
11  } else {
12    return root; // Doublons ignores
13  }
14
15  // 2. Mettre a jour hauteur
16  root->height = 1 + max(getHeight(root->left),
17                        getHeight(root->right));
18
19  // 3. Calculer balance
20  int balance = getBalance(root);
21
22  // 4. Cas de desequilibre
23  // Cas Gauche-Gauche
24  if (balance > 1 && key < root->left->data) {
25    return rightRotate(root);
26  }
```

```
26 }
27
28 // Cas Droite-Droite
29 if (balance < -1 && key > root->right->data) {
30     return leftRotate(root);
31 }
32
33 // Cas Gauche-Droite
34 if (balance > 1 && key > root->left->data) {
35     root->left = leftRotate(root->left);
36     return rightRotate(root);
37 }
38
39 // Cas Droite-Gauche
40 if (balance < -1 && key < root->right->data) {
41     root->right = rightRotate(root->right);
42     return leftRotate(root);
43 }
44
45 return root;
46 }
```

Listing 12 – AVL Insertion

J Complexites des Conteneurs STL

J.1 Conteneurs Sequentiels

Operation	vector	deque	list	forward_list
Acces aleatoire	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Insertion debut	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Insertion fin	$O(1)^*$	$O(1)$	$O(1)$	-
Insertion milieu	$O(n)$	$O(n)$	$O(1)^{**}$	$O(1)^{**}$
Suppression debut	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Suppression fin	$O(1)$	$O(1)$	$O(1)$	-

* Amorti, ** Si iterateur fourni

J.2 Conteneurs Associatifs

Operation	set/map	multiset/multimap	unordered_set/map
Recherche	$O(\log n)$	$O(\log n)$	$O(1)$ moyen, $O(n)$ pire
Insertion	$O(\log n)$	$O(\log n)$	$O(1)$ moyen, $O(n)$ pire
Suppression	$O(\log n)$	$O(\log n)$	$O(1)$ moyen, $O(n)$ pire

J.3 Adaptateurs de Conteneurs

- `std::stack<T>`
 - `push(x)`, `pop()`, `top()` : $O(1)$
 - `empty()`, `size()` : $O(1)$
- `std::queue<T>`
 - `push(x)`, `pop()`, `front()`, `back()` : $O(1)$
 - `empty()`, `size()` : $O(1)$
- `std::priority_queue<T>`
 - `push(x)`, `pop()` : $O(\log n)$
 - `top()` : $O(1)$
 - `empty()`, `size()` : $O(1)$

J.4 Algorithmes STL Principaux

- Recherche
 - `std::find(begin, end, val)` : $O(n)$
 - `std::binary_search(begin, end, val)` : $O(\log n)$ (trie requis)
 - `std::lower_bound/upper_bound` : $O(\log n)$ (trie requis)
- Tri
 - `std::sort(begin, end)` : $O(n \log n)$ (Introsort)
 - `std::stable_sort(begin, end)` : $O(n \log n)$ (stable)
 - `std::partial_sort(begin, mid, end)` : $O(n \log k)$
 - `std::nth_element(begin, nth, end)` : $O(n)$ moyen
- Heap Operations
 - `std::make_heap(begin, end)` : $O(n)$
 - `std::push_heap/pop_heap` : $O(\log n)$
 - `std::sort_heap(begin, end)` : $O(n \log n)$