Algorithmique
Recherches dans un tableau

Chap. 16

1 Algorithmique

1.1 Introduction

Définition 1. *Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions à réaliser afin de résoudre un problème.*

Attention. En informatique, pour qu'un algorithme puisse être implémenté, il est nécessaire de s'assurer que la « suite finie et non ambiguë d'opérations ou d'instructions à réaliser » s'effectue en une *durée finie*.

Lorsqu'on élabore ou étudie un algorithme, il est donc nécessaire de vérifier :

Sa finitude. Il doit se terminer en un temps fini.[1]

Sa correction. Il doit (généralement) donner le bon résultat[2].

Sa performance. Plusieurs algorithmes peuvent permettre de résoudre une même classe de problèmes. Ils ne nécessiteront cependant pas tous l'utilisation de la même quantité de mémoire ou le même nombre d'étapes de calcul, donc la même durée.

Remarque. La performance d'un algorithme porte sur deux aspects : la **durée du calcul** et la **quantité de mémoire** nécessaires à la résolution du problème. *Malheureusement ces deux points s'opposent*. Il est souvent nécessaire d'occuper davantage la mémoire pour gagner en temps de calcul, ou d'écrire plus d'instructions, et donc faire plus de calculs, pour aboutir à une gestion de la mémoire optimale.
Conformément au programme, on limite la *performance algorithmique* à la **complexité algorithmique**.

1.2 Notion de complexité algorithmique

Définition. *La **complexité algorithmique**[3] donne des informations sur la **durée du calcul** nécessaire à la résolution du problème. Plus la complexité algorithmique est petite, moins de calculs sont effectués et plus l'algorithme est performant (sous réserve que la gestion de l'espace mémoire utilisé par l'implémentation de l'algorithme ne constitue pas un problème).*

1. Ce n'est cependant pas toujours une mauvaise nouvelle que certains algorithmes aient besoin d'un « temps infini » pour résoudre un problème : certains choix en cryptographie reposent sur l'idée que « casser » la protection nécessite une durée de calcul trop grande, pour le matériel dont on dispose aujourd'hui.

2. Pour certains problèmes on peut devoir se contenter d'une solution approchée.

3. La complexité des algorithmes sera systématiquement déterminée en classe de terminale. Dans ce chapitre, nous n'étudierons que la complexité de la recherche séquentielle.

On exprime la complexité d'un algorithme en fonction de la ***taille (nombre) des données qu'il manipule*** *en considérant que chaque instruction élémentaire s'effectue en temps constant.*

Il existe plusieurs méthodes pour analyser la complexité d'un algorithme, comme :

L'analyse moyenne. Il s'agit d'évaluer comment varie la durée moyenne des calculs à effectuer en fonction du nombre de données manipulées.

L'analyse optimiste. Il s'agit d'évaluer comment varie le nombre de calculs à effectuer, dans *le scénario le plus favorable*, en fonction du nombre de données manipulées.
Par exemple, on cherche une valeur dans une liste et c'est le première qui apparaît lors de cette recherche.

L'analyse pessimiste (ou du pire cas). Il s'agit d'évaluer comment varie le nombre de calculs à effectuer, dans *le scénario le moins favorable*, en fonction du nombre de données manipulées.
Par exemple, on cherche une valeur dans une liste alors qu'elle n'est pas présente dans la liste.

Lorsqu'on utilise le résultat de l'analyse du « pire cas », aucune mauvaise surprise ne peut intervenir, le pire cas à été envisagé à l'avance.

Les complexités des différents algorithmes varient beaucoup. On peut néanmoins regrouper les algorithmes en quelques grandes familles :

Les algorithmes logarithmiques. Leur notation est de la forme $O(\log N)$. Ces algorithmes sont très performants en temps de traitement. Le nombre de calculs dépend du logarithme du nombre de données à traiter.

Sur la Figure 1., on peut constater que, *plus le nombre de données N à traiter est important, moins le nombre de calcul à effeffectuer augmente rapidement — la valeur de la dérivée est de moins en moins grande*.

La fonction logarithme à utiliser dépend du problème étudié mais, comme la complexité est définie à un facteur près, la base du logarithme n'a pas d'importance.

La complexité logarithmique apparaît dans les problèmes dans lesquels l'ensemble des données peut être décomposé en deux parties égales, qui sont elles-mêmes décomposées en 2[4]. Le logarithme à utiliser est alors la fonction réciproque de $f : x \longmapsto 2x$, c'est à dire $\log_2$ (aussi appelé logarithme entier[5]).

Les algorithmes linéaires. Leur notation est de la forme $O(N)$. *Ces algorithmes sont rapides. Le nombre de calculs dépend, de manière linéaire, du nombre de données initiales à traiter*.

La complexité linéaire apparaît dans les problèmes dans lesquels *on parcourt séquentiellement l'ensemble des données* pour réaliser une opération (recherche d'une valeur, par exemple).

Les algorithmes linéaires et logarithmiques. Leur notation est de la forme $O(N \log N)$.

Cette complexité apparaît dans des problèmes dans lesquels on *découpe répétitivement les données en deux parties, que l'on parcourt séquentiellement* ensuite.

Les algorithmes de type puissance. Leur notation est de la forme $O(N^k)$ où k est la puissance.

4. Cf. dichotomie dans ce document.

5. Le **logarithme entier** d'un nombre x supérieur ou égal à 1 est le *nombre de fois qu'il faut le diviser par deux pour obtenir un nombre inférieur ou égal à 1*.

Une complexité quadratique apparait, par exemple, lorsqu'on parcourt un *tableau à deux dimensions*.

Les algorithmes exponentiels ou factoriels. Leur notation est de la forme $O(e^N)$ ou $O(N!)$. Ce sont les algorithmes les plus complexes.

Le nombre de calculs augmente de façon exponentielle ou factorielle en fonction du nombre de données à traiter. Un algorithme de complexité exponentielle traitera, dans le pire des cas, un ensemble de 10 données en effectuant 22 026 calculs ; un ensemble de 100 données en effectuant $2{,}688 \cdot 10^{43}$ calculs !!!

On dit généralement que les problèmes produisant ce type d'algorithmes sont « non calculables ».

On rencontre ce type d'algorithmes dans les problématiques liées à la programmation de fonctions humaines comme la vision, la reconnaissance des formes ou l'intelligence artificielle.

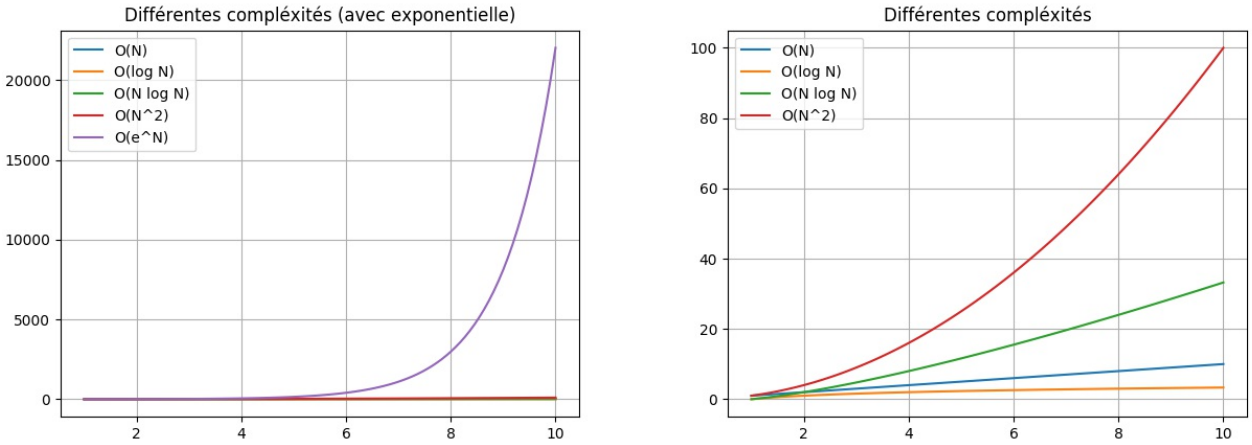

Figure 1. Évolution des algorithmes en fonction du nombre de données

Complexité	**Durée pour $N = 10^6$**
Logarithmique : $O(\log N)$	10 ns
Linéaire : $O(N)$	1ms
Quadratique : $O(N^2)$	1/4 heure
Polynomiale : $O(N^k)$	30 ans si $k = 3$
Exponentielle : $O(2^N)$	plus de 10^{300000} milliards d'années

Tableau 1. Ordres de grandeur des durées d'exécution d'un problème de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde (« Informatique pour tous en CPGE », éditions Eyrolles).

1.3 Terminaison d'un algorithme, variant de boucle

La structure qui généralement doit retenir l'attention lors de l'analyse de la terminaison d'un algorithme est la structure de boucle.

Définition 2. *On appelle* ***variant*** *d'une boucle une fonction qui a pour variable les variables du problème, qui retourne une valeur positive et qui décroît à chacune des itérations de la boucle, jusqu'à s'annuler ou prendre une valeur constante négative qui dépend de la condition d'arrêt de la boucle.*

La découverte d'un variant de boucle permet de conclure que la boucle se termine puisqu'un entier positif ne peut décroître infiniment.

Exemple 1. (Calcul de la plus petite puissance de deux supérieure ou égale à un entier n)

Algorithme 1

Entrée : *variable entière n*
Sortie : variable entière p dont la valeur est égale à la plus petite puissance de deux supérieure ou égale à n

Début
  $p \longleftarrow 1$
  TantQue $p < n$ **faire**
    $p \longleftarrow 2\,p$
  FinTantQue
Fin

La fonction f d'expression $f(p) = n - p$ est un variant de boucle. En effet, tant que $p < n$, $f(p) > 0$ et f décroit sur l'ensemble des valeurs de p. Condition d'arrêt : $p \geqslant n$, donc $f(p) \leqslant 0$.

La boucle se termine donc.

Exercice 1. (Palindrome)

Algorithme 2

Fonction : *palindrome(m)*
Action : *Teste si une chaîne de caractères m est un palindrome*
Début
  $i \longleftarrow 0$
  $j \longleftarrow \text{longueur}(m) - 1$
  TantQue $i \leqslant j$ **faire**
    Si $m[i] = m[j]$ **alors**
      $i \longleftarrow i + 1$
      $j \longleftarrow j - 1$
    Sinon
      Renvoyer Faux
    FinSi
  FintantQue
  Renvoyer Vrai
Fin

1. Décrire au moyen d'un tableau indiquant l'évolution des valeurs des variables le fonctionnement de l'algorithme 2 pour la chaîne de caractères « sauras » puis pour la chaîne de caractères « radar ».

   Remarque. Le premier élément d'une chaîne a pour indice 0.

2. Montrer que $\delta = j - i$ est un variant de la boucle **TantQue**.

3. En déduire que la boucle **TantQue** se termine.

Solution.

1. Pour le mot « sauras » :

i	j
0	5
1	4
2	3

Retourne « Faux ».

Pour le mot « radar » :

i	j
0	4
1	3
2	2
3	1

Retourne « Vrai ».

2. Pour i et j donnés, $\delta_i = j - i$. Au tour suivant, $\delta_{i+1} = j - 1 - (i+1) = j - i - 2 < \delta_i$. La fonction δ est décroissante. De plus, $j \geqslant i$, donc $\delta \geqslant 0$.

3. Condition d'arrêt : $i > j$, donc $\delta < 0$. La boucle se termine dès que la fonction retourne des valeurs négatives.

1.4 Correction d'un algorithme : invariant de boucle

Définition 3. *On appelle **invariant d'une boucle** une propriété qui si elle est vraie avant l'exécution d'une itération le demeure après l'exécution de l'itération.*

Un invariant de boucle doit être vrai avant de commencer la boucle et est alors garanti de rester correct après chaque itération de la boucle. En particulier, l'invariant sera toujours vrai à la fin de la boucle. Les boucles et la récursivité étant fondamentalement similaire, il y a peu de différence entre démontrer un invariant de boucles et prouver qu'un programme est correct en utilisant un raisonnement par récurrence.

Exemple 2. (Calcul de 2^n)

Algorithme 3

Entrée : variable entière n
Sortie : variable entière p dont la valeur est égale à 2^n
Début
  $p \longleftarrow 1$
  Pour k **allant de** 1 à n **faire**
    $p \longleftarrow 2\,p$
  FinPour
Fin

La propriété « à chaque tour de boucle $p = 2^k$ » est un invariant de boucle.

Initialisation. Au premier tour de la boucle : $p = 2 \times 1 = 2^1$.

Conservation. On suppose l'invariant vérifié au tour i de la boucle : $p = 2^i$. Au tour $i+1$, $p_{i+1} = p_i\, 2 = 2^i \times 2 = 2^{i+1}$.

Terminaison. La boucle réalise n tours ; au dernier tour $p = 2^{n-1} \times 2 = 2^n$.

L'algorithme est correct.

Exercice 2. On considère la fonction suivante qui permet de calculer le quotient et le reste de la division euclidienne d'un entier positif par un entier strictement positif :

Algorithme 4

Fonction : *division(a,b)*
Action : *Calcul du quotient q et du reste r de la division euclidienne de l'entier a par l'entier b*
Début
  $q \longleftarrow 0$
  $r \longleftarrow a$
  TantQue $r \geqslant b$ **faire**
    $q \longleftarrow q + 1$
    $r \longleftarrow r - b$
  FinTantQue
  Renvoyer q, r
Fin

1. Décrire le fonctionnement de l'algorithme pour l'entrée (a=17, b=5) au moyen d'un tableau indiquant l'évolution des valeurs des variables au fil des itérations.

2. Montrer que la boucle **TantQue** se termine en utilisant un variant de boucle.

3. Montrer que la propriété $a = \text{bq} + r$ est un invariant de la boucle **TantQue**, en déduire que l'algorithme produit le résultat atendu.

2 Recherche d'un élément dans un tableau

La *recherche* est une opérations fondamentale dans un programme qui gère un ensemble de donnés. La recherche traite deux informations distinctes : la **clé** et la **donnée**. *La clé est l'information utilisée dans la recherche pour localiser la donnée.*

2.1 Recherche séquentielle

La *recherche séquentielle* est une méthode élémentaire qui consiste à comparer la clé recherchée à toutes les autres clés. Rechercher une valeur entière dans une liste d'entiers se résume à parcourir chaque élément de la liste et à vérifier s'il s'agit de l'élément recherché. Dans cet exemple, la clé et la donnée recherchée sont confondues.

Exercice 3. (Recherche séquentielle)

Algorithme 5

Fonction : *recherche(tab, nb, valeur)*
Action : *recherche la valeur « valeur » dans le tableau « tab » de longueur « nb »*
Début
  trouvé $\longleftarrow$ Faux
  $i \longleftarrow 0$
  $i_{\text{val}} \longleftarrow -1$
  TantQue $i < \text{nb}$ **faire**

```
        Si tab[i] = valeur alors
            i_val ⟵ i
        FinSi
        i ⟵ i + 1
    FinTantQue
    Renvoyer i_val
Fin
```

1. Décrire le fonctionnement de l'algorithme pour les valeurs tab $= [1, 8, 3, 5, 9]$, nb $= 5$ et val $= 3$.

2. Démontrer que l'algorithme se termine.

3. Démontrer que l'algorithme est correct.

4. Démontrer que l'algorithme est en $O(N)$.

5. Comment pourrait-on améliorer l'efficacité de cet algorithme ?

Exercice 4.

Algorithme 6

```
Fonction : recherche(tab, nb, valeur)
Action : recherche la valeur « valeur » dans le tableau « tab » de longueur « nb »
Début
    trouvé ⟵ Faux
    i ⟵ 0
    TantQue i < nb et (Non trouvé) faire
        Si tab[i] = valeur alors
            trouvé ⟵ Vrai
        Sinon
            i ⟵ i + 1
        FinSi
    FinTantQue
    Si trouvé = Vrai alors
        Renvoyer i
    Sinon
        Renvoyer -1
Fin
```

1. Expliquer en quoi la version ci-dessus de l'algorithm est plus efficace.

Exercice 5.

Il est encore possible d'améliorer l'algorithm 6 en évitant de faire deux tests à chaque tour de boucle. L'idée consiste à placer la valeur à la dernière position du tableau (après avoir vérifié que cette dernière valeur n'est pas la valeur rechercée) ; c'est ce que l'on appelle une *sentinelle*. Lors de la recherche, on est donc maintenant certain de trouver la valeur (au moins à la fin). À la fin, il suffit de remettre en place la valeur intialement dans le tableau et de conclure.

1. Écrire l'algorithme décrit ci-dessus.

2.2 Recherche dichotomique

Dès que le nombre de données devient important, la *recherche séquentielle* n'est plus envisageable. Sa complexité en $O(N)$ pénalise les autres traitements qui l'utilisent. Il faut diminuer le temps de traitement par l'emploi d'algorithmes plus performants, comme la *recherche dichotomique*. Cet algorithme est basé sur le principe de « diviser pour régner ». On divise l'ensemble de recherche en deux sous-ensembles égaux. On détermine ensuite dans quel sous-ensemble doit se trouver la clé de recherche, puis on poursuit la recherche dans ce sous-ensemble.

Attention. Le préalable à cette méthode de recherche est de disposer d'*un ensemble trié de données*, car la détermination du sous-ensemble dans lequel se poursuit la recherche se fait par comparaison entre la valeur recherchée et les valeurs de début et de fin du sous-ensemble.

Remarque. La division par deux de l'ensemble de données de recherche à chaque appel indique que sa complexité est en $O(\log N)$.

Recherche dans un annuaire comprenant $66 \cdot 10^6$ entrées :

- $66 \cdot 10^6$ tests dans le pire des cas pour la recherche séquentielle ;
- $\log_2(66 \cdot 10^6) = 26$ tests dans le pire des cas pour la recherche dichotomique !!!

Exercice 6.

Algorithme 7

```
Fonction : recherche(tab, début, fin, valeur)
Action : recherche la valeur « val » dans le tableau « tab » entre les indices « début » et « fin »
Début
  résultat⟵ −1
  trouvé ⟵ FAUX
  TantQue (début ⩽ fin) ET (NON trouvé) faire
      milieu⟵ (début + fin)//2
      Si (valeur = tab[milieu]) alors
          résultat ⟵ milieu
          trouvé ⟵ VRAI
      Sinon Si (valeur < tab[milieu]) alors
          fin⟵milieu −1
       Sinon
          début⟵milieu +1
       FinSi
   FinTantQue
   Renvoyer résultat
Fin
```

1. Décrire le fonctionnement de l'algorithme pour les valeurs tab = [1, 8, 3, 5, 9], nb = 5 et val = 3.
2. Démontrer que l'algorithme se termine.
3. Démontrer que l'algorithme est correct.
4. Démontrer que l'algorithme est en $O(\log_2 N)$.

3 Exercices

Exercice 7.

Écrire un programme qui, dans un premier temps, crée une liste comportant 1 000 nombres entiers (compris entre 1 et 10 000) tirés au hasard et qui, ensuite, implémente les algorithmes de recherche séquentielle pour vérifier si un nombre entier, entré par l'utilisateur, est présent dans la liste. Le retour doit correspondre à la position dans la liste du nombre (retourner -1 si le nombre n'est pas présent).

Exercice 8.

Écrire un programme inspiré du programme précédent, qui retourne toutes les positions du nombre recherché (s'il est présent plusieurs fois).

Exercice 9.

Écrire un programme de devinette : l'utilisateur choisit un nombre au hasard (compris entre 1 et 50) et l'ordinateur doit découvrir ce nombre. Faire en sorte que le nombre de tentatives de l'ordinateur lui permette de trouver la bonne valeur à coup sur (choisir de façon pertinente ce nombre).