Algorithmique — Cours, Exercices, TP

Auteur : Antsa Raniriamanjaka

24 avril 2025

Table des matières

Première partie

Les fondations de l’algorithme

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

- Comprendre la différence entre *algorithme*, *programme* et *implémentation*.
- Savoir exprimer un algorithme sous forme de pseudo-code clair.
- Introduire les premières notions de coût temporel et spatial.
- Illustrer la démarche d'analyse sur quelques exemples classiques.

1.1 Motivation : pourquoi les algorithmes ?

Les algorithmes sont les « recettes » qui transforment des données d'entrée en résultats d'intérêt. Ils :

1. offrent un **langage universel** pour décrire la résolution de problèmes ;
2. permettent de **prouver** qu'une méthode est correcte ;
3. servent de base à la **mesure de performance** que l'on optimisera ensuite ;
4. constituent un pont naturel entre *mathématiques* et *informatique*.

Question de réflexion

Peut-on imaginer un domaine scientifique ou industriel qui n'utilise aucun algorithme ? Discutez.

1.2 Définitions de base

1.2.1 Algorithme

Un *algorithme* est une **suite finie d'instructions**, non ambiguës, qui s'exécutent dans un ordre déterminé et qui, appliquées à des données appartenant à un ensemble D, produisent un résultat dans un ensemble R après un nombre fini d'étapes.

1.2.2 Programme

Un *programme* est l'**implantation** concrète d'un algorithme dans un langage (C++, Python, . . .) compréhensible par une machine.

1.2.3 Implémentation vs. Spécification

La spécification décrit *ce que* doit faire le logiciel ; l'implémentation précise *comment* le réaliser. Ces deux niveaux doivent être soigneusement dissociés.

Exercice 1.1 – Spécification ou implémentation ?

Pour chacun des énoncés suivants, dites s'il s'agit d'une spécification ou d'une implémentation et justifiez :

a) Trier un tableau d'entiers en ordre croissant.

b) Répéter `n-1` fois : balayer le tableau et échanger les valeurs des positions `i` et `i+1` si elles sont mal ordonnées.

c) Calculer $\gcd(a, b)$, le plus grand commun diviseur de deux entiers.

1.3 Premier exemple : l'algorithme d'Euclide

1.3.1 Pseudo-code

```
Entr\'ee  : 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
    g <- h
    h <- r
Retourner g
```

Listing 1.1 – Algorithme d'Euclide (version itérative)

1.3.2 Implémentation C++

```cpp
#include <iostream>
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 Analyse de complexité

Le nombre d'itérations est borné par $5 \log_{10}(b)$ (propriété de Lamé). Ainsi l'algorithme s'exécute en temps $O(\log \min(a, b))$ et en espace constante.

Question de réflexion

Pourquoi l'algorithme d'Euclide est-il « optimal » pour le PGCD sur des architectures classiques ?

1.4 Introduction aux notions de coût

1.4.1 Temps d'exécution

On compte le nombre d'opérations élémentaires en fonction de la taille des données : n (longueur du tableau, nombre de bits, . . .). L'objectif : trouver une borne asymptotique — généralement O.

1.4.2 Mémoire utilisée

Même démarche que pour le temps ; certaines optimisations échangent espace et temps.

Exercice 1.2

a) Proposez une version récursive et une version itérative de la factorielle $n!$.
b) Analysez leurs complexités (temps et espace).

1.5 Étude de cas guidée

Travail pratique 1.1 – De l'algorithme au programme

Objectif : implémenter une mini-bibliothèque C++ offrant `pgcd`, `ppcm` et `factorielle`, accompagnée de tests unitaires (Catch2).
Étapes :

1. Écrire un *header* `arith.hpp` déclarant les fonctions.
2. Créer `arith.cpp` avec les implémentations (pas de récursion inutile).
3. Configurer un `CMakeLists.txt` minimal.
4. Rédiger des cas de tests (valeurs limites : 0, 1, grands entiers).
5. Mesurer le temps avec `std::chrono` pour des entrées croissantes.

À retenir

— Un *algorithme* est une séquence finie d'étapes déterministes.
— Un *programme* implémente un algorithme dans un langage cible.
— Toute étude d'algorithme passe par la **preuve de correction** et l'**analyse de complexité**.

Bibliographie