Introduction à la complexité

Valentin Bouquet, François Delbot

valentin.bouquet@parisnanterre.fr

17 septembre 2025

Sommaire

Qu'est-ce qu'un algorithme ?

Définition

Un **algorithme** est une procédure computationnelle bien définie qui

- prend en entrée une ou plusieurs valeurs,
- produit en sortie une ou plusieurs valeurs,
- en un temps fini.

C'est une suite d'étapes logiques qui **transforme des entrées en sorties**.

Problème algorithmique

Un algorithme peut être vu comme un outil pour résoudre un problème computationnel bien spécifié :

- Le problème définit la relation attendue entre entrées et sorties.
- L'algorithme donne une procédure effective pour établir cette relation.

Exemple de problème algorithmique

Problème : Trouver le maximum dans un tableau d'entiers.

Entrée

Un tableau $A = [a_1, a_2, \ldots, a_n]$ d'entiers.

Sortie

La plus grande valeur contenue dans A.

Exemple :

$$A = [3, 7, 2, 9, 4] \quad \Rightarrow \quad \text{Sortie : } 9$$

Algorithmes et implémentations

Attention

Il y a une grande différence entre la description d'un algorithme et son implémentation. L'implémentation est réalisée dans un langage concret, tandis que la description reste une abstraction et décrit la méthode.

Qu'est-ce qu'un algorithme ?

Deux implémentations différentes, un même algorithme

```c
void tri_a_bulle(int *tab, int taille)
{
    int i,j,tmp;
    for(i=0; i<taille; i++)
        for(j=0; j<taille-1; j++)
            if(tab[j]>tab[j+1])
            {
                tmp=tab[j];
                tab[j]=tab[j+1];
                tab[j+1]=tmp;
            }
}

void tri_a_bulle_v2(int *tab, int taille)
{
    int i,j,tmp;
    for(i=0; i<taille; i++)
        for(j=0; j<taille-1; j++)
            if(tab[j]>tab[j+1])
            {
                tmp=tab[j+1];
                tab[j+1]=tab[j];
                tab[j]=tmp;
            }
}
```

Ces deux fonctions implémentent le même algorithme, mais n'ont pas la même implémentation. En effet, à chaque échange, la valeur de la variable tmp est différente.
Il s'agit bien entendu d'un exemple trivial.

Efficacité d'un algorithme

Comparer des fonctions

Nous avons vu qu'il existe différentes méthodes, différents algorithmes, pour résoudre un problème.

Fonction Fibonacci :

1. itérative
2. récursive

Quelle est la meilleure méthode ?

Implémentez les deux versions de Fibonacci et regardez combien de temps est nécessaire pour calculer la valeur fibo(45) ?

Efficacité d'un algorithme

La version itérative s'exécute instantanément (sur ma machine). La valeur du paramètre n ne semble pas avoir d'impact visible sur le temps d'exécution.

```
1   int fibo(int n)
2   {
3       int um1=1,um2=0,u,i;
4       if(n<2) return n;
5       for(i=2;i<=n;i++)
6       {
7           u=um1+um2;
8           um2=um1;
9           um1=u;
10      }
11      return u;
12  }
```

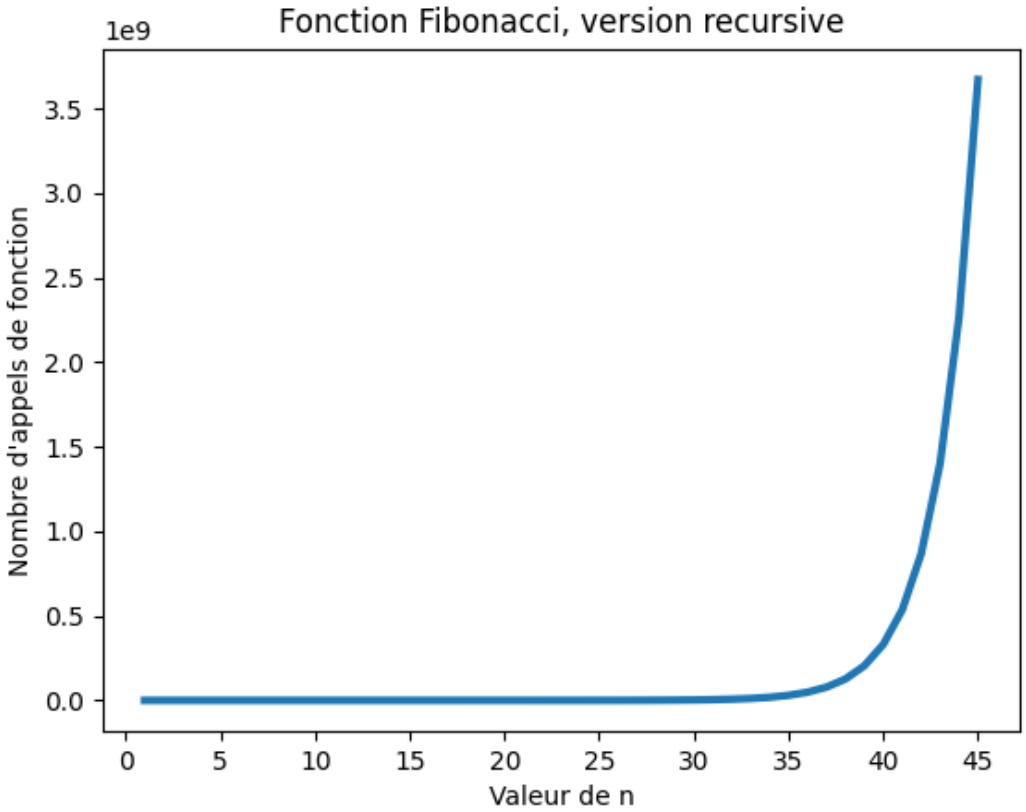
Efficacité d'un algorithme

La version récursive s'exécute de plus en plus lentement en fonction de la valeur du paramètre n.

```
1          int fibo_rec(int n)
2          {
3              if(n<2) return n;
4              return fibo_rec(n-1) + fibo_rec(n-2);
5          }
```

Pour observer cette évolution, nous allons utiliser une variable globale qui va augmenter de 1 à chaque fois qu'on rentre dans la fonction fibo_rec.

Efficacité d'un algorithme



On assiste à une explosion du nombre d'appels de fonctions, du fait de la récursivité.

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Temps de développement ?
2. Temps d'exécution ?
3. Quantité de mémoire utilisée ?
4. Facilité de lecture ?

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Le temps de développement correspond au travail du développeur. Du point de vue de l'utilisateur, on se fiche pas mal du temps passé pour produire un logiciel. Ce qui compte, c'est qu'il fonctionne et qu'il soit agréable à utiliser.
2. Temps d'exécution ?
3. Quantité de mémoire utilisée ?
4. Facilité de lecture ?

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Temps de développement ?
2. Le temps d'exécution correspond au temps nécessaire pour obtenir un résultat. Il dépend de l'algorithme, de la qualité de l'implémentation, et de la machine utilisée.
3. Quantité de mémoire utilisée ?
4. Facilité de lecture ?

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Temps de développement ?
2. Temps d'exécution ?
3. La quantité de mémoire utilisée dépend directement de la qualité de l'implémentation et de l'algorithme utilisé. La mémoire est une ressource limitée.
4. Facilité de lecture ?

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Temps de développement ?
2. Temps d'exécution ?
3. Quantité de mémoire utilisée ?
4. La Facilité de lecture d'un code n'est pas à négliger. Plus un code est simple à lire, plus vous augmentez votre productivité et celle de votre équipe. Toutefois, cela ne peut influer que le temps de développement, de maintenance et la qualité d'utilisation.

Efficacité d'un algorithme

Quels critères pour évaluer l'efficacité ?

1. Temps de développement ?
2. Temps d'exécution !
3. Quantité de mémoire utilisée ?
4. Facilité de lecture ?

Exercices.

Nous allons voir ensemble deux exercices. Le but est de vous montrer l'impact d'une réflexion sur le temps d'exécution d'un programme.

Exercice 1

Les facteurs premiers de 13195 sont $5, 7, 13$ et 29.

Quel est le plus grand facteur premier du nombre 600851475143 ?

Exercices.

Exercice 2

La suite des nombres triangulés est générée en ajoutant les entiers naturels un par un. Ainsi, le septième nombre triangulé sera $1+2+3+4+5+6+7=28$.
Les premiers termes de cette suite sont : $1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$
Pour chacun de ces nombres, la liste de leurs diviseurs est la suivante :
1 : 1
3 : 1, 3
6 : 1, 2, 3, 6
10 : 1, 2, 5, 10
15 : 1, 3, 5, 15
21 : 1, 3, 7, 21
28 : 1, 2, 4, 7, 14, 28

Quelle est la valeur du premier nombre triangulé ayant plus de 500 diviseurs ?

Exercices.

Première approche :

1. Énumérer les nombres triangulés. Pas besoin de considérer les autres nombres.
2. Tant que le nombre a moins de 500 diviseurs :
   1. Énumérer la liste des diviseurs.
   2. Si le nombre possède plus de diviseurs que ce qu'on a rencontré jusqu'à présent, afficher le nombre et le nombre de ses diviseurs.

Exercices.

```
1   int nb_diviseurs(int n)
2   {
3       int i,nb=0;
4       for(i=1;i<=n;i++)
5       {
6           if(n%i==0) nb=nb+1;
7       }
8
9       return nb;
10  }
11
12  void brute_force()
13  {
14      int i=2,nb=1,nb_div=0,nb_div_max=0;
15      while(nb_div<=500)
16      {
17          nb_div=nb_diviseurs(nb);
18          if(nb_div>nb_div_max)
19          {
20              nb_div_max = nb_div;
21              printf("%d:%d\n",nb,nb_div);
22          }
23
24          nb=nb+i;
25          i++;
26      }
27  }
```

Exercices.

Temps d'exécution sur ma machine : trop long, j'ai arrêté au bout de 15 minutes. Bref, une horreur !

Comment améliorer le temps d'exécution ? Il faut déterminer l'endroit où se situent les opérations les plus nombreuses. A votre avis ?

Exercices.

Pour compter le nombre de diviseurs d'un nombre n, on considère tous les nombres compris entre 1 et n (inclus) :

- Pour 1, on teste 1 valeur.
- Pour 1000, on teste 1000 valeurs.
- ...

Or, si un entier a divise n, cela signifie qu'il existe un autre entier b tel que $a \times b = n$. On peut donc compter directement deux diviseurs. On va chercher les diviseurs en partant de 1 et en allant jusqu'à $\sqrt{n}$ car il n'existe pas de diviseur plus grand.

```
1        int nb_diviseurs(int n)
2        {
3            int i,nb=0;
4            for(i=1; i*i<=n; i++)
5            {
6                if(n%i==0) nb=nb+2;
7            }
8            return nb;
9        }
```

Exercices.

```
1:2
6:4
28:6
36:10
120:16
300:18
528:20
630:24
2016:36
3240:40
5460:48
25200:90
73920:112
157080:128
437580:144
749700:162
1385280:168
1493856:192
2031120:240
2162160:320
17907120:480
76576500:576
```

1. Ancienne fonction :
   - Temps d'exécution : Trop long !
   - 316 005 702 376 divisions.
2. Nouvelle fonction :
   - Temps d'exécution : 2.535 secondes.
   - 54 145 952 divisions.

On observe donc que la nouvelle fonction réalise 5836 fois moins de divisions que la première version. Cela laisse supposer un temps de calcul de plus de 4h pour la première.

Efficacité d'un algorithme

Comparer le temps d'exécution de différentes fonctions, n'est pas une bonne idée.

Les raisons

- **Du scientifique :** montrer qu'un algorithme est plus rapide qu'un autre sur des exemples n'est pas une preuve.
- **De l'ingénieur :** de nombreux programmes tournent sur la machine et peuvent perturber la mesure. Les mesures sont trop dépendantes de l'ordinateur.
- **Du miagiste :** il faudrait programmer les fonctions pour avoir la réponse.

Alors ? Comment fait-on ?

Sommaire

Instructions élémentaires

Instructions élémentaires

- les opérations arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière, ...)
- la comparaisons de données (relation d'égalité, d'infériorité, ...)
- le transferts de données (lecture et écriture dans un emplacement mémoire)
- les instructions de contrôle (branchement conditionnel et inconditionnel, appel à une fonction auxiliaire, ...)

Instructions élémentaires

Instructions élémentaires

- Il est impératif que les données représentant les nombres soient codées sur un nombre fixe de bits (comme en C).
- L'exponentiation est-elle une instruction élémentaire ?
- La comparaison de deux chaines de caractères est-elle une instruction élémentaire ?

Taille de l'entrée

Très souvent, le nombre d'opérations élémentaires réalisées par un algorithme va dépendre de la taille des entrées.

- Nombre d'éléments du tableau
- Valeur d'un entier

Algorithme de tri, primalité d'un nombre, énumération des sous-ensembles ...

Mesure du temps d'exécution

Réalisation d'un tableau

valeur de n	$log_2(n)$	n	n^2	n^3	1.1^n	2^n	$n!$	n^n
10								
50								
100								
500								
1000								
10^4								
10^5								
10^6								

A vous de réaliser un programme, permettant de mesurer les temps pour les différents temps et tailles d'entrées.

Mesure du temps d'exécution

La fonction clock

Cette fonction retourne une approximation de la durée écoulée d'utilisation du processeur par le programme. Cette durée est exprimée en unités d'horloge (de type clock_t).

La constante CLOCKS_PER_SEC indique le nombre d'unités d'horloge par seconde.

```
1  #include <stdio.h>
2  #include <time.h>
3
4  int main()
5  {
6      int i;
7      clock_t exemple;
8      for(i=0;i<200;i++)
9          printf("Exemple stupide et chronophage.\n");
10     exemple = clock();
11     printf("Nombre cycles par seconde : %ld\n",CLOCKS_PER_SEC);
12     printf("Nombre de cycles du programme : %ld\n",exemple);
13     printf("Temps du programme : %f",(float)exemple/(float)CLOCKS_PER_SEC);
14     return 0;
15 }
```

Mesure du temps d'exécution

```c
#include <stdio.h>
#include <time.h>

int main()
{
    int i;
    clock_t exemple;
    for(i=0;i<200;i++)
    printf("Exemple stupide et chronophage.\n");
    exemple = clock();
    printf("Nombre cycles par seconde : %ld\n",CLOCKS_PER_SEC);
    printf("Nombre de cycles du programme : %ld\n",exemple);
    printf("Temps du programme : %f",(float)exemple/(float)
        CLOCKS_PER_SEC);
    return 0;
}
```

```
Exemple stupide et chronophage.
Exemple stupide et chronophage.
Nombre cycles par seconde : 1000
Nombre de cycles du programme : 2552
Temps du programme : 2.552000
Process returned 0 (0x0)   execution time : 2.638 s
```

Mesure du temps d'exécution

Exemple pour n^2, avec $n = 10^4$

```c
#include <stdio.h>
#include <time.h>
float test(long long n)
{
    long long i, nb = 1, ite = n*n;
    clock_t debut,fin;
    debut = clock();
    for(i=0;i<ite;i++)
    {
        nb = nb* i;
    }
    fin = clock();
    return (float)(fin-debut)/(float)CLOCKS_PER_SEC;
}

int main()
{
    printf("n*n avec n=10^4 : %f",test((long long)pow(10,4)));
    return 0;
}
```

Mesure du temps d'exécution

Réalisation d'un tableau

Voici, pour une partie du tableau, les temps nécessaires à l'exécution d'un algorithme en fonction de la taille de l'entrée et du nombre d'instructions nécessaires pour l'algorithme. A titre indicatif, nous avons considéré que la machine effectue 10^9 opérations à la seconde.

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ années
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} années
10^4	13 ns	10 µs	133 µs	100 ms	17 s	
10^5	17 ns	100 µs	2 ms	10 s	11,6 jours	
10^6	20 ns	1 ms	20 ms	17 mn	32 années	

Figure: Temps nécessaire à l'exécution d'un algorithme en fonction du nombre de ses instructions.

Notez les différences d'échelle considérables qui existent entre les ordres de grandeurs.