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.

Bibliographie