Algorithmique — Cours, Exercices, TP

Auteur : Antsa Raniriamanjaka

24 avril 2025

Table des matières

Première partie

Les fondations de l'algorithme

Chapitre 1

Algorithmes et programmes : notions fondamentales

« Un programme n'est qu'un algorithme écrit dans une langue que la machine comprend. »
— *Donald E. Knuth*

Objectifs du chapitre

- Distinguer clairement *algorithme*, *programme* et *implémentation*.
- Savoir décrire un algorithme simple en pseudo-code.
- Comprendre pourquoi il est utile de mesurer le temps et la mémoire.
- S'exercer sur des problèmes très élémentaires (tri, PGCD, factorielle).

1.1 Pourquoi parler d'algorithmes ?

Un **algorithme** est à l'informatique ce qu'une recette est à la cuisine : un ensemble d'étapes précises qui transforment des *ingrédients* (les données d'entrée) en un *plat fini* (le résultat). Derrière chaque application — banque en ligne, réseau social, GPS, moteur de recherche — se cachent des algorithmes. Étudier leur conception permet :

1. de **formaliser** la résolution de problèmes ;
2. de **prouver** qu'une méthode fonctionne toujours ;
3. d'estimer le **temps** et la **mémoire** nécessaires ;
4. d'améliorer continuellement l'efficacité des logiciels.

Question de réflexion

Citez trois activités quotidiennes (hors informatique) qui suivent déjà un algorithme implicite. Que se passerait-il si les étapes étaient exécutées dans le désordre ?

1.2 Vocabulaire de base

1.2.1 Algorithme

Suite finie d'**instructions non ambiguës** exécutées dans un ordre bien défini. Chaque pas doit être si clair qu'une personne (ou une machine) ne peut l'interpréter de deux façons différentes.

1.2.2 Programme

Traduction d'un algorithme dans un **langage** que l'ordinateur comprend (C++, Python, Java. . .). Le compilateur ou l'interpréteur sert de pont entre nos idées et la machine.

1.2.3 Spécification et implémentation

— **Spécification** : « que doit faire le programme ? » (ex. « trier les nombres par ordre croissant »).
— **Implémentation** : « comment va-t-il s'y prendre ? » (ex. *méthode de tri par insertion*).

Séparer les deux évite de se perdre dans les détails trop tôt.

Exercice 1.1 – Spécification ou implémentation ?

Pour chaque phrase, cochez S (spécification) ou I (implémentation) et justifiez votre choix :

a) « Trouver le plus grand nombre dans une liste. »

b) « Parcourir la liste et mémoriser la valeur maximale rencontrée. »

c) « Chiffrer un message selon la clé fournie. »

d) « Pour chaque caractère, appliquer un décalage de trois positions dans l'alphabet (chiffrement de César). »

1.3 Premier exemple : le plus grand commun diviseur

Nous allons calculer le **PGCD** de deux entiers grâce à l'algorithme millénaire d'Euclide. Pourquoi ce choix ? Il est court, toujours correct et ses performances se mesurent facilement.

1.3.1 Pseudo-code pas à pas

```
Entree : deux entiers strictement positifs a, b (a ≥ b)
Sortie : g = pgcd(a, b)

g ← a, h ← b
Tant que h ≠ 0 faire
    r ← g mod h   // reste de la division de g par h
    g ← h
    h ← r
Retourner g
```

Listing 1.1 – Algorithme d'Euclide (version itérative)

Lecture ligne à ligne

— g contient la valeur courante du PGCD présumé.
— Tant que le reste h n'est pas nul, on poursuit la division.
— Quand h vaut 0, la dernière valeur non nulle de g est le PGCD.

1.3.2 Implémentation C++ minimale

```cpp
#include <iostream>
#include <cstdint>

std::uint64_t pgcd(std::uint64_t a, std::uint64_t b) {
    while (b != 0) {
        auto r = a % b;
        a = b;
        b = r;
    }
    return a;
}

int main() {
    std::uint64_t x, y;
    std::cin >> x >> y;
    std::cout << pgcd(x, y) << "\n";
}
```

Listing 1.2 – euclid.cpp

1.3.3 Et si on mesurait le coût ?

À chaque tour de boucle, on fait une division euclidienne. Le mathématicien Gabriel Lamé a montré qu'il suffit de $5 \log_{10}(b)$ itérations au plus lorsque b est le plus petit des deux nombres. Le temps d'exécution est donc proportionnel à $\log \min(a, b)$, noté $O(\log n)$.

Question de réflexion

Essayez manuellement l'algorithme pour $a = 48$, $b = 18$. Combien d'itérations obtenez-vous ?

1.4 Premiers pas vers l'analyse de complexité

1.4.1 Temps (nombre d'opérations)

On compte *combien* d'étapes élémentaires (addition, comparaison, etc.) le programme effectue en fonction de la taille de l'entrée (n). Cette mesure s'appelle la *complexité temporelle*. On utilise souvent la notation O (« grand O ») pour donner une borne supérieure grossière.

1.4.2 Mémoire (espace occupé)

Même idée : combien de cases supplémentaires devons-nous réserver ? Pour l'algorithme d'Euclide, on se contente de trois variables entières ; l'espace est *constant* ($O(1)$).

Exercice 1.2 – Itératif vs récursif

a) Donnez une version *récursive* du calcul de la factorielle $n!$.
b) Donnez la version *itérative*.
c) Comparez les deux en temps et en espace (pile d'appels).

1.5 Mini-projet guidé

Travail pratique 1.1 – De l'algorithme au programme

But : créer une mini-bibliothèque C++ nommée `arith` contenant `pgcd`, `ppcm` et `factorielle`, puis en vérifier la correction par des tests unitaires (Catch2).
Étapes proposées :

1. Écrire un fichier d'en-tête `arith.hpp` (déclarations).
2. Implémenter dans `arith.cpp`.
3. Configurer un `CMakeLists.txt` minimal pour la compilation.
4. Rédiger des tests couvrant : cas triviaux (0, 1), cas usuels, cas « limites » (valeurs proches de la capacité d'un `uint64_t`).
5. Mesurer les temps moyens d'exécution pour des entrées de plus en plus grandes à l'aide de `std::chrono`.

À retenir

— Un **algorithme** = recette finie et non ambiguë.
— Un **programme** = algorithme + langage + machine.
— Spécification (« quoi ? ») ≠ implémentation (« comment ? »).
— Avant d'optimiser, on mesure : temps $O(\cdot)$ et mémoire.

Chapitre 2

Outils mathématiques pour l'analyse

« Les mathématiques sont la grammaire de la science. »
— *Carl Friedrich Gauss*

Objectifs du chapitre

- Poser les briques mathématiques indispensables : sommes, suites, logarithmes.
- Comprendre et *utiliser* les notations asymptotiques (O, Θ, Ω).
- Savoir démontrer une propriété par induction **et** savoir l'appliquer à un algorithme concret.
- Acquérir une méthode pour résoudre les relations de récurrence les plus fréquentes (dichotomie, tri fusion, etc.).

2.1 Pourquoi des maths dans l'informatique ?

Un algorithme n'est pas seulement un bout de code ; c'est une idée *prouvable*. Les mathématiques servent :

1. à **décrire** précisément le problème et l'algorithme ;
2. à **démontrer** sa correction (le résultat est toujours bon) ;
3. à **quantifier** ses ressources (temps, mémoire, bande passante, énergie). [1]

Question de réflexion

Pensez à un jeu vidéo, une application bancaire et un réseau social. Où, selon vous, se cachent les calculs mathématiques ? (Indices : trajectoires de personnages, chiffrement RSA / ECC, moteurs de recommandation.)

2.2 Sommes, suites et notations de base

2.2.1 Suites numériques

Une *suite* $(u_n)_{n\geq 0}$ est une fonction de $\mathbb{N}$ vers $\mathbb{R}$. Elle peut être définie :

1. Voir l'introduction de [1] pour une discussion classique.

— **explicitement** : $u_n = 3n + 2$;
— **récursivement** : $u_{n+1} = 2u_n + 1$ avec $u_0 = 0$.

En pratique, on utilise une suite pour **compter** combien d'opérations $T(n)$ sont nécessaires lorsqu'on agrandit la taille de l'entrée n. Ainsi, chaque terme de la suite montre « le coût quand on ajoute un élément de plus ».

2.2.2 Sommes arithmétiques et géométriques

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \text{(Gauss : addition « en miroir »)}$$

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r} \quad (r \neq 1) \qquad \text{(raison } r \text{ ; géométrique)}$$

Application : si un tri naïf effectue $(n-1) + \cdots + 2 + 1$ comparaisons, il exécute $\frac{n(n-1)}{2} \in \Theta(n^2)$ opérations.

2.2.3 Logarithmes — rappels essentiels

Soient a et b deux réels strictement positifs :

— $\log_b a$ = exposant x tel que $b^x = a$.
— Bases usuelles : 2 (binaire), 10 (décimale), e (naturel).
— **Changement de base** : $\log_b a = \dfrac{\log_k a}{\log_k b}$.
— **Croissance lente** : $\log n \ll n^{\varepsilon}$ pour tout $\varepsilon > 0$.

Exercice 2.1 – Manipuler les logarithmes

Montrez que $\log_2(n!) \in \Theta(n \log n)$. *Indication :* utilisez la somme $\sum_{k=1}^{n} \log_2 k$ et comparez-la à une intégrale.

2.3 Principe d'induction : mode d'emploi

2.3.1 Induction simple (ou faible)

1. **Base** : vérifier $P(n_0)$.
2. **Hérédité** : supposer $P(k)$ vraie et montrer $P(k+1)$.

Si les deux étapes tiennent, $P(n)$ est vraie $\forall n \geq n_0$.

2.3.2 Induction forte

Hypothèse plus large : on suppose $P(n_0), \dots, P(k)$ pour démontrer $P(k+1)$. Utile quand la valeur courante dépend de *plusieurs* valeurs précédentes (ex. suite de Fibonacci).

Exercice 2.2 – Somme des premiers entiers

Prouver que $S(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Solution

— *Base* ($n = 1$) : $S(1) = 1$ et $\frac{1(1+1)}{2} = 1$; ok.

— *Induction* : supposons la formule vraie pour $n = k$. Alors $S(k+1) = S(k) + (k+1) = \frac{k(k+1)}{2} + k + 1$. On factorise : $= \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$; formule vraie pour $k+1$.

Exercice 2.3 – Induction et recherche dichotomique

1. Commentez chaque étape du pseudo-code suivant :

```
RechercheDicho(A[0..n-1], x)
    gauche ← 0, droite ← n-1
    Tant que gauche ≤ droite faire
        m ← ⌊(gauche+droite)/2⌋
        Si A[m] = x alors retourner m
        Sinon si A[m] < x alors gauche ← m+1
                           sinon droite ← m-1
    retourner -1                      // absent
```

2. Montrez par induction forte sur la taille du sous-tableau que l'algorithme renvoie toujours l'indice de x s'il existe, et -1 sinon.

2.4 Relations de récurrence

Dès qu'un algorithme se définit *sur lui-même* — via un appel récursif ou une division en sous-problèmes — son coût obéit à une **relation de récurrence**. Résoudre cette équation, c'est transformer une description *locale* (« coût d'un appel ») en une formule *globale* $T(n)$ qui vaut pour toute taille d'entrée.

2.4.1 Méthodes classiques de résolution

Déroulement (ou *iteration method*) On *déplie* la récurrence plusieurs fois ; un motif apparaît (somme arithmétique, géométrique...), que l'on additionne jusqu'à atteindre le cas de base $T(1)$.

Substitution

1. Formuler une *conjecture* raisonnable $f(n)$.

2. Prouver par induction (souvent forte) que $T(n) \le f(n)$ ou $T(n) = f(n)$.

Arbre de récursion On dessine les appels comme un arbre : chaque niveau contient le coût cumulé des sous-problèmes. La somme des niveaux donne $T(n)$ et révèle visuellement où se concentre la dépense (racine, feuilles, partout. . .).

Théorème maître (Master Theorem) Applicable aux récurrences $T(n) = aT(n/b) + f(n)$ avec $a \ge 1$, $b > 1$. Trois cas ; la comparaison entre $f(n)$ et $n^{\log_b a}$ décide du résultat (voir tableau en annexe A).

2.4.2 Exemple — Coût de la recherche dichotomique

Récurrence

$$T(n) = T\big(\tfrac{n}{2}\big) + 1, \qquad T(1) = 1.$$

a) **Déroulement**

$$\begin{aligned} T(n) &= T\big(\tfrac{n}{2}\big) + 1 \\ &= T\big(\tfrac{n}{4}\big) + 1 + 1 \\ &\;\vdots \\ &= T(1) + \underbrace{1 + \cdots + 1}_{k \text{ fois}} \end{aligned}$$

où $n/2^k = 1 \Rightarrow k = \log_2 n$. $\Rightarrow T(n) = 1 + \log_2 n \in \Theta(\log n)$.

b) **Substitution rapide** Hypothèse : $T(n) \le c \log n$. $T(n) = T(n/2) + 1 \le c \log(n/2) + 1 = c(\log n - 1) + 1 \le c \log n$ dès que $c \ge 1$.

c) **Arbre de récursion** Chaque niveau coûte exactement 1 comparaison ; il y a $\log_2 n + 1$ niveaux $\Rightarrow$ même résultat.

d) **Théorème maître** $a = 1$, $b = 2$, $f(n) = 1$, $n^{\log_b a} = 1$. Cas 2 $\Rightarrow T(n) = \Theta(\log n)$.

Exercice 2.4 – Discussion

Expliquez en une phrase pourquoi $\log n$ apparaît inévitablement lorsqu'un algorithme divise la taille de l'entrée par 2 à chaque étape.

2.5 Notation asymptotique

2.5.1 O, Ω, Θ — définitions formelles

Lorsque nous évaluons un algorithme, nous voulons savoir comment son temps d'exécution ou sa consommation mémoire *croissent* avec la taille de l'entrée n. Compter le nombre *exact* d'opérations ($3n^2 + 7n - 4$, par exemple) serait trop précis : les constantes 3, 7 ou 4 dépendent du langage, du processeur, parfois même du compilateur. Ce qui nous intéresse vraiment, c'est l'**ordre de grandeur** : le coût se comporte-t-il comme une droite (n), une parabole (n^2) ou, pire, une exponentielle (2^n) ?

Les notations O, Ω et Θ forment un langage standard pour exprimer ces ordres de grandeur en « oubliant » les détails machines et les constantes multiplicatives. Elles permettent :

— de comparer rapidement deux algorithmes : $\Theta(n \log n)$ battra toujours $\Theta(n^2)$ pour des entrées suffisamment grandes ;
— d'estimer si un problème reste solvable quand l'entrée passe de quelques milliers à plusieurs millions d'éléments ;
— de raisonner sur la faisabilité *avant* d'écrire la moindre ligne de code.

En pratique, la notation asymptotique est donc la *boussole* qui guide nos choix d'algorithmes et d'optimisations.

Borne supérieure O : $f(n) \in O\big(g(n)\big) \iff \exists\, C > 0,\ \exists\, n_0 \in \mathbb{N},\ \forall n \geq n_0,\ f(n) \leq C\, g(n).$
Borne inférieure Ω : $f(n) \in \Omega\big(g(n)\big) \iff \exists\, c > 0,\ \exists\, n_0,\ \forall n \geq n_0,\ f(n) \geq c\, g(n).$
Même ordre Θ : $f(n) \in \Theta\big(g(n)\big) \iff f \in O(g)\ et\ f \in \Omega(g)$. Autrement dit, f est *à la fois* bornée au-dessus et au-dessous par g à multiplicateur constant près.

Remarque pratique. Lorsqu'on écrit qu'un algorithme tourne en $O(n \log n)$, on déclare qu'il existe *une* constante C telle que, pour une entrée assez grande, son temps d'exécution ne dépasse jamais $C\, n \log n$. Les détails machine-dépendants (vitesse CPU, langage, etc.) sont absorbés dans ce C.

Exercice 2.5 – Comparer des croissances

Classez les fonctions suivantes du plus lent au plus rapide : $\log n$, $n^{0.5}$, $n \log n$, 2^n, $n!$, n^2.

2.6 Probabilités élémentaires pour l'analyse d'algorithmes

2.6.1 Pourquoi la probabilité en algorithmique ?

Nombre d'algorithmes modernes (Quicksort randomisé, algorithmes de Monte-Carlo, protocoles réseau) prennent des décisions au hasard ; leur *coût* n'est plus déterministe. Les outils probabilistes permettent alors de :

— décrire la « loi » des temps d'exécution possibles ;
— calculer une **espérance** (coût moyen) et parfois la probabilité de dépasser un seuil critique ;
— prouver qu'un algorithme est *rapide avec grande probabilité*.

2.6.2 Vocabulaire et axiomes de base

— **Univers Ω** : ensemble de tous les résultats possibles d'une expérience aléatoire (ex. lancer un dé : $\{1, \ldots, 6\}$).
— **Événement $A \subseteq \Omega$** : sous-ensemble d'issues.
— **Probabilité $\mathbb{P}[A]$** : fonction telle que $\mathbb{P}\big[\Omega\big] = 1$ et, si A, B disjoints, $\mathbb{P}[A \cup B] = \mathbb{P}[A] + \mathbb{P}[B]$.

2.6.3 Espérance et variables indicatrices

Variable aléatoire fonction $X : \Omega \to \mathbb{R}$ (ex. valeur obtenue sur un dé).

Espérance $\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega)\, \mathbb{P}[\{\omega\}]$.

Linéarité pour *toutes* X, Y : $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ même si X et Y sont dépendantes.

Indicateur $\mathbf{1}_A(\omega) = 1$ si $\omega \in A$, sinon 0. Utile pour compter : si $X = \sum_i \mathbf{1}_{A_i}$ alors $\mathbb{E}[X] = \sum_i \mathbb{P}[A_i]$.

2.6.4 Petit exemple classique

Combien de lancers de pièce faut-il *en moyenne* avant d'obtenir pile ?

$$\mathbb{E}[X] = p \cdot 1 + (1-p)p \cdot 2 + (1-p)^2 p \cdot 3 + \cdots = \frac{1}{p}, \quad p = \tfrac{1}{2} \Rightarrow \mathbb{E}[X] = 2.$$

2.6.5 Application : coût moyen d'une boucle aléatoire

Supposons une boucle qui se répète tant qu'un événement de probabilité $p = \frac{1}{2}$ ne se produit pas ; le travail à chaque itération vaut α. Son coût moyen est $\mathbb{E}[T] = \alpha\, \mathbb{E}[X] = 2\alpha$.

Exercice 2.6 – Taille d'un préfixe aléatoire

On parcourt un tableau jusqu'à rencontrer la première valeur négative. Chaque cellule est négative avec probabilité $q = \frac{1}{4}$, indépendamment des autres. Calculez l'espérance du nombre de lectures effectuées.

Question de réflexion

Pourquoi la linéarité de l'espérance est-elle particulièrement précieuse quand les variables sont dépendantes ? Donne un exemple tiré d'un algorithme où cette propriété simplifie drastiquement le calcul.

À retenir

- **Sommes et logarithmes** : les briques de base pour compter.
- **Induction** : la preuve standard pour algorithmes et formules.
- **Récurrences** : quatre outils (déroulement, substitution, arbre, maître).
- **Notation asymptotique** : comparer les croissances sans se perdre dans les constantes.
- **Espérance** : mesurer le coût moyen des algorithmes randomisés.

2.7 Exercices

Exercice 2.7 – Somme arithmétique

Calculez, par induction simple, la valeur de la somme $S(n) = \sum_{i=1}^{n}(2i-1)$ et vérifiez que $S(n)$ est égale à n^2.

Exercice 2.8 – Manipuler les logarithmes

Montrez que pour tout $n \geq 1$,

$$\log_2(n!) \;=\; \Theta\bigl(n \log n\bigr).$$

(On pourra encadrer la somme $\sum_{k=1}^{n} \log_2 k$ par une intégrale.)

Exercice 2.9 – Induction forte et nombres de Fibonacci

Prouvez, par induction forte, que pour tout $n \geq 0$, le n-ième nombre de Fibonacci vérifie $F_n \leq 2^n$.

Exercice 2.10 – Récurrence linéaire

Soit $T(n) = 3T(n-1) + 2$ avec $T(0) = 4$.

a) Trouvez la forme fermée de $T(n)$ par déroulement.

b) Donnez l'ordre de grandeur asymptotique de $T(n)$.

Exercice 2.11 – Complexité espérée d'un algorithme aléatoire

Un algorithme répète une opération coûtant une unité tant que un événement de probabilité $p = \frac{1}{3}$ ne se produit pas. Quel est le coût moyen de l'algorithme ?

Exercice 2.12 – InsertionSort — formuler la récurrence

Algorithme

```
InsertionSort(A[0..n-1]):
    pour i ← 1 a n-1:
        cle ← A[i]; j ← i-1
        tant que j ≥ 0 et A[j] > cle:
            A[j+1] ← A[j]; j ← j-1
        A[j+1] ← cle
```

a) Établissez la récurrence du *pire cas* pour le coût $T(n)$.

b) Déduisez le classement de ce coût en notation O.

Chapitre 3

Structures linéaires fondamentales

« La vraie question n'est pas de savoir si les ordinateurs pensent,
mais s'ils peuvent nous aider à mieux organiser nos données. »
— *Robert Sedgewick*

Les tableaux, listes, piles et files forment la « charpente » de la plupart des algorithmes : rechercher, trier, gérer un historique, synchroniser des processus, etc. Maîtriser leurs propriétés et leur coût est indispensable avant d'aborder les graphes ou les paradigmes de conception plus avancés.

Objectifs du chapitre

- Définir clairement chaque structure et ses opérations de base.
- Mesurer les complexités en temps et en mémoire.
- Mettre en évidence les cas d'usage typiques.

3.1 Tableau statique (*array*)

3.1.1 Définition et propriétés

Un **tableau statique** (souvent appelé *array* en C/C++) est un bloc de mémoire *contiguë* réservé une seule fois : sa taille n est fixée à la compilation ou lors de son allocation et ne peut plus changer ensuite.

- **Accès aléatoire :** l'adresse de la case i se calcule par $\texttt{base} + i \times \texttt{sizeof(type)} \Rightarrow$ lecture / écriture en $O(1)$.
- **Insertion / suppression** au milieu nécessitent le décalage de tous les éléments à droite $\Rightarrow$ $O(n)$ dans le pire cas.
- **Avantages :** compacité mémoire, excellent *cache locality*, simplicité arithmétique.
- **Inconvénient :** taille rigide, coût linéaire pour réorganiser les données.

Exercice 3.1 – Accès direct

Soit un tableau `A` de 1 000 000 d'entiers (32 bits).

a) Combien d'opérations mémoire faut-il pour lire `A[723456]` ?

b) Expliquez pourquoi on affirme que le temps d'accès est *constant* même avec un tableau aussi grand.

Solution

a) Une seule : le processeur calcule l'adresse `base + 723456 × 4` octets puis lit 4 octets en un cycle de cache (idéalement).

b) La formule d'adressage dépend uniquement de i, pas de n ; elle exige un nombre fixe d'instructions arithmétiques, d'où la notation $O(1)$.

Exercice 3.2 – Tableau statique en C++ & calcul de somme

Déclarez un tableau statique de 10 entiers, initialisez-le avec les valeurs 1 à 10 puis écrivez une fonction `sumArray` qui renvoie la somme des éléments. Analysez la complexité.

Solution.

```cpp
#include <cstddef>   // std::size_t
#include <iostream>

constexpr std::size_t N = 10;
int main() {
    int A[N] = {1,2,3,4,5,6,7,8,9,10};    // declaration
                                          //   + initialisation

    auto sumArray = [](const int* arr, std::size_t n) {
        int s = 0;
        for (std::size_t i = 0; i < n; ++i) s += arr[i]; // O(n)
        return s;
    };

    std::cout << "Somme = " << sumArray(A, N) << '\n';
                                         // affiche la somme
}
```

Listing 3.1 – array_sum.cpp

— `A` est stocké sur la pile ; son adresse de base est fixe.

— La boucle effectue exactement n lectures et additions $\Rightarrow$ $O(n)$.

3.2 Tableau dynamique (vector)

Pour pallier la taille fixe d'un tableau, on utilise un **tableau dynamique** (ex. `std::vector` en C++) qui s'agrandit automatiquement.

3.2.1 Analyse amortie (*Amortised Analysis*)

L'idée fondamentale est de mesurer le *coût moyen* d'une opération sur une longue séquence d'appels, plutôt que son pire cas isolé.

Doublage de capacité. Un `std::vector` (ou tout tableau dynamique) maintient deux informations :

— `size` – nombre d'éléments actuellement stockés ;
— `capacity` – taille maximale du bloc mémoire réservé.

Tant que `size` < `capacity`, `push_back` coûte $O(1)$: on écrit simplement à l'indice courant. Lorsque `size` == `capacity`, on :

1. alloue un nouveau bloc deux fois plus grand ;
2. copie ou déplace les éléments dans le nouveau bloc ;
3. libère l'ancien bloc.

Cette opération *exceptionnelle* coûte $O(n)$, mais elle n'arrive qu'après n insertions en temps constant. Sur une séquence de m insertions, le nombre total de copies est borné par $2m \Rightarrow$ *coût moyen* par insertion = 2 soit $O(1)$ **amorti**.

Syntaxe minimale C++ (push_back)

```cpp
void push_back(const T& value) {
    if (sz_ == cap_) {              // plus de place ?
        cap_ *= 2;                  // double la capacite
        T* bigger = new T[cap_];
        std::move(data_, data_ + sz_, bigger); // copie/move O(sz_)
        delete[] data_;
        data_ = bigger;
    }
    data_[sz_++] = value;           // insertion O(1)
}
```

Listing 3.2 – push_back avec agrandissement

Ici, le coût $O(n)$ du `move` n'est payé qu'une seule fois pour les n insertions qui précèdent. L'analyse amortie garantit donc un temps $O(1)$ *en moyenne* par appel `push_back`.

Question de réflexion

Pourquoi la stratégie « doubler la taille » est-elle plus efficace qu'ajouter une seule case à chaque débordement ?

3.3 Liste chaînée

Une **liste chaînée (singly-linked list)** est un ensemble ordonné de nœuds disposés en mémoire *non contiguë*. Chaque nœud stocke :

— une `cle` (la valeur à conserver) ;
— un seul pointeur, `next`, qui indique l'adresse du nœud suivant — ou `nullptr{}` si l'on atteint la fin.

Le premier nœud est référencé par un pointeur `head` ; on accède ensuite aux autres éléments en *chaînant* les `next` les uns après les autres.

Complexité des opérations principales

Insertion en tête créer un nœud et rediriger `head` $\Rightarrow$ $O(1)$.

Suppression en tête déplacer `head` sur `head->next` $\Rightarrow$ $O(1)$.

Recherche d'une clé parcourir séquentiellement les nœuds jusqu'à trouver la valeur ou atteindre la fin $\Rightarrow$ $O(n)$ dans le pire cas.

Ainsi, la liste chaînée sacrifie l'accès direct (pas d'indice comme dans un tableau) mais excelle pour les ajouts ou retraits fréquents au tout début de la structure.

Exercice 3.3 – Parcours

Écrire une fonction C++ qui renvoie la longueur d'une liste chaînée ; analyser sa complexité.

Solution :

```cpp
#include <cstddef>   // std::size_t

struct Node {
    int   key;
    Node* next;
    explicit Node(int k, Node* n = nullptr) : key{k}, next{n} {}
};

std::size_t length(const Node* head) noexcept
{
    std::size_t n = 0;
    for (auto p = head; p != nullptr; p = p->next) ++n;  // O(n)
    return n;
}
```

Listing 3.3 – list_length.cpp

Remarque. Les listes implémentent naturellement les piles et files sans déplacement d'éléments, contrairement aux tableaux.

3.4 Pile (stack)

3.4.1 Principe LIFO (Last – In, First – Out)

Une **pile** est une structure qui n'autorise l'accès qu'à un seul extrémité : on y **empile** (avec `push`) et on y **dépile** (avec `pop`) toujours au même endroit, appelé « sommet » (`top`). Le dernier élément ajouté sera donc systématiquement le premier retiré : c'est la règle **Last-In, First-Out**. Cette contrainte rend les opérations extrêmement simples : chaque `push`, `pop` ou lecture du sommet s'effectue en temps constant $O(1)$, car il suffit de modifier un pointeur (pile implémentée par liste) ou une case en bout de tableau (pile implémentée par vecteur). Les piles servent notamment :

— à mémoriser les appels récursifs (pile d'exécution d'un programme) ;
— à parcourir un graphe en profondeur (DFS) ;
— à évaluer une expression arithmétique écrite en notation postfixée (algorithme de Dijkstra « shunting-yard »).

En résumé, la pile sacrifie l'accès direct aux éléments intermédiaires pour gagner une simplicité et une rapidité optimales sur les opérations de tête.

Question de réflexion

Comment la pile d'appels d'un programme C++ s'appuie-t-elle sur ce concept ?

3.5 File (queue)

3.5.1 Principe FIFO

Une **file** applique la discipline *First-In, First-Out* (FIFO) : enqueue ajoute en queue, dequeue retire en tête, toutes deux en $O(1)$ avec une implémentation circulaire.

Exercice 3.4 – Buffer circulaire

Implémentez un buffer circulaire de taille fixe ; démontrez que enqueue/dequeue sont $O(1)$.

Solution :

```
template<class T, std::size_t CAP>
class CircularQueue {
public:
    CircularQueue() : front_{0}, sz_{0} {}

    bool enqueue(const T& v) {                // O(1)
        if (sz_ == CAP) return false;         // plein
        std::size_t rear = (front_ + sz_) % CAP;
        buf_[rear] = v; ++sz_;
        return true;
    }
    bool dequeue() {                          // O(1)
        if (sz_ == 0) return false;           // vide
        front_ = (front_ + 1) % CAP; --sz_;
        return true;
    }
    const T& front() const { return buf_[front_]; }
    bool empty() const noexcept { return sz_ == 0; }
private:
    T           buf_[CAP];
```

```
    std::size_t front_, sz_;
};
```

Listing 3.4 – `circular_queue.hpp`

3.6 Comparatif rapide

Structure	**Accès aléatoire**	**Insertion tête**	**Recherche**
Tableau (static)	$O(1)$	$O(n)$	$O(n)$
Vecteur (dynamic)	$O(1)$	$O(n)$ amorti en fin	$O(n)$
Liste chaînée	$O(n)$	$O(1)$	$O(n)$
Pile	$O(1)$ (`top`)	$O(1)$	—
File	$O(1)$ (`front`)	$O(1)$	—

À retenir

- **Tableau statique** : accès aléatoire en $O(1)$, insertion/suppression au milieu en $O(n)$.
- **Tableau dynamique (`vector`)** : `push_back` en $O(1)$ amorti grâce au doublage de capacité, redimensionnement en $O(n)$.
- **Liste chaînée simple** : insertion et suppression en tête en $O(1)$, parcours et recherche en $O(n)$.
- **Pile (stack)** : opérations `push`, `pop` et `top` en $O(1)$ (LIFO).
- **File (queue)** : opérations `enqueue` et `dequeue` en $O(1)$ avec implémentation circulaire (FIFO).
- **Choix de structure** : privilégier le tableau ou la liste selon la fréquence des accès aléatoires vs des insertions/suppressions rapides.

Exercices

Travail pratique 3.1 – Mini STL simplifiée

Objectif. Produire une bibliothèque C++ `ministl::` contenant :

- — un vecteur dynamique (doublement de capacité) ;
- — une liste simple avec itérateur ;
- — une pile et une file basées sur la liste.

Étapes.

1. Rédiger les `.hpp` et `.cpp` séparés.
2. Ajouter des tests unitaires (Catch2) : push/pop, débordement, itérations.
3. Mesurer le temps d'un million d'insertions dans le vecteur et la liste ; interpréter les différences à la lumière de la complexité.

Exercice 3.5

Expliquez pourquoi l'accès `A[i]` dans un tableau statique est équivalent à l'arithmétique d'adresse base $+ i \times$ taille.

Exercice 3.6

Donnez un exemple concret où l'on préfère une liste à un vecteur, malgré la lenteur du parcours séquentiel.

Exercice 3.7

Calculez le coût amorti d'une stratégie de $+10\ \%$ de capacité au lieu de $\times 2$ pour un tableau dynamique.

Exercice 3.8 – Adresse et accès dans un tableau

Soit un tableau statique d'entiers `int A[100]` dont l'adresse de base est `0x1000` et dont chaque entier occupe 4 octets. Écrivez l'expression de l'adresse de `A[i]` en hexadécimal, puis expliquez pourquoi l'accès à cet élément se fait en temps constant $O(1)$.

Exercice 3.9 – Longueur d'une liste chaînée

Écrivez en C++ une fonction `std::size_t length(Node* head)` qui renvoie le nombre de nœuds d'une liste chaînée simple. Analysez rigoureusement sa complexité (en pire cas et au meilleur cas).

Exercice 3.10 – Pile avec vecteur

Vous voulez implémenter une pile LIFO en utilisant `std::vector<T>`. Décrivez les algorithmes de `push`, `pop` et `top`, puis donnez leur complexité temporelle (pire cas et amortie pour `push`).

Exercice 3.11 – Analyse amortie — +10 %

On adopte pour un tableau dynamique la stratégie suivante : lorsqu'il est plein, on augmente sa capacité de 10 % (au lieu de la doubler). Démontrez que le coût amorti d'un `push_back` reste $O(1)$ et calculez la constante approximative dans votre majoration.

Exercice 3.12 – File circulaire — complexité

On stocke les éléments dans un buffer circulaire de capacité fixe `CAP`, avec indices `head` et `tail`. Expliquez pourquoi les opérations `enqueue` et `dequeue` s'exécutent en $O(1)$ dans le pire cas (sans amortissement).

Travail pratique 3.2 – Simulation d'historique de navigation Web

Objectif : modéliser l'historique d'un navigateur avec deux piles (`back` et `forward`), et permettre les opérations :

- `visit(url)` : visiter une nouvelle URL (vide la pile `forward`).
- `back()` : revenir à la page précédente (pop/push entre piles).
- `forward()` : avancer si un retour a été effectué.
- `current()` : obtenir l'URL courante.

Étapes :

1. Déclarez une classe `BrowserHistory` avec deux `std::stack<std::string>`.
2. Implémentez chacune des opérations ci-dessus, en vérifiant les conditions de pile vide.
3. Rédigez un programme de test interactif : lecture de commandes `visit`, `back`, `forward`, affichage de `current`.
4. Mesurez la complexité de chaque opération et justifiez qu'elles sont toutes en temps constant $O(1)$.

Bibliographie

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4 edition, 2022.