République Algérienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Université de 8 Mai 1945 – Guelma -
Faculté des Mathématiques, d'Informatique et des Sciences de la matière
Département d'Informatique

Polycopié de cours

2ème année licence
Option : Informatique

Algorithmique et Complexité

Par : Dr. Chohra Chemseddine

Année universitaire 2023/2024

Table des matières

Avant propos

Ce cours s'adresse aux étudiants de deuxième année de licence en informatique. Il est destiné à leur donner les bases nécessaires pour comprendre la notion de complexité algorithmique, une matière centrale et fondamentale dans votre cursus d'ingénieur en informatique. L'algorithmique, en tant que discipline est l'art de concevoir des algorithmes, c'est-à-dire des suites d'instructions bien définies permettant de résoudre un problème ou d'accomplir une tâche donnée. Elle est à la base de nombreuses applications informatiques que nous utilisons au quotidien. De la recherche d'itinéraires optimisés sur une carte à la recommandation de produits en ligne, en passant par le tri de nos e-mails, les algorithmes sont omniprésents et jouent un rôle majeur dans la performance et l'efficacité des logiciels que nous utilisons. L'un des aspects cruciaux que nous aborderons dans ce cours est la notion de complexité algorithmique. La complexité d'un algorithme mesure sa performance en termes de ressources qu'il consomme, telles que le temps d'exécution et l'espace mémoire. Comprendre et analyser la complexité algorithmique est indispensable pour évaluer la qualité d'un algorithme et sa capacité à traiter efficacement des données de plus en plus volumineuses. Nous étudierons donc les différentes méthodes d'analyse de complexité, les notations utilisées, et comment évaluer la pertinence d'un algorithme en fonction des contraintes spécifiques de chaque situation.

Ce cours est divisé en quatre (4) chapitres, dont voici un bref aperçu :

1. **Complexité algorithmique : rappels d'algorithmique :** Ce chapitre est consacré à rappeler les notions de base en algorithmique, à savoir les algorithmes, les structures de données, les types de données, etc. Ces notions ont déjà été abordées en première année (les mudules d'algorithmique et structures de données 1 et 2), mais nous allons les rappeler ici en insistant sur les aspects qui nous intéressent dans ce cours. Nous allons également introduire dans ce chapitre la notion de complexité algorithmique, en donnant plusieurs exemples d'algorithmes et en analysant leur complexité.
2. **Algorithmes de tri :** Le tri est une opération fréquente dans le traitement de données, nous explorerons dans ce chapitre différentes techniques de tri, de la plus

simple à la plus sophistiquée. Nous apprendrons à évaluer leur complexité respective et à choisir le tri le mieux adapté à chaque cas d'utilisation.

3. **Les arbres :** Les arbres sont des structures de données très utilisées en informatique, notamment dans les bases de données, les systèmes de fichiers, et les algorithmes de recherche. Nous étudierons dans ce chapitre les arbres binaires, les arbres binaires de recherche, les arbres de recherche équilibrés, etc. Nous verrons comment les utiliser pour résoudre des problèmes concrets et comment évaluer leur complexité.
4. **Les graphes :** Enfin, nous aborderons les graphes, qui sont des structures de données non linéaires permettant de modéliser de nombreuses situations du monde réel, telles que les réseaux sociaux, les réseaux informatiques, les circuits électriques, etc. Nous explorerons les algorithmes de parcours de graphes, les algorithmes de recherche de chemins, et d'autres problèmes classiques associés aux graphes. Nous étudierons également les algorithmes de plus courts chemins, qui sont très utilisés dans les systèmes de navigation et les applications de cartographie.

Tous les exemples et exercices de ce cours sont écrits en langage C pour faciliter l'implémentation et les tests, et aussi pour s'inscrire dans la continuité des modules d'algorithmique vus en première année. Cependant, les concepts et les techniques que nous allons étudier sont applicables à n'importe quel autre langage de programmation.

Nous espérons que ce cours sera une expérience enrichissante et stimulante pour vous. En vous fournissant des connaissances approfondies dans ce domaine, nous souhaitons vous doter d'outils essentiels pour résoudre des problèmes complexes et relever les défis de l'informatique moderne. Nous vous encourageons à participer activement aux discussions, à poser des questions et à travailler en groupe pour une meilleure compréhension des concepts présentés. La maîtrise de l'algorithmique est une compétence fondamentale qui vous servira tout au long de votre parcours d'ingénieur en informatique et au-delà.

Prérequis : Ce cours suppose que vous avez déjà suivi les modules d'algorithmique et structures de données 1 et 2 en première année, et que vous avez une bonne maîtrise de la programmation en C.

Chapitre 1

Complexité algorithmique

1.1 Introduction

Dans ce chapitre, nous allons explorer les qualités et caractéristiques des algorithmes. Nous mettons en évidence les critères qui permettent de juger de la qualité d'un algorithme et l'importance d'écrire des algorithmes efficaces et robustes. Ensuite nous abordons la notion cruciale de complexité algorithmique. La complexité d'un algorithme mesure la quantité de ressources qu'il consomme en fonction de la taille de l'entrée. Autrement dit, elle évalue le temps d'exécution et la mémoire requise par l'algorithme pour traiter des données. Nous étudierons les différentes formes de complexité, notamment la complexité temporelle (temps d'exécution) et la complexité spatiale (consommation mémoire). Comprendre la complexité algorithmique est essentiel pour évaluer les performances d'un algorithme, anticiper ses limites et choisir la meilleure approche pour résoudre un problème donné. Nous présentons les règles de calcul de complexité au meilleur, au pire et au cas moyen.

Pour mieux assimiler les concepts abordés, nous illustrerons chaque notion avec des exemples concrets. Vous aurez l'occasion de voir comment les qualités d'un algorithme se traduisent en code, comment calculer sa complexité, et comment interpréter les résultats obtenus. Ces exemples pratiques vous aideront à mieux visualiser les concepts théoriques et à les appliquer dans des situations réelles.

1.2 Qualités et caractéristiques d'un algorithme

Les algorithmes sont un moyen de résoudre des problèmes. Ils sont utilisés dans de nombreux domaines, notamment en mathématiques, en informatique, en économie, etc. Ils sont omniprésents dans notre vie quotidienne, Lorsque vous utilisez un GPS pour trouver le chemin le plus court entre deux points, utilisez un moteur de recherche pour trouver des informations sur Internet, ou utilisez un logiciel de reconnaissance faciale pour déver-

rouiller votre smartphone, en fait vous êtes en train de vous servir d'un algorithme pour résoudre un problème. Plusieurs algorithmes peuvent être utilisés pour résoudre un même problème. Certains sont plus efficaces que d'autres, c'est-à-dire qu'ils consomment moins de ressources (temps et mémoire) pour résoudre le problème. Il est donc important de savoir évaluer la qualité d'un algorithme et de choisir le plus efficace pour résoudre un problème donné. Dans cette section, nous allons voir les caractéristiques qui permettent de juger de la qualité et de l'efficacité d'un algorithme. Les présentés ci-dessous peuvent avoir plus ou moins d'importance selon le contexte d'utilisation de l'algorithme :

1. **La précision :** l'algorithme doit donner des résultats corrects et précis pour tous les cas d'entrée possibles, en résolvant le problème spécifié conformément aux spécifications. Il ne doit pas contenir d'erreurs logiques et doit produire des résultats attendus.
2. **Simplicité et clarté :** un bon algorithme doit être simple et facile à comprendre. Il doit être écrit de manière claire et lisible, avec des étapes faciles à suivre. Une bonne structure de l'algorithme facilite la maintenance et la compréhension du code en cas de besoin.
3. **Robustesse :** l'algorithme doit être capable de traiter des cas imprévus et de gérer des situations exceptionnelles sans échouer brutalement ou fournir des résultats incorrects. Il doit être résistant aux données incorrectes ou inattendues.
4. **Modularité :** un algorithme modulaire est composé de plusieurs modules indépendants qui peuvent être réutilisés dans d'autres algorithmes. Il est plus facile de maintenir et de déboguer un algorithme modulaire. De plus, il est plus facile de le réutiliser dans d'autres programmes. Concevoir un algorithme modulaire permet une grande scalabilité et facilite la maintenance à cause du fait qu'il soit conçu pour permettre des modifications et des extensions et même des adaptations pour résoudre des problèmes similaires.
5. **Efficacité :** c'est la caractéristique la plus importante dans le contexte de ce chapitre. Un algorithme efficace est celui qui consomme moins de ressources (temps et mémoire) pour résoudre un problème. Il doit être rapide et économique. Il doit être capable de traiter de grandes quantités de données dans un temps raisonnable. L'efficacité d'un algorithme est mesurée par sa complexité algorithmique.

D'autres caractéristiques peuvent être ajoutées à cette liste, comme la portabilité, le paralélisme, etc. Mais dans le cadre de ce cours, nous nous intéressons principalement à la complexité algorithmique.

1.3 Définition de la complexité algorithmique

La complexité algorithmique est une mesure de la quantité de ressources qu'un algorithme consomme pour résoudre un problème. Elle évalue le temps d'exécution et la mémoire requise par l'algorithme pour traiter des données. La complexité algorithmique est une notion importante en informatique. Elle permet d'évaluer les performances d'un algorithme, d'anticiper ses limites et de choisir la meilleure approche pour résoudre un problème donné. Elle permet également de comparer plusieurs algorithmes pour un même problème et de choisir le plus efficace. La complexité algorithmique est étudiée dans le cadre de l'analyse des algorithmes. La complexité est mesurée en fonction de la taille de l'entrée. La taille de l'entrée est le nombre de données à traiter par l'algorithme. Elle peut être exprimée en nombre d'éléments, en nombre de bits, en nombre de caractères, etc. Par exemple, la taille de l'entrée d'un algorithme de tri est le nombre d'éléments à trier. Il y a deux types de complexité algorithmique : la complexité temporelle et la complexité spatiale.

1.3.1 Complexité temporelle

La complexité temporelle d'un algorithme est la mesure du temps d'exécution de l'algorithme en fonction de la taille de l'entrée. Elle est exprimée en nombre d'opérations élémentaires effectuées par l'algorithme. Les opérations élémentaires peuvent être des opérations arithmétiques, des opérations logiques, des opérations d'entrée/sortie, etc. Le temps d'exécution d'un algorithme dépend de la machine sur laquelle il est exécuté, de la taille de l'entrée et de la qualité de l'implémentation de l'algorithme. Il est donc difficile de mesurer le temps d'exécution exact d'un algorithme. C'est pourquoi on mesure le temps d'exécution en fonction de la taille de l'entrée. On peut ainsi comparer plusieurs algorithmes pour un même problème et choisir le plus efficace.

1.3.2 Complexité spatiale

La complexité spatiale d'un algorithme est la mesure de la mémoire requise par l'algorithme pour traiter des données en fonction de la taille de l'entrée. Elle est exprimée en nombre d'unités de mémoire (bits, octets, etc.). La complexité spatiale est également un facteur important à prendre en compte lors de la conception d'algorithmes, en particulier lorsqu'on travaille sur des systèmes embarqués ou des systèmes avec des ressources mémorielles limitées. Un algorithme avec une complexité spatiale trop élevée peut entraîner des problèmes de performance, notamment des problèmes de consommation excessive de mémoire et des problèmes de débordement de mémoire (memory overflow). Il est donc essentiel de trouver un équilibre entre l'efficacité en termes de temps d'exécution et l'utilisation de l'espace mémoire lors de la conception d'algorithmes optimisés.

De nos jours, la complexité temporelle est la plus utilisée pour mesurer la complexité algorithmique. C'est pourquoi, dans ce cours, nous nous intéressons principalement à la complexité temporelle. La complexité spatiale est également importante, mais elle est moins utilisée sauf dans le cas de systèmes embarqués ou de systèmes avec des ressources mémorielles limitées. Dans la suite de ce cours, nous allons écrire simplement "complexité" pour désigner la complexité temporelle, si nous ne précisons pas le contraire.

Il est aussi important de noter que la complexité algorithmique peut dépendre soit de la taille de l'entrée, soit de la valeur des données, soit des deux. Par exemple, le calcul de la somme des n premiers entiers est un problème dont la complexité dépend de la valeur de n même si la taille mémoire de n est constante. Ce détail ne sera pas expliqué à chaque exemple dans ce cours, mais il est important de le garder à l'esprit lors de l'analyse de la complexité d'un algorithme.

1.4 Calculs de la complexité

Le calcul de la complexité d'un algorithme implique d'analyser comment le temps d'exécution varie en fonction de la taille de l'entrée. Pour ce faire, il faut d'abord identifier les opérations élémentaires effectuées par l'algorithme. Ensuite, il faut compter le nombre d'opérations élémentaires effectuées par l'algorithme en fonction de la taille de l'entrée (généralement notée n). Enfin, il faut exprimer le nombre d'opérations élémentaires en fonction de n et simplifier l'expression obtenue. Le résultat obtenu est la complexité de l'algorithme. La notation "Big O" est la plus utilisée pour exprimer la complexité d'un algorithme. Elle permet de définir une borne supérieure sur le temps d'exécution de l'algorithme (au pire des cas). Pour un algorithme qui fait $c \cdot n$ opérations élémentaires pour une entrée de taille n, et c est une constante, la complexité est notée $O(n)$. La constante c est généralement omise, car elle n'a pas d'impact sur la complexité. De même pour les termes de degrés inférieures dans le cas d'une expression polynomiale. Par exemple, si un algorithme fait $c \cdot n^2 + d \cdot n + e$ opérations élémentaires pour une entrée de taille n, on écrit que la complexité est de $O(n^2)$. Il existe d'autres notations pour exprimer la complexité, comme la notation Θ, la notation Ω. Ces notations sont moins utilisées, mais elles permettent de définir des bornes inférieures et des cas moyens sur le temps d'exécution de l'algorithme. Dans ce cours, nous allons nous intéresser principalement à la notation "Big O", les autres notations seront abordées brièvement dans la section 1.5.5.

1.4.1 Calcul élémentaire de la complexité

Le calcul élémentaire de la complexité permet d'évaluer la quantité de temps d'exécution ou d'espace mémoire utilisée par chaque opération en fonction de la taille de l'entrée.

Chaque instruction ou étape de l'algorithme et éxaminée et on lui attribue un coût en termes de temps ou d'espace pour cette opération. Une fois que chaque opération fondamentale a été identifiée, on détermine combien de fois chaque opération est effectuée en fonction de la taille de l'entrée. Par exemple, si une boucle s'exécute n fois, alors le coût de l'opération à l'intérieur de cette boucle sera multiplié par "n". Ensuite, en combinant les coûts de toutes les opérations fondamentales, on obtient la complexité totale de l'algorithme en termes de temps d'exécution ou d'espace mémoire en fonction de la taille de l'entrée.

Le calcul élémentaire de complexité permet de réaliser une analyse fine de la performance de l'algorithme et de comprendre quelles parties de l'algorithme contribuent le plus à sa complexité globale. Cela permet également d'identifier les points critiques qui pourraient nécessiter une optimisation pour améliorer l'efficacité de l'algorithme. Il est important de noter que le calcul élémentaire de complexité est une approche théorique qui fournit une estimation de la complexité en fonction de la taille de l'entrée. Dans la pratique, les performances réelles de l'algorithme peuvent être influencées par d'autres facteurs tels que l'architecture matérielle, le langage de programmation, les optimisations du compilateur, etc. Cependant, l'analyse des opérations fondamentales reste un outil essentiel pour comprendre le comportement global de l'algorithme et pour comparer différentes solutions algorithmiques.

1.4.2 Règles du calcul de complexité "en pire des cas"

Lorsque l'on calcule la complexité "au pire cas" d'un algorithme, on s'intéresse à la performance maximale de l'algorithme, c'est-à-dire le temps d'exécution ou l'espace mémoire requis lorsque l'entrée est la plus défavorable. Pour une tâche donnée x, à effectuer sur une entrée de taille n, nous allons noter $T_x(n) = O(f(n))$ pour dire que le temps d'exécution de l'algorithme est borné par une fonction $f(n)$ lorsque l'entrée est de taille n. Voici les règles de calcul de complexité "au pire cas" :

1. **Opérations élémentaires :** une opération élémentaire est une opération qui prend un temps constant pour s'exécuter. Par exemple, une opération arithmétique, une opération logique, une opération d'entrée/sortie, etc. Le temps d'exécution d'une opération élémentaire est indépendant de la taille de l'entrée. Par conséquent, la complexité d'une opération élémentaire est $O(1)$.
2. **Séquence d'instructions :** si une séquence d'instructions S est composée de k instructions $S_1, S_2, ..., S_k$, alors la complexité de la séquence S est la somme des complexités des instructions $S_1, S_2, ..., S_k$. En d'autres termes, nous pouvons écire :

$$T_S(n) = \sum_{i=1}^{k} T_{S_i}(n) \tag{1.1}$$

3. **Choix :** si une instruction S est composée de plusieurs instructions S_1, S_2, ..., S_k qui sont exécutées en fonction d'une condition (if ou $switch$), alors la complexité de l'instruction S dépend de l'instruction S_i ayant la complexité la plus élevée. On écrit aussi :

$$T_S(n) = \max_{1 \leq i \leq k} T_{S_i}(n) \tag{1.2}$$

4. **Boucle :** si une instruction S est une boucle qui s'exécute n fois au maximum, et que le corps de la boucle a une complexité $T_B(n)$, alors la complexité de l'instruction S est donnée par la formule :

$$T_S(n) = n \cdot T_B(n) \tag{1.3}$$

   Ce raisonnement est valable pour les boucles for ainsi que pour les boucles $while$. Dans le cas d'une boucle $while$, il faut faire attention à ce que la condition de sortie de la boucle soit bien atteinte. Sinon, la boucle s'exécutera indéfiniment et l'algorithme ne se terminera jamais. Dans ce cas, la complexité de la boucle est $O(\infty)$.

5. **Boucles imbriquées :** si une instruction S est composée de k boucles imbriquées, et que le corps de la boucle la plus interne a une complexité $T_B(n)$, alors la complexité de l'instruction S est donnée par la formule :

$$T_S(n) = \prod_{i=1}^{k} n_k \cdot T_{B_i}(n) \tag{1.4}$$

   où n_k est le nombre d'itérations de la boucle k.

6. **Appel de fonction :** si une instruction S est un appel de fonction f qui a une complexité $T_f(n)$, alors la complexité de l'instruction S est la même que celle de la fonction f.

7. **Récursivité :** dans le cas des fonctions récursives, il est nécessaire d'exprimer la complexité de la récursion en fonction de la taille de l'entrée et de calculer le nombre total d'appels récursifs dans le pire cas. Pour cela, il faut définir une relation de récurrence qui exprime la complexité en fonction de la taille de l'entrée. Ensuite, il faut résoudre la relation de récurrence pour obtenir une expression explicite de la complexité en fonction de la taille de l'entrée. Enfin, il faut simplifier l'expression obtenue et déterminer la complexité de la récursion.

1.5 Exemples de calculs de complexités

Dans cette section, nous allons illustrer les règles de calcul de complexité avec des exemples concrets. Nous allons calculer la complexité de plusieurs algorithmes simples. Nous allons également voir comment les qualités d'un algorithme se traduisent en code, comment calculer sa complexité, et comment interpréter les résultats obtenus. Ces exemples pratiques vous aideront à mieux visualiser les concepts théoriques et à les appliquer dans des situations réelles.

1.5.1 Complexité linéaire

La complexité linéaire est une complexité de type $O(n)$. Elle est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est proportionnel à la taille de l'entrée. Par exemple, si un algorithme fait $c \cdot n$ opérations élémentaires pour une entrée de taille n, alors la complexité de cet algorithme est $O(n)$. La recherche linéaire est un exemple d'algorithme dont la complexité est linéaire. L'algorithme de recherche linéaire est utilisé pour rechercher un élément dans un tableau. Il parcourt le tableau élément par élément et compare chaque élément avec la valeur recherchée. Si l'élément est trouvé, l'algorithme s'arrête et renvoie la position de l'élément. Sinon, il renvoie une valeur spéciale (-1 par exemple) pour indiquer que l'élément n'a pas été trouvé. Voici un exemple d'implémentation d'une fonction de de recherche linéaire en langage C :

```
1  int linear_search(int *array, int size, int value) {
2    for (int i = 0; i < size; i++) {
3      if (array[i] == value) {
4        return i;
5      }
6    }
7    return -1;
8  }
```

Algorithm 1.1 – Recherche linéaire

L'opération élémentaire qui nous intéresse dans cet algorithme est la comparaison entre deux entiers. Cette opération est effectuée dans l'instruction if à la ligne 3. Elle est exécutée à chaque itération de la boucle for. La boucle for est exécutée n fois pour une entrée de taille n. La boucle peut éventuellement s'arrêter avant d'avoir parcouru tout le tableau si l'élément est trouvé, mais dans le pire des cas, si l'élément n'est pas trouvé, la boucle sera exécutée n fois. Donc la complexité de la fonction $linear\_search$ est $O(n)$, car elle fait au maximum n opérations élémentaires pour une entrée de taille n.

1.5.2 Complexité constante

La complexité constante est une complexité de type $O(1)$. Elle est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est indépendant de la taille de l'entrée. Par exemple, si un algorithme fait c opérations élémentaires pour une entrée de taille n, avec c une constante, alors la complexité de cet algorithme est $O(1)$. La recherche d'un élément dans un tableau trié est un exemple d'algorithme dont la complexité est constante. Nous allons prendre comme exemple le calcul de la somme des n premiers entiers avec n un entier positif. A première vue, on pourrait penser qu'un algorithme qui résout ce problème doit parcourir tous les entiers de 1 à n et les additionner comme il est montré ci-dessous :

```
1  int sum_n(int n) {
2    int sum = 0;
3    for (int i = 1; i <= n; i++) {
4      sum += i;
5    }
6    return sum;
7  }
```

Algorithm 1.2 – Somme des n premiers entiers - Complexité linéaire

Cet algorithme fait n opérations élémentaires pour une entrée de taille n. Donc sa complexité est $O(n)$. Cependant, il existe une solution plus efficace pour résoudre le même problème. En effet, la somme des n premiers entiers peut être calculée en utilisant la formule de la somme des n premiers éléments d'une suite arithmétique ayant pour premier terme $a_1 = 1$ et pour raison $r = 1$. La somme des n premiers entiers est donnée par la formule suivante :

$$S_n = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} \tag{1.5}$$

Cette formule permet de calculer la somme des n premiers entiers en effectuant trois opérations élémentaires. Voici une implémentation de la fonction sum_n qui utilise la formule 1.5 :

```
1  int sum_n(int n) {
2    return (n * (n + 1)) / 2;
3  }
```

Algorithm 1.3 – Somme des n premiers entiers - Complexité constante

Cette fonction fait toujours trois opérations élémentaires quelle que soit la valeur de n. Donc sa complexité est $O(1)$.

Il est important de noter que la complexité constante ne signifie pas nécessairement qu'un algorithme est toujours plus rapide pour toutes les tailles d'entrée. Elle signifie simplement que le temps d'exécution ou la consommation mémoire ne dépend pas de la taille de l'entrée. La complexité constante est généralement souhaitable pour des opérations qui doivent être très rapides et ne pas dépendre de la taille des données. Cependant, dans certains cas, des algorithmes avec une complexité supérieure peuvent être plus efficaces pour des tailles d'entrée importantes. Le choix de l'algorithme dépend du contexte et des besoins spécifiques de chaque problème à résoudre.

1.5.3 Complexité logarithmique

La complexité logarithmique est une complexité de type $O(\log n)$. Elle est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est proportionnel au logarithme de la taille de l'entrée. Par exemple, si un algorithme fait $c \cdot \log n$ opérations élémentaires pour une entrée de taille n, alors la complexité de cet algorithme est $O(\log n)$. Un exemple d'algorithme dont la complexité est logarithmique est le calcul de la puissance d'un nombre. Comme pour l'exemple précédent, on pourrait penser qu'un algorithme qui calcule x^n avec n un entier positif doit effectuer n multiplications comme il est montré ci-dessous :

```
1  int pow_n(int x, int n) {
2    int result = 1;
3    for (int i = 0; i < n; i++) {
4      result *= x;
5    }
6    return result;
7  }
```

Algorithm 1.4 – Puissance d'un nombre - Complexité linéaire

Cependant, il existe une solution plus efficace pour résoudre le même problème. En effet, la puissance d'un nombre x à la puissance n peut être calculée en utilisant la formule de récurrence suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot (x^{(n-1)/2})^2 & \text{si } n \text{ est impair} \\ (x^{n/2})^2 & \text{si } n \text{ est pair} \end{cases} \tag{1.6}$$

Cette formule permet de calculer la puissance d'un nombre x à la puissance n en divisant le problème en sous-problèmes plus petits. La complexité de cet algorithme est logarith-

mique, car à chaque étape, le problème est divisé par deux. Voici une implémentation de la fonction *pow_n* qui utilise la formule 1.6 :

```
1  int pow_n(int x, int n) {
2    if (n == 0) {
3      return 1;
4    } else if (n % 2 == 0) {
5      int y = pow_n(x, n / 2);
6      return y * y;
7    } else {
8      int y = pow_n(x, (n - 1) / 2);
9      return x * y * y;
10   }
11 }
```

Algorithm 1.5 – Puissance d'un nombre - Complexité logarithmique

L'opération élémentaire de la fonction *pow_n* est la multiplication. Cette opération est effectuée à la ligne 6 et à la ligne 9. La fonction *pow_n* est récursive. Elle s'appelle elle-même avec un paramètre plus petit. La complexité est donnée par la relation de récurrence suivante :

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n/2) + 2 & \text{si } n \text{ est impair} \\ T(n/2) + 1 & \text{si } n \text{ est pair} \end{cases} \tag{1.7}$$

On peut facilement démontrer par récurrence que $T(n) = O(\log_n n)$. Mais une autre façon de voir les choses est de remarquer que le paramètre n est divisé par deux à chaque appel récursif, en d'autre termes, le paramètre perd un bit à chaque appel. Donc le nombre d'appels récursifs est proportionnel au nombre de bits de la valeur de n, ce qui implique que la complexité de la fonction *pow_n* est $O(\log_2 n)$, car elle fait au maximum $2 \cdot \log_2 n$ opérations élémentaires dans le cas où tous les appels sont effectués avec des valeurs impaires.

1.5.4 Complexité quadratique

La complexité quadratique est une complexité de type $O(n^2)$. Elle est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est proportionnel au carré de la taille de l'entrée. Par exemple, si un algorithme fait $c \cdot n^2$ opérations élémentaires pour une entrée de taille n, alors la complexité de cet algorithme est $O(n^2)$. Un exemple d'algorithme dont la complexité est quadratique est la recherche d'une paire d'éléments dans un tableau dont la somme est égale à une valeur donnée. Cet algorithme parcourt le

tableau et compare chaque élément avec tous les autres éléments du tableau. Si la somme de deux éléments est égale à la valeur donnée, l'algorithme s'arrête et renvoie 1 (vrai). Sinon, il renvoie une 0 (faux) pour indiquer que la paire n'a pas été trouvée. Voici un exemple d'implémentation d'une fonction de recherche d'une paire d'éléments dans un tableau dont la somme est égale à une valeur donnée en langage C :

```
1  int pair_sum(int *array, int size, int value) {
2    for (int i = 0; i < size; i++) {
3      for (int j = 0; j < size; j++) {
4        if (array[i] + array[j] == value) {
5          return 1;
6        }
7      }
8    }
9    return 0;
10 }
```

Algorithm 1.6 – Recherche d'une paire d'éléments dont la somme est égale à une valeur donnée

L'opération élémentaire qui nous intéresse dans cet algorithme est la comparaison entre deux entiers. Cette opération est effectuée dans l'instruction if à la ligne 4. Elle est exécutée à chaque itération de la boucle for interne. La boucle for interne est exécutée n fois pour une entrée de taille n. La boucle for externe est également exécutée n fois pour une entrée de taille n. Donc la complexité de la fonction $pair\_sum$ est $O(n^2)$, car elle fait au maximum n^2 opérations élémentaires pour une entrée de taille n.

Il est possible d'améliorer la complexité de cet algorithme en commençant par trier le tableau avec un algorithme de tri efficace (*quicksort*, *mergesort*, etc.). Ensuite, on peut utiliser une recherche dans les deux sens pour trouver la paire d'éléments (la technique *twopointers*). Cet algorithme sera présenté dans le chapitre 2 comme l'une des applications des algorithmes de tri.

La complexité quadratique est un cas spécifique de la complexité polynomiale. Un algorithme dont la complexité est polynomiale est un algorithme dont la complexité est de la forme $O(n^k)$ avec k un entier positif. La complexité polynomiale est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est proportionnel à une puissance de la taille de l'entrée. Par exemple, si un algorithme fait $c \cdot n^k$ opérations élémentaires pour une entrée de taille n, alors la complexité de cet algorithme est $O(n^k)$.

1.5.5 Complexité au mieux, au pire et en moyenne

La complexité au mieux, au pire et en moyenne sont des notations utilisées pour exprimer la complexité d'un algorithme dans des cas particuliers. La complexité au mieux est la complexité de l'algorithme dans le meilleur des cas. C'est-à-dire lorsque l'entrée est la plus favorable. Elle est notée Ω. La complexité au pire est la complexité de l'algorithme dans le pire des cas. C'est-à-dire lorsque l'entrée est la plus défavorable. Elle est notée O. La complexité en moyenne est la complexité de l'algorithme dans le cas moyen. C'est-à-dire lorsque l'entrée est aléatoire. En d'autres termes, c'est la complexité observée sur un grand nombre d'entrées aléatoires. La complexité au mieux est toujours inférieure ou égale à la complexité en moyenne, qui est elle-même inférieure ou égale à la complexité au pire. Prenons comme exemple l'algorithme de recherche linéaire implémenté dans l'algorithme 1.1.

— Dans le meilleir cas, l'élément recherché est à la première position du tableau. Dans ce cas, l'algorithme s'arrête après une seule itération et renvoie la position de l'élément. Donc la complexité au mieux de cet algorithme est $\Omega(1)$.

— La complexité au pire de cet algorithme est $O(n)$ comme nous l'avons déjà expliqué dans la section 1.5.1.

— La complexité en moyenne est un peu plus compliquée à calculer. Elle dépend de la distribution des données dans le tableau ainsi que de la probabilité de trouver l'élément recherché. Si on suppose que les données sont distribuées uniformément dans le tableau, et que l'élément recherché appartient certainement au tableau, alors nous avons $T(n) = \frac{n}{2}$ (la moyenne de 1 à n). Si nous supposons que l'élément recherché appartient au tableau avec une probabilité p, alors la complexité en moyenne est donnée par la formule

$$T(n) = p \cdot \frac{n}{2} + (1 - p) \cdot n \tag{1.8}$$

Dans le cas où $p = 1$, nous avons $T(n) = \frac{n}{2}$, et dans le cas où $p = 0$, nous avons $T(n) = n$. Donc la complexité en moyenne est comprise entre $\Omega(1)$ et $O(n)$.

1.5.6 Complexité exponentielle

La complexité exponentielle est une complexité de type $O(c^n)$. Elle est utilisée pour exprimer la complexité d'un algorithme dont le temps d'exécution est proportionnel à une exponentielle de la taille de l'entrée. Par exemple, si un algorithme fait c^n opérations élémentaires pour une entrée de taille n, alors la complexité de cet algorithme est $O(c^n)$. Un exemple d'algorithme dont la complexité est exponentielle est le la recherche exhaustive d'une solution à un problème (clé de chiffrement, mot de passe, etc.). Cet algorithme

génère toutes les solutions possibles et vérifie si la solution est correcte. Si la solution est correcte, l'algorithme s'arrête et renvoie la solution. Sinon, il continue à générer des solutions. Dans l'exemple de recherche d'un mot de passe, si nous supposons que le mot de passe est composé de n caractères, et que chaque caractère peut prendre c valeurs différentes, alors le nombre total de solutions possibles est c^n. Donc la complexité de cet algorithme est $O(c^n)$.

Les algorithmes dont la complexité est exponentielle (ou factorielle) sont très lents et pas efficaces. Ils sont très rarement utilisés dans des applications réelles pour résoudre des problèmes de taille petite ou moyenne. Cependant, ils peuvent être utilisés pour résoudre des problèmes de taille importante si aucune autre solution plus efficace n'est connue pour résoudre le problème. Les problème qui n'ont pas de solution efficace connue sont appelés des problèmes NP-complets, mais leur étude dépasse le cadre de ce cours.

1.6 Conclusion

Dans ce chapitre, nous avons présenté les concepts de base de la complexité algorithmique. Nous avons vu comment mesurer la complexité d'un algorithme en fonction de la taille de l'entrée. Nous avons également vu comment calculer la complexité d'un algorithme en utilisant les règles de calcul de complexité. Nous avons illustré ces règles avec des exemples concrets. Dans le chapitre suivant, nous allons nous concenter sur les algorithmes de tri pour démonter que pour un problème donné, il existe plusieurs algorithmes possibles, et que certains algorithmes sont plus efficaces que d'autres pour résoudre le même problème.