
# Algorithmique

**Mathy Christiane**

**Bachelier en Informatique, orientation développement d'applications**

**Bloc 1 – UE9 – Programmation intermédiaire**

Avril 2005 – Révision de février 2024

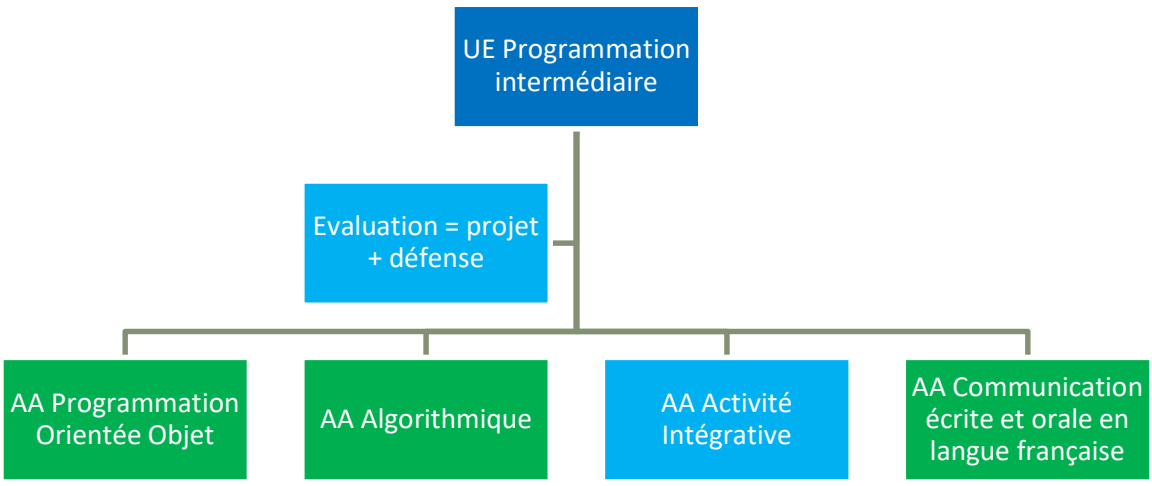
# 1 AVANT-PROPOS À L'ÉDITION 2024

L'alignement « justifié » de ce document a volontairement été abandonné, pour être adapté aux personnes dyslexiques. Je remercie Christiane Mathy de nous permettre à tous d'utiliser les notes de cours qu'elle a rédigé. Dans cette nouvelle édition, j'ai essentiellement effectué une relecture de mise en page. Si vous avez des suggestions quant au contenu ou la présentation de ce document, vous pouvez contactez s.lienardy@helmo.be.

Simon Liénardy, février 2024

# 2 L'UE « PROGRAMMATION INTERMÉDIAIRE »

Ce cours (et les laboratoires associés) fait partie de l'Unité d'Enseignement « Programmation intermédiaire ». Les projets de programmation deviennent de plus en plus complexes, que ce soit en lignes de code ou en ressources matérielles mobilisées. L'activité d'apprentissage de Programmation Orientée Objet explique comment le style orienté objet aide à garder un code structuré, tandis que l'activité d'apprentissage d'Algorithmique étudie les algorithmes afin de minimiser les ressources matérielles requises par un programme. L'activité d'apprentissage nommée « Activité Intégrative » demande à l'étudiant de combiner les notions étudiées en algorithmique et programmation orientée objet, pour réaliser un projet de complexité moyenne. Enfin, l'activité d'apprentissage de Communication prépare l'étudiant à la défense de ce projet devant un enseignant.



## 2.1 THÈMES DU COURS D'ALGORITHMIQUE

Via cette matière, l'étudiant est invité à une **prise de recul par rapport à ses pratiques de programmation** :

- Analyse critique de la **qualité d'un programme**, appliquée tant à des algorithmes célèbres qu'aux programmes écrits par l'étudiant : nous procèderons à des comparaisons, tenterons d'apporter des améliorations, et étudierons en détails certains algorithmes. Parmi les méthodes utilisées, expliquées ou citées, mentionnons les tests systématiques, le calcul de complexité, la validation des boucles, la vérification de prédicats.

Nous aborderons également certaines parties spécifiques en lien avec le cours de programmation avancée: révision de la **récursivité** et étude approfondie des **structures de données complexes** (piles, files, listes chainées, arbres, tables hachées…)

Nous verrons sur l'un ou l'autre exemple comment ces structures de données complexes peuvent être exploitées pour mettre en œuvre certains algorithmes issus des **principales familles** existantes (optimisation, machine learning, arbres de décision, min-max, algorithmes génétiques, pour ne citer qu'eux).

Enfin, nous nous intéresserons à l'**impact environnemental du numérique** dans sa globalité, afin de dégager des pistes pour le réduire.

## 2.2 MODALITÉS D'ÉVALUATION

Elles sont décrites dans la fiche descriptive de l'UE, disponible depuis cette adresse : https://www.helmo.be/fr/formations/i180-informatique-orientation-developpement-dapplications/programme/30224

Elles sont complétées par les informations présentes sur l'espace de cours (plateforme e-learning).

Le cours théorique se déroule en auditoire. Il est assorti d'une mise en pratique sur ordinateurs.

## 2.3 SUPPORTS DE COURS

Ce document est un support de cours : il constitue le syllabus d'algorithmique, et doit être vu comme un complément aux diaporamas exploités lors des séances de théorie; de plus, certains approfondissements nécessitent l'aide de l'enseignant, et la mise en pratique sur ordinateur est indispensable.

Chaque année, certains compléments à ce syllabus (chapitres ou parties de chapitres supplémentaires) sont fournis aux étudiants via la plateforme e-learning (HELMo Learn). Ils font partie intégrante de la matière.

L'ensemble des ressources utiles est repris dans l'espace dédié à ce cours sur HELMo Learn. Les transparents et les énoncés des activités de laboratoires y seront dévoilés au fur et à mesure de l'avancement du cours.

## 2.4 RÉFÉRENCES PRINCIPALES

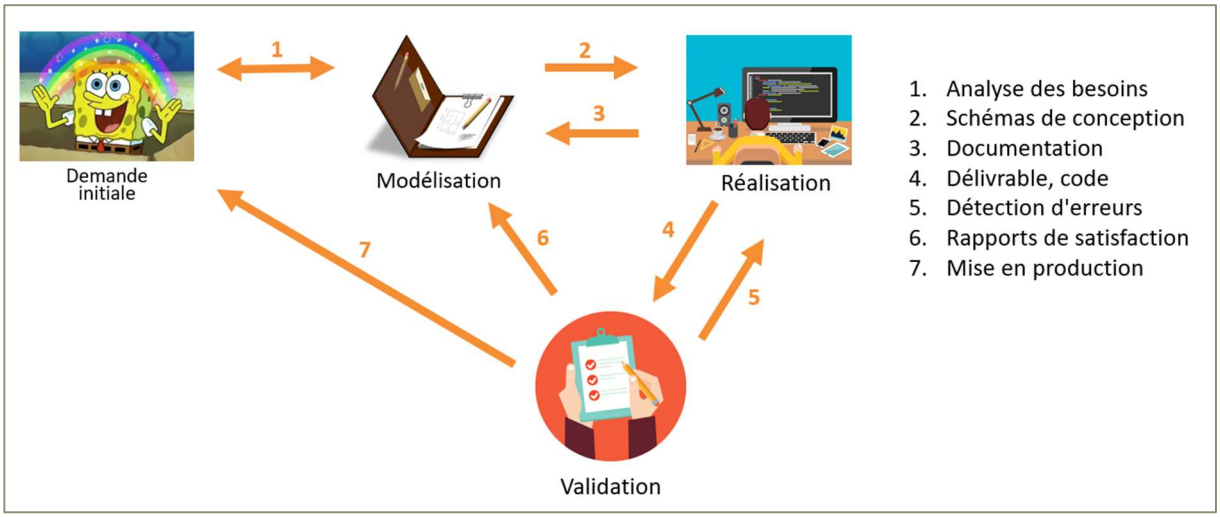

- 🕮 A. Clarinval (09-2002), Syllabus *Concepts et méthodes de la programmation*, HEMES
- 🕮 38 auteurs (2008), *L'art du beau code*, Andy Oram et Greg Wilson, O'Reilly
- 🕮 Robert Sedgewick (2004), *Algorithmes en Java*, Pearson Education
- 🕮 J.-C.Heudin (2016), *Comprendre le Deep Learning - Une introduction aux réseaux de neurones.* Science-eBook.
- 🕮 Florence Pinaud (2018), *#MaVieSous algorithmes – Débats et portraits*. Nathan
- 🕸 Site créé par Pascal LOEWENGUTH et consacré à l'algorithmique et à la programmation, http://lwh.free.fr/pages/algo/algorithmes.htm
- 🕸 Interstices - Site de culture scientifique créé par des chercheurs, lancé à l'initiative de l'INRIA, en partenariat avec le CNRS, les Universités et l'ASTI, http://interstices.info/jcms/jalios_5127/accueil
- 🕸 Divers documents au format pdf provenant du site de l'EPFL - Ecole Polytechnique Fédérale de Lausanne (2006) - http://www.epfl.ch/
- 🕸 Commons (2017, Décembre 22). *Neurone formel*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Neurone_formel
- 🕸 Dr Alexandre, L. (2017, Mai). *Laurent Alexandre - Nous sommes les idiots utiles de l'Intelligence Artificielle*. Récupéré sur Regards connectés: http://regards-connectes.fr/laurent-alexandre/
- 🕸 Hockney, A. (2018, Février 16). *Ask a data scientist - What is Machine Learning?* Récupéré sur CodeAcademy news: https://news.codecademy.com/what-is-machine-learning/
- 🕸 Lin, C., Vinegar, Z., Ross, E., & Silverman, J. (s.d.). Artificial Neural Networks. Consulté en Février 2018, sur Brilliant.org: https://brilliant.org/courses/artificial-neural-networks/
- 🕸 Documentation officielle du langage Java, et tutoriels officiels.
- ➔ Voir la webographie sur HELMo Learn pour plus de références.

# 3 INTRODUCTION – NOTION D'ALGORITHME

## 3.1 RAPPELS DES ÉTAPES DE « FABRICATION » D'UN PROGRAMME

Vous avez étudié les méthodes d'analyse **MERISE** et **UML**, qui permettent

- D'identifier clairement les besoins des clients, les futurs utilisateurs de l'application à développer ;
- De traduire ces besoins en modèles d'analyse.




*©Vincent Martin – Cours d'analyse du bloc 1*

**Dans le cadre du cours d'algorithmique, nous nous focaliserons sur les étapes 2 et 3 :**

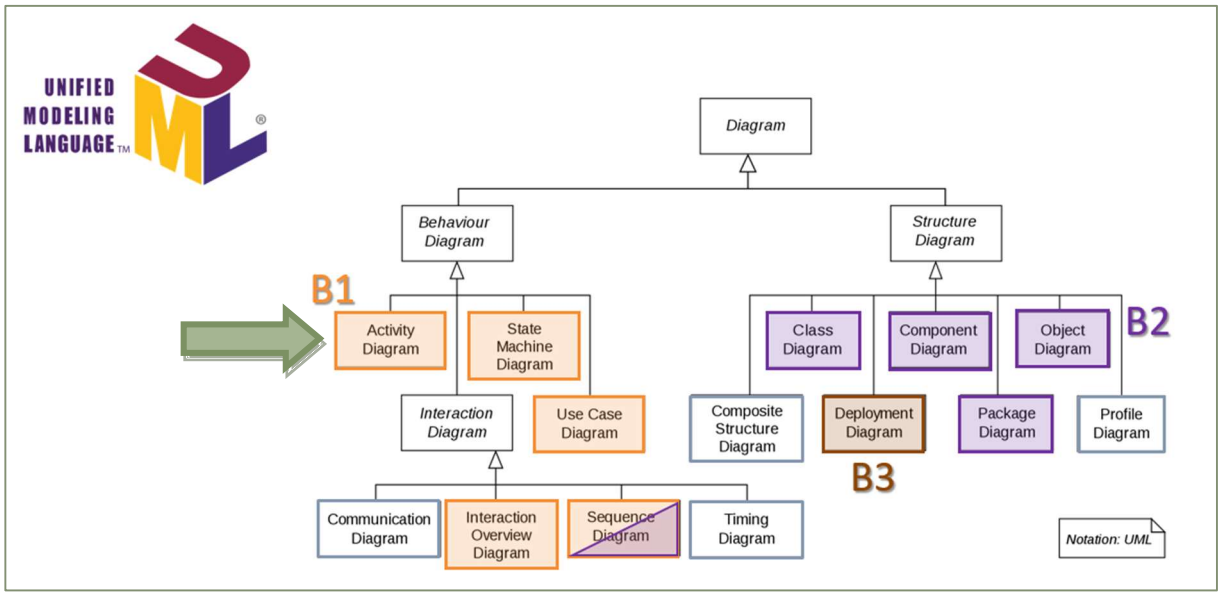
- Nous nous intéresserons à certaines tâches récurrentes dans le monde de la programmation, comme le tri, la recherche d'informations, ou la gestion de files d'attente ;
- Nous réfléchirons à des solutions les plus indépendantes possibles du langage choisi pour les implémenter, et du système sur lequel elles seront exécutées.

Néanmoins, **les laboratoires nous permettront d'illustrer notre propos à l'aide du langage Java, et d'étudier les structures de données complexes qu'il propose.**

Parmi les différents modèles[1] UML existant et que vous avez étudiés au premier quadrimestre, le diagramme d'activités pourra nous être d'une grande utilité. En effet, on

[1] Un modèle est une représentation abstraite et simplifiée (i.e. qui exclut certains détails), d'une entité (phénomène, processus, système, etc.) du monde réel en vue de le décrire, de l'expliquer ou de le prévoir. Concrètement, un modèle permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. Il reflète ce que le concepteur

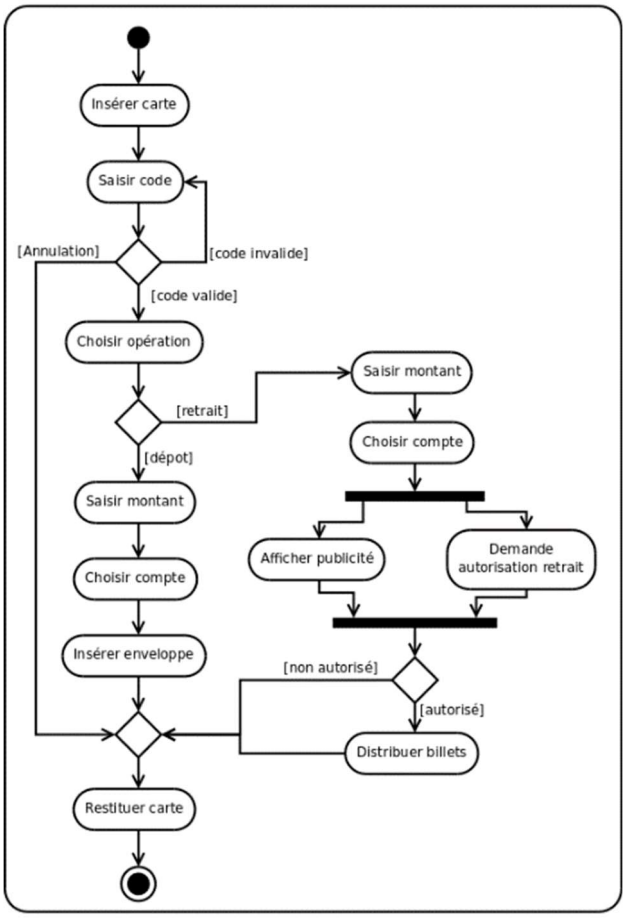
peut attacher un diagramme d'activités à n'importe quel élément de modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément. Ainsi, ils peuvent être utiles dans la phase de réalisation, car ils permettent une description précise des opérations à traduire en code, et sont particulièrement intéressants pour des opérations dont le comportement est complexe ou sensible.




*©Vincent Martin – Cours d'analyse du bloc 1*

croit important pour la compréhension et la prédiction du phénomène modélisé. https://laurent-audibert.developpez.com/Cours-UML/?page=introduction-modelisation-objet

Exemple modélisant le fonctionnement d'une borne bancaire, issu de https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-activites :




## 3.2 DÉFINITIONS

Qu'est-ce qu'un algorithme? Comment l'expliquer précisément et complètement ? Commençons par citer quelques définitions informelles, qui aident à comprendre ce concept, puis passons à des définitions plus explicites, rigoureuses et détaillées.

### 3.2.1 Définitions informelles

- Procédure bien définie permettant de résoudre un problème.
- Séquence d'étapes qui transforme les données en entrée *(input)* en données en sortie (*output*).
- Un algorithme, pour une tâche donnée, est la décomposition de cette tâche en une séquence finie d'étapes élémentaires permettant de la réaliser.

### 3.2.2 Définitions formelles

- Un **algorithme** est une **suite finie** de règles à appliquer, dans un **ordre déterminé**, à un **nombre fini** de données, pour arriver, en un **nombre fini** d'étapes, à un résultat, *et cela quelles que soient les données traitées*. [Encyclopedia Universalis]

- Un **algorithme** est une méthode de **composition d'opérations** conduisant à une **solution certaine** de tous les problèmes appartenant à une classe bien définie (fascicule de concepts et méthodes – A. Clarinval)

Un algorithme correspond donc à la partie **conceptuelle** d'un programme, indépendante du langage de programmation. Le programme est alors **l'implémentation** de l'algorithme, dans un langage de programmation (et sur un système) particulier.

**En résumé** : programme = algorithme exprimé dans un langage donné.
**Mais** à un problème donné correspond·ent 0, 1 ou plusieurs algorithme(s) solution(s).

## 3.3 EXEMPLE DU LABYRINTHE DE PLEDGE

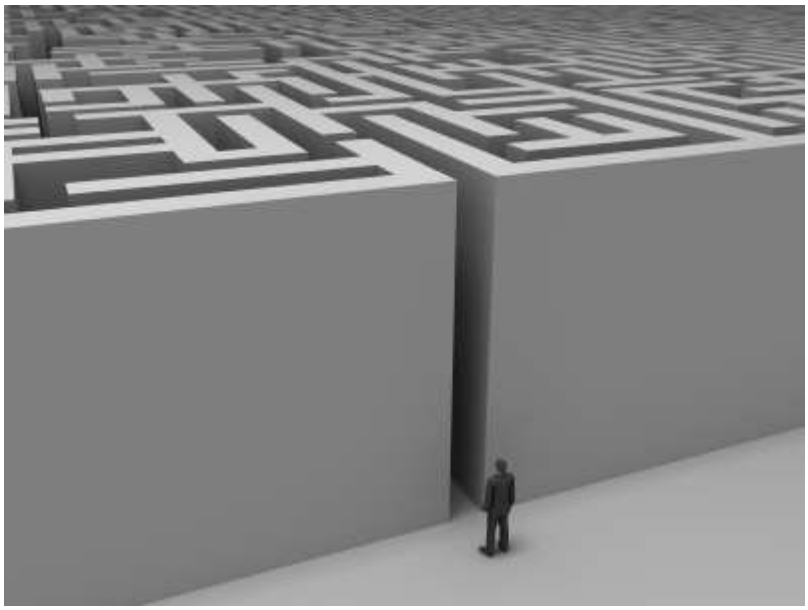


*Figure 1 - Labyrinthe (Source: labyrinthe-géométrique-orientation-619914 par Taken sous licence CC0 Public Domain )*

**Comment sortir d'un labyrinthe plongé dans l'obscurité ?**[2]
Plusieurs stratégies peuvent être imaginées, mais elles ne fonctionnent pas nécessairement pour toutes les configurations possibles de labyrinthes…

Supposons que tous les angles du labyrinthe soient droits. On n'a alors que deux possibilités, tourner à droite ou à gauche selon un angle de 90°.
On compte les changements de direction en augmentant d'un point lorsque l'on tourne à gauche et en diminuant d'un point lorsque l'on tourne à droite (y compris la première fois que l'on tourne à droite quand on atteint un mur).

Au début, le décompte est à zéro. Les deux actions à répéter sont alors les suivantes :

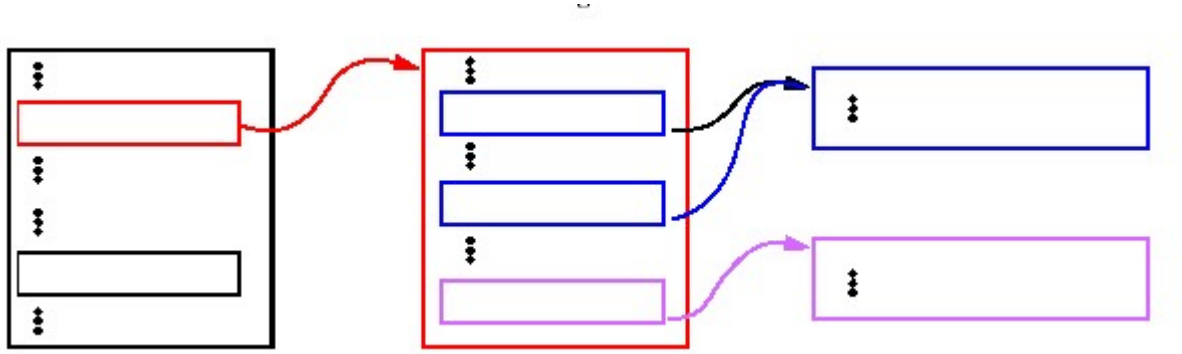
1. Aller tout droit jusqu'au mur, passer à l'instruction 2 ;
2. Longer le mur par la droite (ou par la gauche, mais toujours dans le même sens) jusqu'à ce que le décompte des changements de direction atteigne zéro, passer à l'instruction 1 ;

Cet algorithme porte le nom de celui qui l'a découvert : un petit garçon de douze ans, qui s'appelait John Pledge !

[2] Voir sur http://interstices.info/jcms/c_46065/l-algorithme-de-pledge.

# 4 ÉLABORATION DE SOLUTIONS ALGORITHMIQUES: APPROCHES LES PLUS FRÉQUENTES

On résout généralement un problème par une ***analyse descendante***, c'est-à-dire une analyse **du plus général au plus spécifique**. Cette analyse se fait à l'aide de **blocs imbriqués**[3] correspondant à des traitements de plus en plus spécifiques, eux aussi décrits à l'aide **d'algorithmes**.[4]



Il n'existe pas de méthode miracle pour fabriquer les solutions algorithmiques requises pour les problèmes auxquels l'informaticien peut être confronté. Cependant, un certain nombre de schémas généraux de résolution peuvent être utilisés. Parmi ces schémas, on peut citer les paradigmes[5] suivants :

- L'approche **incrémentale**: à chaque étape de l'algorithme, on ajoute un élément à la solution (exemple: tri par insertion ou tri à bulle)
- La recherche **exhaustive**: on teste toutes les solutions possibles! (Potentiellement un nombre infini...). Exemple: attaque par force brute pour trouver un mot de passe.
- L'algorithme **glouton**: on effectue un choix optimum local, en espérant obtenir un optimum global. Exemple du rendu de monnaie: on commence par rendre la pièce ou le billet le plus grand, qui soit plus petit que la somme à rendre

---

[3] Notons le parallèle évident avec l'implémentation du programme décomposé en fonctions

[4] La méthode de résolution est donc itérative: on résout par analyses successives, à des niveaux de détails de plus en plus précis.

[5] Paradigme : conception théorique dominante ayant cours à une certaine époque dans une communauté scientifique donnée, qui fonde les types d'explication envisageables, et les types de faits à découvrir dans une science donnée. (Définition du Centre National de Ressources Textuelles et Lexicales - France)

- **Diviser pour régner**: **divise** le problème en un certain nombre de sous-problèmes, **règne** sur les sous-problèmes en les résolvant directement, ou de manière récursive, **recombine** les solutions obtenues des sous-problèmes pour en déduire la solution du problème de départ. Exemples: tri rapide (*quicksort*), *heapsort…*

  Le schéma général d'une approche «diviser pour régner» est donc le suivant:

  Pour résoudre un problème P traitant des données D:
  1. si les données sont suffisamment simples ou petites, appliquer un algorithme existant pour ces cas simples,
  2. sinon: décomposer les données en sous-ensembles plus petits, $d_1$, ..., $d_n$ puis, pour chaque sous-ensemble,
     - résoudre le problème P pour ce sous-ensemble $d_i$ afin d'obtenir une solution partielle $s_i$
     - recombiner les solutions $s_i$ pour obtenir une solution globale S au problème P

# 5 TYPES D'ALGORITHMES ET PRINCIPAUX DOMAINES D'UTILISATION

Historiquement, **les algorithmes ont d'abord été dédiés à la résolution de problèmes arithmétiques**, comme le calcul du plus grand commun diviseur (PGCD) de deux nombres (algorithme d'Euclide) ou la résolution d'équations. **Ils ont été formalisés bien plus tard avec l'avènement de la logique mathématique** (étude des mathématiques en tant que langage**) et l'émergence des ordinateurs**, qui permettaient de les mettre en œuvre de manière automatisée. C'est une femme, la mathématicienne et pionnière des sciences informatiques Ada Lovelace, qui, en 1843, réalise la première implémentation d'un algorithme sous forme de programme (calcul des nombres de Bernoulli).

Parmi les formalismes utilisés, citons par exemple celui des machines de Turing, qui permettent de définir précisément ce qu'on entend par "étapes", par "précis" et par "non ambigu" et qui donnent un cadre scientifique pour étudier les propriétés des algorithmes. Cependant, ce n'est pas le seul, et suivant le formalisme choisi, on obtient des approches algorithmiques différentes pour résoudre un même problème. Par exemple l'algorithmique récursive, l'algorithmique parallèle ou l'informatique quantique donnent lieu à des présentations d'algorithmes différentes de celles de l'algorithmique itérative.

**Grâce à l'informatique, l'algorithmique s'est beaucoup développée dans la deuxième moitié du XXe siècle**, notamment par le travail de Donald Knuth, auteur du traité « The Art of Computer Programming », qui décrit de très nombreux algorithmes.

On peut distinguer:

- des **algorithmes généralistes**, qui s'appliquent à toute donnée (numérique ou non numérique) : par exemple les algorithmes liés au chiffrement, ou qui permettent de mémoriser les données ou de les transmettre ;
- des **algorithmes dédiés à un type de données particulier** (par exemple ceux liés au traitement d'images).

Il existe un certain nombre d'**algorithmes classiques**, utilisés pour résoudre des problèmes ou plus simplement pour illustrer des méthodes de programmation[6] :

- Algorithmes ou problèmes classiques (du plus simple ou plus complexe)
    - échange, ou comment échanger les valeurs de deux variables : problème classique illustrant la notion de variable informatique
    - Algorithmes de recherche, ou comment retrouver une information dans un ensemble structuré ou non (par exemple, la recherche dichotomique)
    - algorithme de tri, ou comment trier un ensemble de nombres le plus rapidement possible ou en utilisant le moins de ressources possible
    - problème du voyageur de commerce, problème du sac à dos, problème SAT et autres algorithmes ou approximations de solutions pour les problèmes combinatoires difficiles (dit NP-complets)
- Algorithmes ou problèmes illustrant la programmation récursive
    - tours de Hanoï
    - huit dames, placer huit dames sur un échiquier sans qu'elles puissent se prendre entre elles,
    - suite de Conway,
    - algorithme de dessins récursifs (fractale) pour le Tapis de Sierpiński, la Courbe du dragon, le Flocon de Koch…
- Algorithmes dans le domaine des mathématiques
    - calcul de la factorielle d'un nombre, de la Fonction d'Ackermann ou de la suite de Fibonacci,
    - algorithme du simplexe,
    - fraction continue d'un nombre quadratique, permettant d'extraire une racine carrée, cas particulier de la méthode de Newton
    - dans le domaine de l'algèbre : l'algorithme d'unification, le calcul d'une base de Gröbner d'un idéal de polynôme et plus généralement presque toutes les méthodes de calcul symbolique,
    - théorie des graphes, qui donne lieu à de nombreux algorithmes, notamment pour trouver le chemin optimal entre deux points,
    - test de primalité.
- Algorithmes pour et dans le domaine de l'informatique
    - cryptologie et compression de données
    - Informatique musicale
    - algorithme génétique, informatique décisionnelle
    - Analyse et compilation des langages formels (voir Compilateur et Interprète)
    - allocation de mémoire (ramasse-miettes)

---

[6] Cette liste d'algorithmes est issue de l'encyclopédie en ligne Wikipedia, page https://fr.wikipedia.org/wiki/Algorithmique , consultée le 11 mai 2018. Le lecteur intéressé pourra également explorer la liste d'algorithmes fournie à la page https://fr.wikipedia.org/wiki/Liste_d%27algorithmes

Si on les classe par domaine, **on retrouve aujourd'hui des algorithmes dans de très nombreuses matières**, telles que :

- la cryptographie, assurant la confidentialité, l'authenticité et l'intégrité de messages ou de données,
- le routage d'informations, mécanisme par lequel des chemins sont sélectionnés dans un réseau pour acheminer les données d'un expéditeur jusqu'à un ou plusieurs destinataires. Le routage est une tâche exécutée dans de nombreux réseaux, tels que le réseau téléphonique, les réseaux de données électroniques comme Internet, mais aussi les réseaux de transports
- le calcul d'itinéraire optimal, par exemple celui effectué par un GPS,
- la planification (l'organisation dans le temps de la réalisation d'objectifs, sur une durée et avec des étapes précises) et l'utilisation optimale des ressources,
- le traitement d'images, qu'elles soient médicales, géographiques ou autres,
- le traitement de texte en particulier, les logiciels de bureautique en général,
- la bio-informatique, champ de recherche multidisciplinaire où travaillent de concert biologistes, médecins, informaticiens, mathématiciens, physiciens et bio-informaticiens, dans le but de résoudre un problème scientifique posé par la biologie[7]
- les jeux, la réalité virtuelle,
- les tactiques sportives,
- la gestion d'un portefeuille boursier,
- le diagnostic médical,
- la gestion du « big data » (établissement de profils de consommation par exemple)[8]
- la traduction, le Natural Language Processing (exemple : ChatBot pour l'assistance automatisée aux clients sur un site d'e-commerce)
- etc.

Dans la suite de ce cours, nous nous focaliserons sur certains **exemples informatiques classiques**, comme les algorithmes de tri ou la gestion efficace de grandes collections de données.

Les **travaux pratiques** en programmation orientée objet, algorithmique, et pour l'activité intégrative, permettront la mise en œuvre d'autres algorithmes (sac à dos ou minimax par exemple).

[7] Ainsi, l'acronyme NBIC désigne le regroupement des NanoTechnologies, Biotechnologies, de l'Informatique et des sciences Cognitives

[8] Voir https://fr.wikipedia.org/wiki/Big_data pour plus d'exemples

Nous pourrions également nous intéresser tout particulièrement à un domaine en pleine expansion : le Machine Learning, qui fait partie des techniques et algorithmes pour le développement d'une **Intelligence Artificielle.**

# 6 QUALITÉS OU CARACTÉRISTIQUES D'UN ALGORITHME

Avant d'étudier plus en détails certains algorithmes, définissons des critères permettant de les différencier et de décider si cela vaut la peine de les implémenter (ou pas! )

## 6.1 DOMAINE

**Ensemble des valeurs en entrée *(input data*) pour lesquelles l'algorithme donne une solution au problème à résoudre.**

> Ainsi, dans le cas du labyrinthe de Pledge, on doit pouvoir sortir du labyrinthe quelle que soit la taille ou la configuration du labyrinthe et quelle que soit notre position initiale… Mais certaines configurations sont impossibles, et donc à éliminer du domaine.

À vérifier : le **domaine initial du problème doit être égal au domaine de l'algorithme développé**.[9]

## 6.2 EXACTITUDE

**Un algorithme est qualifié d'exact si l'exécution, pour *toute* instance[10] du problème posé, produit le résultat escompté.**

Pour être exact, un algorithme doit d'abord garantir sa **terminaison**: il se terminera après un temps fini. Comment en apporter la preuve? En trouvant une fonction entière positive qui décroit strictement à chaque « pas » de l'algorithme.

Ce n'est évidemment pas suffisant : si l'algorithme se termine en donnant une proposition de solution, alors il faut que cette solution soit **correcte**.

Enfin, l'algorithme doit se terminer et fournir une solution pour chacune des entrées du domaine (**complétude**)

| Terminaison + correction + complétude = exactitude |
|---|

Remarque : ne confondez pas exactitude et **robustesse** : la robustesse mesure la capacité d'un programme à s'exécuter sans erreur, même dans des cas extrêmes (pannes de courant, surchauffe, attaque de pirates informatiques, fichiers ou réseau non disponible…). La robustesse qualifie plutôt un programme qu'un algorithme, car la teneur des mécanismes à

[9] Vous retrouverez la notion de domaine dans le cours de bases de données.

[10] Une instance d'un problème est obtenue en fournissant des valeurs précises pour tous les paramètres du problème (toutes les données en entrée)

mettre en place varie en fonction de l'environnement dans lequel s'exécutera le programme et en fonction des possibilités du langage de programmation dans lequel on implémentera l'algorithme.

Comment prouver ou vérifier l'exactitude d'un algorithme ?

### 6.2.1 Les tests

Pour s'assurer de l'exactitude d'un algorithme, on exécute le programme sur un nombre n d'instances du problème, ce qui produit un ensemble de n résultats qui :

- Soit contient un ou plusieurs résultats erronés : on peut en déduire que l'algorithme est incorrect.
- Soit ne contient pas de résultat erroné : peut-on en déduire avec certitude que l'algorithme est correct ?

Exemple 1: trouver la valeur absolue d'un entier `i`

Solution proposée : `-i` est la valeur absolue de `i`

Correct ?

Exemple 2 : déterminer si un nombre est premier

Solution proposée :

```
Entrer x
SI x modulo 2 = 1
ALORS x est premier
SINON x n'est pas premier
FIN SI
```

Correct ?

**Conclusion : les tests peuvent servir à montrer la présence d'erreurs, non leur absence !**

***Il s'agit donc d'une technique d'appoint aidant à déceler les erreurs de détail qui auraient pu passer inaperçues aux stades précédents de la mise au point.***

*Une autre leçon importante [...] est que les essais effectués pour évaluer la validité d'un programme ne sauraient être quelconques si l'on veut obtenir des renseignements significatifs. En particulier,* ***il est illusoire d'attendre beaucoup des tests d'un programme s'ils n'ont pas été prévus dès l'écriture de celui-ci : si tests il y a, ils constituent une étape prévue et planifiée du développement, et non une vérification supplémentaire effectuée sur un programme qui a été conçu sans tenir compte de cette contrainte.***

*Notons que la construction de "jeux d'essai" représentatifs pour un programme donné est une tâche non triviale : il s'agit de choisir un ensemble de cas permettant de tester, sinon chaque branche du programme (car le nombre de cas possibles est infini lorsque le programme contient des boucles indéfinies), du moins un ensemble représentatif. [...]*

*D'un point de vue pratique, on notera que dans une importante équipe de programmation il est impératif de confier le test d'un élément de programme à un groupe différent de celui qui a écrit cet élément : les programmeurs écrivent souvent leurs programmes en ayant en tête, plus ou moins consciemment, un cas de traitement qui n'est pas forcément le plus général, et ont tendance à ne tester que ce cas particulier.*

Comment doit être constitué un **jeu de données d'essai** ? Il ne doit pas être improvisé, mais raisonné; il doit obéir à un **plan de tes**t.

– Le jeu d'essai doit contenir un échantillon des différents ***cas*** distingués par l'algorithme,
– y compris les ***cas d'erreurs*** reconnues et traitées par le programme;
– il doit toujours comprendre les ***cas limite***s,
  – en particulier, celui de l'***ensemble vide*** ou de la valeur 0
– et, pour la détection des dépassements de capacité dans les calculs, ceux des ***valeurs extrême***s;
– enfin — le plus difficile ! —, un maximum de ***combinaisons*** de cas devrait être testé (les tester *tous* est, pratiquement et même théoriquement, impossible).

*Exemple illustrant l'intérêt de tester les cas limites :*
Supposons que les jours de la semaine soient indiqués par les nombres 1 à 7. La première idée qui vient à l'esprit pour déterminer le jour suivant est de programmer :

```
demain = aujourdhui+1;
```

si `aujourdhui = 7`, le calcul donne `8` alors que le résultat doit être `1`. On corrige donc comme ceci :

```
demain = reste(aujourdhui+1,7);
```

le résultat est maintenant correct pour `aujourdhui = 7`,

mais il devient incorrect si `aujourdhui = 6` ! La programmation valide est finalement : `demain = reste(aujourdhui,7)+1`.[11]

Les tests sont également capitaux lorsqu'on opère des modifications majeures sur un programme existant : 80% du travail d'un développeur consiste en effet à assurer la maintenance (correction de bugs) et la mise à jour de logiciels existant, qu'il n'a généralement pas écrits ! Il faut pouvoir garantir par exemple que l'ajout d'une fonctionnalité n'a pas eu d'impact négatif sur celles qui existaient déjà, ou que la modification de l'une n'a pas altéré son fonctionnement ou rendu d'autres fonctions inopérantes ou erronées. C'est d'ailleurs l'un des objectifs du **développement piloté par les tests** (*Test Driven Development*, ***TDD***), méthode préconisant d'écrire les tests unitaires avant même le code source. Même dans ce type de démarche, **certaines fonctionnalités particulières dont l'efficacité est cruciale nécessitent l'élaboration soigneuse d'un plan de test**.

### 6.2.2 Utilisation d'une référence

Une autre manière de prouver l'exactitude d'un algorithme consiste à invoquer le fait que le programme est basé sur une méthode de résolution du problème dont la qualité est reconnue. Pour le prouver, on peut recourir à plusieurs méthodes :

- L'**argumentation**: on fournit une explication rigoureuse des différentes étapes rencontrées dans le programme. Celle-ci doit se faire pour tous les types de données en entrée.
- La **démonstration** :

**Démontrer** un programme, c'est déduire du *texte* même de ce programme les propriétés de son *exécution*, théoriquement avec la même sécurité que le logicien, au départ des prémisses d'un raisonnement, déduit la conclusion.

La question de la démonstration des programmes a été une des questions centrales de la théorie de la **programmation structurée** développée aux alentours de 1970, principalement par E.W. DIJKSTRA et C.A.R. HOARE. Au terme de ce débat qui a duré plusieurs années, la conclusion est qu'une telle démonstration est *théoriquement possible mais impraticabl*e.

*Est-ce à dire, alors, que les démonstrations ne sont d'aucun intérêt pour le programmeur ? Non pas ! La leçon [...] est qu'il est vain (en général) de chercher à démontrer à posteriori la validité d'un programme existant. Par contre, une méthode de programmation particulièrement fructueuse, proposée initialement par Dijkstra, consiste à* ***développer le programme à partir de la démonstration*** *ou, plus généralement, à développer simultanément*

[11] Extrait du chapitre 10 du fascicule de Concepts et Méthodes de la Programmation – André Clarinval – 09/2002

*programme et démonstration. [...] les assortir d'arguments convaincants, fournissant des ébauches de démonstrations possibles; apporter un soin particulier aux boucles, en indiquant aussi souvent que possible les invariants associés; construire de façon rigoureuse les éléments les plus délicats du programme et ceux dont la validité est particulièrement cruciale, en démontrant complètement qu'ils sont corrects; et écrire les autres éléments de façon à être convaincu (et à convaincre le lecteur) qu'ils pourraient être démontrés si cela était demandé (de même, lorsqu'on étudie une langue étrangère, accepte-t-on de faire des "impasses" en lisant un roman écrit dans cette langue, car la recherche systématique de toutes les tournures empêcherait d'aller jusqu'au bout; mais on isole de temps en temps une page témoin, correspondant par exemple à un moment important de l'action, qu'on s'efforce d'analyser en détail et peut-être de traduire, pour se persuader qu'il n'y a pas d'impossibilité théorique à une compréhension totale).*

***L'idée selon laquelle un programme doit être écrit comme si on pouvait avoir à le justifier par une démonstration nous parait de nature à améliorer considérablement la qualité des programmes produits.***

*Quelle est la différence essentielle, pourra-t-on demander, entre la démarche proposée et la programmation habituelle ? D'une certaine manière, tout programme suppose une démonstration plus ou moins consciente de sa validité; plus précisément, on rédige un programme à partir d'une conviction, justifiée ou non, que chacune de ses instructions doit amener à un certain état de programme.*

*[...]*

*Cette remarque éclaire le principe de la méthode de programmation suggérée : il s'agit essentiellement de* ***rendre explicites toutes les hypothèses qui sont obligatoirement présentes dans l'esprit du programmeur lorsqu'il écrit un programme, et qui pouvaient ne pas être formulées clairement au départ.***

Le langage Java propose une méthode spéciale **assert** ***(expr_booléenne ): "message"***, qui teste l'expression fournie et, si celle-ci n'est pas vérifiée, clôture l'exécution du programme après avoir affiché le message indiqué.

Concrètement, et comme le suggère le texte reproduit ci-dessus, pour étayer ses ébauches de démonstration, le programmeur devrait ***expliciter ses hypothèse***s.

À quels endroits d'un programme peut-on trouver semblables sous-entendus et placer des assertions qui les explicitent ?

– À l'*entrée* dans un **sous-programme** appelé (précondition portant sur les paramètres) et au *retour* dans le sous-programme appelant (postcondition portant sur les résultats du sous-programme appelé).

– Avant d'entrer dans une construction de **sélection** (SI ou SELON) qui ne teste pas tous les cas *théoriquement possible*s, mais seulement les cas *possibles en pratique* (précondition sur les cas possibles).

– Avant d'entamer une **boucle** (itérative ou récursive) et après la fin de la boucle (test des *modifications* d'état induites par l'exécution de la boucle), au début de chaque itération de la boucle (test des *invariants* de la boucle, c'est-à-dire les choses non changeantes).

*Illustration en Java : calcul de la puissance entière d'un nombre réel :* $N^P$ .

Créez un projet Eclipse et activez les assertions via les propriétés du projet (dans la catégorie "Run", complétez les *VM options* en indiquant `-enableassertions`)

```
static float puissance (float nombre, int exposant) {
  // Précondition
  assert (exposant >= 0 || nombre != 0):
    "erreur - division par zéro";
  float resultat = 1;
  int compteur = exposant;
  // À ce stade, compteur == exposant
  // la boucle qui suit ne sera exécutée que si exposant > 0
  while( compteur > 0) {
     resultat *= nombre; compteur -= 1;
  } // À ce stade, compteur <= 0;
  assert (exposant >= 0 && compteur == 0 ||
          exposant < 0 && compteur == exposant);
  // La boucle qui suit ne sera exécutée que si exposant < 0
  while (compteur < 0) {
     resultat /= nombre; compteur += 1;
  } // À ce stade, compteur == 0
  assert (compteur == 0);
  return resultat;
}
```

L'appel suivant `puissance(0.0f,-3)` provoquera l'affichage suivant :

```
Exception in thread "main" java.lang.AssertionError: erreur -
division par zéro
At algoexempleassertpuissance.AlgoExempleAssertPuissance.puissance
(AlgoExempleAssertPuissance.java:21)
at algoexempleassertpuissance.AlgoExempleAssertPuissance.main
(AlgoExempleAssertPuissance.java:18)
Java Result: 1
```

## 6.3 EFFICACITÉ

**L'efficacité d'un algorithme est une mesure des ressources à mettre en œuvre pour l'exécuter.**

Elle permet de qualifier un algorithme et de discriminer entre différents algorithmes solutionnant le même problème. **On cherche à minimiser la quantité de ressources mises en œuvre,** à savoir:

- le nombre d'opérations effectuées (mesure du temps d'exécution)
- l'espace mémoire requis
- le nombre de transferts entre zones de stockage

### 6.3.1 Nombre d'opérations

Une étape algorithmique (opération) prend du temps pour s'exécuter. Cela est lié au nombre d'opérations machines fondamentales exécutées par le microprocesseur. Donc **le nombre d'opérations est une mesure du temps d'exécution, indépendante du type de processeur sur lequel le programme s'exécutera au final**.

### 6.3.2 Espace mémoire

On s'intéresse ici à l'espace mémoire temporaire dont le programme a besoin pour fonctionner (quantité de mémoire RAM). Généralement on essaie de réaliser un compromis entre l'espace et le temps nécessaires.

### 6.3.3 Transferts

Les transferts en RAM sont bien plus rapides que les transferts vers le disque dur, eux-mêmes plus rapides que les transferts via un réseau. Il est donc important d'en tenir compte.

**Ces 3 paramètres permettent de "mesurer" 3 types d'efficacité :**

- l'efficacité **inhérente à l'algorithme**. On quantifie alors les trois composantes ci-dessus lorsque la taille du problème croît, comme, par exemple, pour trier un tableau de $10^3$, $10^4$ ou $10^5$ éléments. Cela peut se faire par simple examen de la structure du programme.
- l'efficacité **dans un contexte technologique donné** : on évalue les mécanismes du système d'exploitation, comme la gestion de la mémoire virtuelle par exemple.
- l'efficacité **spécifique à un équipement donné** : on cible dans ce cas la rapidité des disques durs, de la mémoire RAM, des processeurs, ...

Dans la suite du cours, **nous allons nous focaliser sur le temps et l'espace pour mesurer l'efficacité inhérente à l'algorithme**.

## 6.4 EXEMPLE : RECHERCHE PAR DICHOTOMIE

Objectif: déterminer si un élément *x* appartient à un ensemble E d'éléments triés :

$E = \{x_1, \ldots, x_n\}$

- Entrées du programme de recherche: x (l'élément recherché) et E (qui peut se présenter sous la forme d'un tableau **trié**)
- Sortie: *vrai* ou *faux* selon que *x* appartient ou non à E.
- Schéma de résolution: appartient(x,E):

```
si E est vide, la solution est faux
sinon
  si E = {x1}, la solution est vrai si x=x1,
                                  faux sinon
  sinon (E = {x1, ..., xn})
     décomposer E comme suit:
          Emilieu = {xn/2} avec n/2 la division entière de n
                            par 2
          Egauche = {x1, ..., x(n/2)-1}
          Edroite = {x(n/2)+1, ..., xn}
     si appartient(x,Emilieu) est vrai,
     alors la solution est vrai
     sinon si x < xn/2
       alors la solution = appartient(x, Egauche)
       sinon la solution = appartient(x, Edroite)
```

L'idée de la solution (dans le cas général d'un ensemble de plus d'un élément) est donc :

a) Comparer *x* avec l'élément situé au milieu de l'intervalle;
b) si *x* est plus petit, le rechercher de la même manière dans le demi-intervalle de gauche;
c) si *x* est plus grand, le rechercher de la même manière dans le demi-intervalle de droite.

### 6.4.1 Jeu de tests à utiliser :

1) Cas à distinguer par rapport au tableau.

– Puisque l'algorithme se fonde sur une division par 2, il doit être testé pour deux tableaux contenant un nombre pair et un nombre impair d'éléments, c'est-à-dire pour les deux cas de la division exacte et de la division avec reste.

– Doivent également être testés les cas limites de la recherche dans un ensemble vide et dans un ensemble singleton.

2) Cas à distinguer par rapport à l'élément cherché.

– L'algorithme doit être testé pour la recherche de trois éléments présents dans le tableau, situés respectivement à la position médiane (cas a), dans la moitié de gauche (cas b) et dans la moitié de droite (cas c).

– La recherche d'un élément non présent dans le tableau (cas d'échec) doit également être testée.

3) Combinaisons de cas.

– Les quatre cas du point 2 doivent être testés pour chacun des types de tableaux distingués au point 1

– La recherche d'un élément non présent dans le tableau peut être subdivisée en plusieurs cas (élément plus petit ou plus grand que tous les éléments, élément dont la valeur se situe entre 2 éléments du tableau)

### 6.4.2 Caractéristiques/Qualités

Le **domaine** de cet algorithme est défini par les données en entrée (essentiellement le tableau et l'élément recherché) et les préconditions qui s'appliquent à elles (la principale étant que le tableau doit être trié).

La propriété de **complétude** assure que l'algorithme fournira la solution pour toute donnée en entrée respectant les préconditions, c'est-à-dire pour tout jeu de valeurs appartenant au domaine de l'algorithme.

Son **exactitude** peut être vérifiée par un plan de tests soigneusement conçu (voir ci-dessus) mais celui-ci ne suffit pas à garantir l'absence d'erreur. Elle peut être prouvée par utilisation d'une référence (la recherche dichotomique est un grand classique! ) si on démontre que notre algorithme respecte bien cette méthode de résolution reconnue.

Son **efficacité** sera étudiée dans le prochain chapitre.

# 7 NOTION DE COMPLEXITÉ

## 7.1 INTRODUCTION

Dans le chapitre précédent, nous avons vu que l'efficacité d'un algorithme peut être mesurée en temps d'exécution, en quantité mémoire requise ou en nombre de transferts nécessaires.

Cependant, les notions de temps de traitement et d'espace mémoire sont dépendants de la machine physique utilisée pour implémenter l'algorithme, on a donc besoin d'une **mesure absolue, indépendante de toute implémentation**. De plus, nous souhaitons quantifier l'efficacité d'un algorithme **lorsqu'il est amené à traiter un nombre de données important**, car c'est dans ce cas que la rapidité ou l'économie mémoire deviennent cruciaux.

Un tel mode de mesure sera appelé la ***complexité (algorithmique) du problème à résoudre***.

Partons d'un exemple en Java. Dans un programme simulant le tirage du loto, la fonction de calcul des numéros de boules doit s'assurer qu'elle ne délivre pas de numéros en double.

```
public class GrilleLotto {
   private int[] grille = new int[6 + 1]; // Numéros tirés
   private int nbreTiragesEffectues = 0;

   private boolean numéro_unique (int numéro){
      /*… Voir plus loin*/
   }

   public int tirage_boule(){ // Utilise numéro_unique()
      int n;
      do {
         n = 1 + ((int)(Math.random() * 42)); // n dans[1,42]
      } while (!numéro_unique(n)) ;

      // Mise à jour de la grille
      this.grille[nbreTiragesEffectues] = n;
      nbreTiragesEffectues += 1;

      return n;
   }
}
```

La version que voici de la fonction de vérification numéro_unique compare le nouveau numéro à chacun de ceux qui ont déjà été délivrés.

```
private boolean numéro_unique (int numéro){
   for (int indiceTirage = 0;
        indiceTirage < nbreTiragesEffectues;
        indiceTirage++){
      if (this.grille[indiceTirage] == numéro)
         return false;
   }
   return true;
}
```

La version suivante de la même fonction utilise un tableau de booléens indiquant l'état "tiré ou disponible" des 42 numéros possibles. La vérification d'un nouveau numéro nécessite d'examiner dans ce tableau la seule position correspondant au numéro.

```
private static boolean[] déjà_tiré = // numéros tirés
   {false, // la position [0] sera inutilisée
          false,false,false,false,false,false,false,
          false,false,false,false,false,false,false,
          false,false,false,false,false,false,false,
          false,false,false,false,false,false,false,
          false,false,false,false,false,false,false,
          false,false,false,false,false,false,false};

private boolean numéro_unique (int numéro)
{
   if (déjà_tiré[numéro]) return false;
   else {
      déjà_tiré[numéro] = true;
      return true;
   }
}
```

**Quelle solution choisir?**

- Du point de vue de l'utilisation de la mémoire RAM, la première version n'a nécessité aucun ajout, tandis que la seconde a entrainé l'utilisation d'un tableau de 43 booléens. On pourrait donc pencher pour la première si l'espace mémoire était vraiment compté.

- Intéressons-nous à présent au temps d'exécution :

- Dans la première version de la fonction `numéro_unique`, la vérification du $n^e$ numéro tiré nécessite l'examen de `n-1` positions dans le tableau représentant la grille du Loto. L'algorithme implémenté par `numéro_unique` nécessite donc, à chaque appel, un nombre d'instructions proportionnel à N, où N désigne le nombre de boules du tirage (6+1 dans le cas du loto).

- Dans la seconde version de la fonction `numéro_unique`, on teste toujours une seule position du vecteur de booléens. La vérification du $n^e$ numéro tiré nécessite un nombre d'instructions fixe (constant), alors qu'il dépend de N dans la version précédente.

- Si, au total, on souhaite tirer N numéros tous différents, on effectuera au minimum[12] N appels à la méthode `numéro_unique.` Dans sa première version, cela occasionnera 0 comparaison au premier appel (la grille est encore vide), 1 comparaison lors du second appel (comparaison du numéro candidat avec le numéro déjà dans la grille), 2 comparaisons lors du troisième appel (comparaison du numéro candidat avec les 2 numéros déjà tirés), etc., soit un nombre total de comparaisons égal à 0+1+2+3+...+(N–1) = **$(N-1) \times N / 2$ comparaisons.**
  Dans la seconde version de `numéro_unique,` on effectuera au minimum **N** comparaisons (on vérifie la valeur du booléen à l'indice correspondant au numéro candidat).
  Sauf pour $N < 3$, on a toujours $N \leq (N-1) \times N / 2$. Mais, surtout, plus N est grand, plus l'écart de performance est grand entre les deux solutions!

- Alors que, dans la seconde version de la fonction `numéro_unique`, le temps d'exécution pour obtenir un tirage du Loto est strictement proportionnel au nombre N de numéros du tirage, dans la première version, il croît plus vite que N. Cela se constate aisément sur le schéma suivant, comparant les 2 courbes dans le cas où le nombre de numéros à tirer augmente, et en posant que le temps d'exécution de chaque test est de 9ms, et que le temps d'exécution de la fonction complète nécessite encore 10ms.

[12] Au *minimu*m, c'est-à-dire si le tirage ne donne effectivement aucun numéro en double et que, donc, il y ait un seul appel à `numéro_unique` pour chaque numéro de la combinaison gagnante.

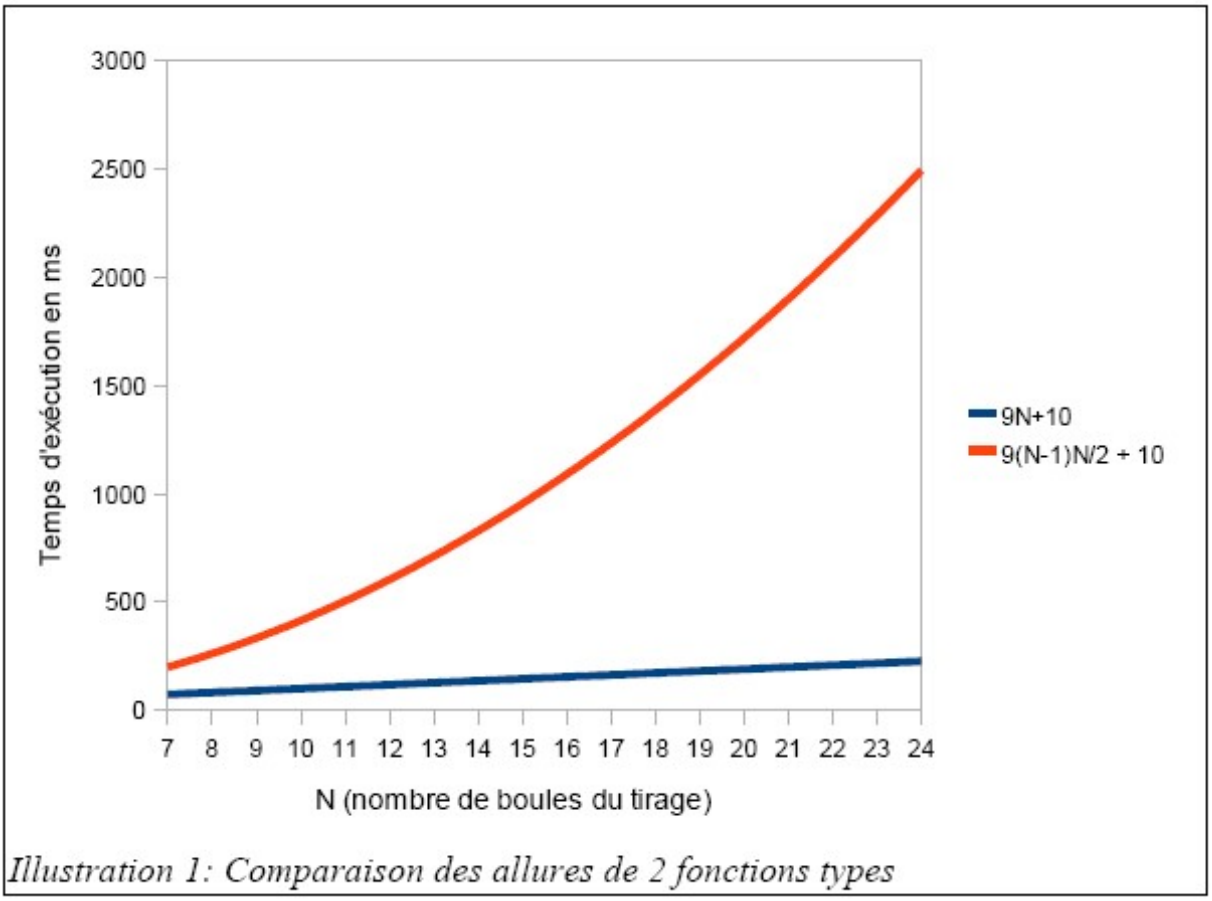
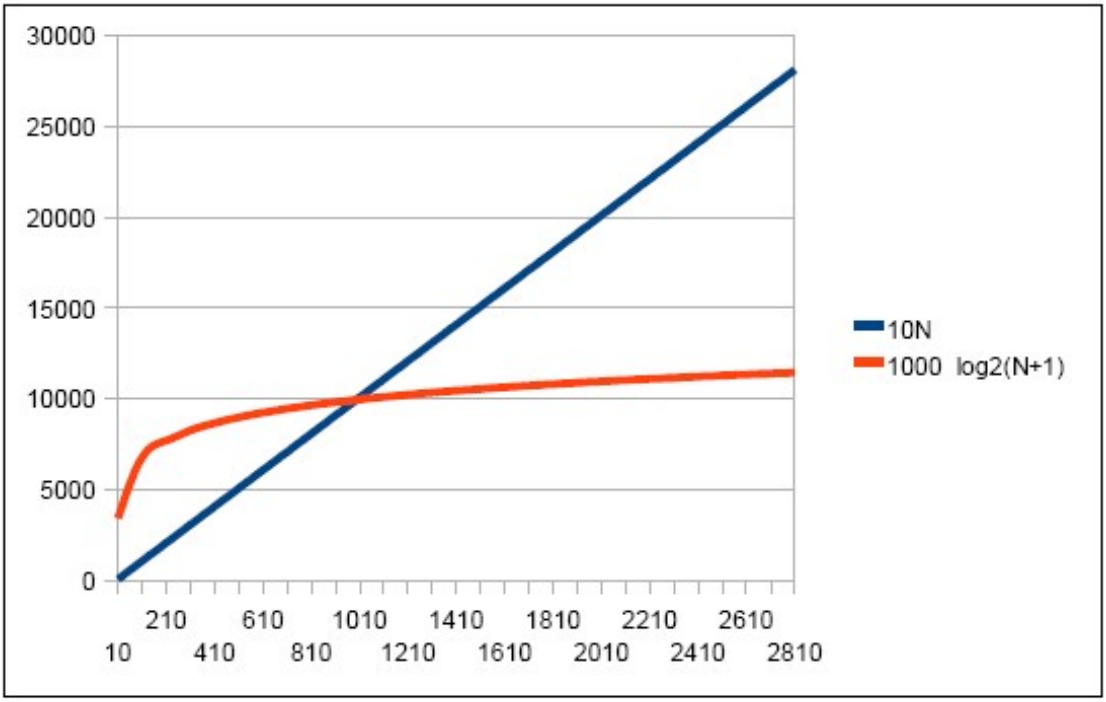



*Illustration 1: Comparaison des allures de 2 fonctions types*

Pour certains algorithmes, le temps d'exécution croît moins vite que le nombre d'éléments traités. C'est le cas de la recherche **dichotomique d'un élément dans un tableau (de dimension N)**, qui nécessite de tester *au plus* $\log_2(N+1)$ positions du tableau. Le paragraphe suivant vous en donnera l'explication.


30000
25000
20000
15000
10000
5000
0
10
210
410
610
810
1010
1210
1410
1610
1810
2010
2210
2410
2610
2810
10N
1000 log2(N+1)

## 7.2 CALCUL DE LA COMPLEXITÉ TEMPORELLE THÉORIQUE D'UN ALGORITHME

**Il est vital de *prévoir et maitriser* la croissance du temps d'exécution d'un programme à mesure qu'augmente le nombre de données traitées**, comme en témoigne Christine Solnon dans le livre « #MaVieSous algorithmes » :

23 / *interview*

# « Certains problèmes sont si complexes... »



**Christine Solnon** / *Spécialiste en IA, Professeur en science informatique à l'INSA de Lyon et chercheuse au laboratoire LIRIS*

### Peut-on résoudre tous les problèmes avec des algorithmes ?

« Certains problèmes sont si complexes que nous n'avons pas d'algorithme pour les résoudre avec un ordinateur. En langage informatique, on dit que ce sont des problèmes "indécidables". En 1936 déjà, le mathématicien anglais Alan Turing avait montré qu'il existait des problèmes que sa machine ne pourrait jamais résoudre. Comme nos ordinateurs fonctionnent exactement sur le même modèle que celui qu'avait conçu Turing, ils ont les mêmes limites.

Certains problèmes plus simples ne peuvent pas trouver de solution informatique pour une question de temps. Le nombre d'opérations de l'algorithme est si grand qu'un ordinateur mettrait plusieurs siècles pour arriver au bout. C'est le cas de la tournée du voyageur de commerce qui doit visiter plusieurs boutiques. Il connaît le temps nécessaire pour aller d'une boutique à l'autre, et cherche l'ordre dans lequel les visiter de sorte que le temps de trajet soit le plus court possible.
Il existe un algorithme simple qui consiste à comparer tous les ordres possibles. Mais il est incroyablement long dès qu'il y a plus de quelques arrêts. Avec 20 boutiques, il y a déjà 20 choix pour le premier arrêt. Pour le second, il faut en choisir un parmi 19, pour le troisième, un parmi 18, etc. Au final on compte 2 432 902 008 176 640 000 tournées possibles. Si l'ordinateur énumère un milliard de tournées par seconde, il mettra quand même plus de 70 ans pour toutes les comparer !

Quand on entend dire que les programmes d'intelligence artificielle pourraient prendre le contrôle sur le monde, on ferait bien de se souvenir qu'il existe des problèmes assez simples que les ordinateurs ne savent pas bien résoudre et pour lesquels l'intelligence humaine est souvent meilleure. »

Pour illustrer comment procéder, prenons un cas simple : un algorithme traitant plusieurs données ou valeurs comporte généralement une boucle (ou, ce qui revient ici au même, une

cascade d'appels récursifs) et se présente schématiquement comme ceci (pour n valeurs à traiter):

| *algorithme schématique* | *durée pour n valeurs* |
|---|---|
| `INITIALISATION` | $d_{init}$ |
| `while(condition) {` | $d_{cond}$ .(n+1) |
| `   TRAITEMENT` | $d_{trait}$ . n |
| `}` | |
| `TERMINAISONS` | $d_{term}$ |

Pour rester neutre par rapport au type de processeur utilisé et à ses performances, la durée est représentée par le nombre d'instructions élémentaires à exécuter. Une affectation, une addition, sont des exemples d'opérations élémentaires. Ainsi, $d_{init}$ représente le nombre d'instructions exécutées dans la phase d'initialisation, avant d'entrer dans la boucle.

$d_{cond}$ représente le nombre d'instructions élémentaires exécutées pour évaluer la condition (le « gardien de la boucle »). Remarquez que ce gardien est évalué n+1 fois : les n premières fois, il sera évalué à « vrai » pour traiter les n valeurs, et la dernière fois, il sera évalué à « faux » pour que la boucle puisse s'arrêter.

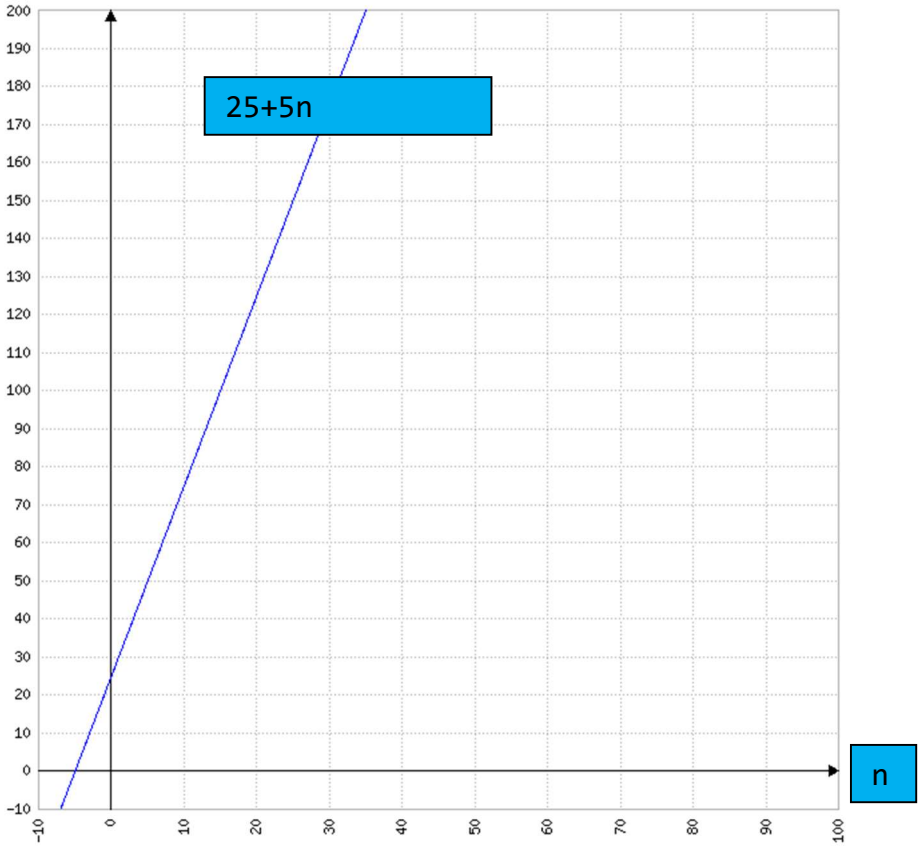
La durée totale d'exécution de la boucle ci-dessus pour le traitement de $n$ valeurs est :

$$(d_{init} + d_{cond} + d_{term}) + (d_{cond} + d_{trait}) \cdot n$$

ce qui peut s'écrire :

$$k + d.n \text{ (où } k = d_{init} + d_{cond} + d_{term} \text{ et } d = d_{cond} + d_{trait}).$$

Il s'agit donc d'une fonction linéaire en n. Si on a 10 éléments de plus à traiter, le nombre d'instructions élémentaires à exécuter augmentera de d fois 10. Si on a 0 élément à traiter, alors le temps d'exécution sera quand même égal à k. Le tracé ci-dessous illustre le cas où k vaut 25 et d vaut 5 :




Sur l'axe des abscisses (axe horizontal), le nombre de valeurs à traiter (n) augmente ici par pas de 10. Sur l'axe des ordonnées (axe vertical), on représente le nombre d'instructions à exécuter pour traiter ces données.

On voit que s'il n'y a aucun élément à traiter (n=0), on doit exécuter quand même les 25 instructions d'initialisation et de terminaison (25+5.0 = 25). Ensuite, à chaque fois que le nombre de données à traiter grandit, le nombre d'instructions élémentaires à exécuter augmente de 5 (soit la valeur de d dans k+d.n) comme l'illustre ce tableau de valeurs (où on a chaque fois 10 données de plus à traiter).

| n | 25+5n |
|---|---|
| 0 | 25 |
| 10 | 75 |
| 20 | 125 |
| 30 | 175 |
| 40 | 225 |
| 50 | 275 |
| 60 | 325 |
| 70 | 375 |
| 80 | 425 |
| 90 | 475 |
| 100 | 525 |

Lorsqu'on effectue ce genre de mesure pour un algorithme quelconque, en calculant exactement les durées impliquées (en nombre d'instructions élémentaires à exécuter), on obtient une fonction appelée **Ordre de Complexité Temporelle (OCT)** de l'algorithme. Dans le cas d'une boucle parcourant n éléments, on vient de voir que l'OCT sera une fonction linéaire, par exemple 25 + 5n comme ci-dessus.

La séquence de traitement répétitive peut contenir des variations (constructions SI ou SELON); **on prendra alors en compte la durée du cas le plus défavorable**, c'est-à-dire de la séquence de traitement la plus longue.

Remarque : Si l'on dispose d'informations statistiques sur l'état des données traitées, on peut également prendre en compte la durée moyenne.

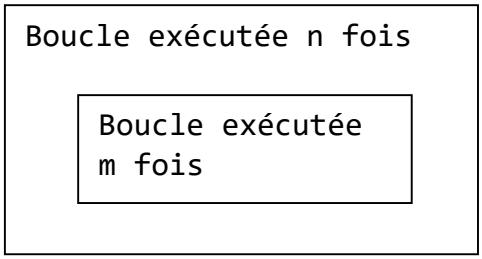
Il est généralement sans intérêt de connaitre avec exactitude chacune des durées reprises dans cette formule; il suffit de constater que, si le nombre d'éléments traités augmente de *n* unités, la durée totale d'exécution augmente ici d'une quantité *d*n* strictement proportionnelle. Négligeant de prendre en compte les constantes k et d, on dit que la **complexité temporelle théorique** de cet algorithme est *« de l'ordre de* n », ce qui se note **O(n)** et se lit *« en* ô de n ». **Il s'agit d'une approximation**, non pas de sa durée d'exécution, mais **de la fonction de *croissance* de cette durée, c'est-à-dire de la manière dont la durée d'exécution augmente lorsque n augmente**. Rappelons que la durée est mesurée en termes de nombre d'instructions élémentaires à exécuter, et que n représente le nombre de données en entrée.

La complexité temporelle théorique d'un algorithme est *de l'ordre* d'une certaine fonction de *n*, que l'on obtient à partir de l'ordre de complexité temporelle de l'algorithme, mais en faisant abstraction des temps unitaires (constants) et en ne considérant que le nombre d'itérations.

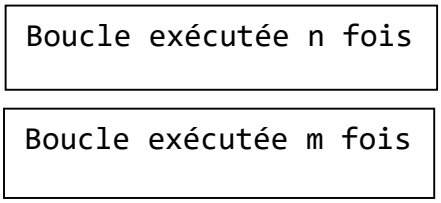
On rencontre fréquemment des mesures comme les suivantes - de la pire à la meilleure : $O(2^n)$, $O(n^2)$, O(n log n ), $O(\log_2 n)$, O(1). La dernière indique une durée d'exécution constante, c'est-à-dire complètement indépendante du nombre de données à traiter.

Nous avons pris comme exemple simple le cas d'une seule boucle parcourant et traitant n éléments. Quelle serait la **complexité temporelle théorique (CTT) d'un algorithme renfermant plusieurs boucles** ? Si une première boucle parcourt n éléments, et qu'une seconde boucle en traite m, on obtient ces complexités temporelles théoriques:

- boucles emboitées (la seconde boucle fait partie du code à l'intérieur de la première): CTT = O(n.m)




- boucles successives (la première boucle est exécutée entièrement avant d'exécuter la seconde): CTT = O(n+m).




Pour vous en convaincre, n'hésitez pas à refaire l'exercice qui nous a permis de trouver l'OCT d'une boucle simple.

**Lorsqu'un algorithme procède par divisions successives de l'ensemble de données à traiter**, comme dans le cas de la recherche dichotomique, **il est en O(log n)** : $\log_2 n$ si l'ensemble de données est divisé par 2 à chaque étape[13], par exemple.

[13] En effet, le logarithme en base 2 d'un nombre équivaut au nombre de fois que ce nombre est divisible par deux.

### 7.2.1 Exemple

Soient 3 algorithmes dont les ordres de complexité temporelle sont les suivants:

$OCT_1 = 2n + 100$ soit un algorithme de complexité temporelle théorique (CTT) en O(n)

OCT2 = 4 n + 75 soit un algorithme de complexité temporelle théorique (CTT) en O(n)

$OCT_3 = 3\ n^2 + 2n + 5$ soit un algorithme de complexité temporelle théorique (CTT) en $O(n^2)$

Les 2 premiers algorithmes ont la même complexité temporelle théorique, même si leurs OCT (ordres de complexité temporelle) diffèrent. En revanche le troisième est quadratique, c'est-à-dire en $O(n^2)$

Pour de petites valeurs de n (nombre de données à traiter faible), ces 3 algorithmes nécessitent l'exécution d'un nombre d'instructions assez équivalent : pour 3 données à traiter, 106 instructions, 87 instructions et 38 instructions. C'est même l'algorithme quadratique ($OCT_3$) qui semble le meilleur.

Cependant, dès que le nombre de données à traiter dépasse un certain seuil, ce sont clairement les algorithmes linéaires qui sont à privilégier, si on veut minimiser le nombres d'insctructions à exécuter : pour n=1.000, on atteint déjà 2.100, 4.075 et 3.002.005 instructions, respectivement !

### 7.2.2 On retiendra :

- Ce qui nous intéresse en général est le comportement des algorithmes pour des tailles croissantes et très grandes des données en entrée.
- Pour mesurer l'efficacité inhérente à l'algorithme étudié, la durée est mesurée grâce au nombre d'instructions élémentaires à exécuter.
- L'OCT (**Ordre de Complexité Temporelle**) est une fonction exprimant la manière dont la durée d'exécution d'un algorithme augmente lorsque le nombre ou la taille des données en entrée augmente.
- La **CTT (Complexité Temporelle Théorique)** est une approximation de l'OCT.
- Pour exprimer les CTT on utilise la notation $\mathcal{O}$("grand O"):

  Soit g(n) l'OCT de notre algorithme.
  On a : $g \in O(f) \Leftrightarrow \exists n_0 > 0 \; \exists c > 0 : g(n) \leq c.f(n) \; \forall \; n > n\_0$
  ce que l'on peut traduire comme ceci : g est dit d'ordre f si et seulement si toutes les valeurs de g sont bornées par les valeurs de f, à partir d'un n suffisamment grand. On s'intéresse donc au comportement asymptotique des fonctions de complexité : pour un nombre de données n proche de l'infini. Le but est de ne pas devoir calculer l'OCT exact de l'algorithme, mais de pouvoir évaluer son ordre de croissance en trouvant une fonction f qui le borne.
- Doit-on considérer le nombre d'opérations dans le pire des cas *(worst-case analysi*s), le meilleur des cas *(best-case analysi*s), le cas moyen *(average-case analysi*s) ?
  Notre choix se portera sur l'**analyse dans le pire des cas**, plus éventuellement le cas moyen quand ce sera possible.
- Les constantes calculées dans l'OCT ont leur importance pour **comparer des algorithmes de complexité équivalente**, et pour **comparer des algorithmes lorsque la taille des données est "petite"**.

### 7.2.3 Quelques chiffres

Pour une taille du problème égale à 1.000 (c'est-à-dire un nombre d'éléments à traiter n = 1.000) et un temps d'exécution de chaque instruction de 1 microseconde (µs) :

| Fonction | Taux de croissance | Exemple d'OCT | Temps |
|---|---|---|---|
| O(1) | Constant | 10 | 10 µs |
| O(log n) | Logarithmique | 20 log n + 3 | 63 µs |
| O(n) | Linéaire | 20 n + 50 | 20 ms 50 µs |
| $O(n^2)$ | Quadratique | $20\ n^2 + 30\ n + 50$ | 20s 30ms 50µs |
| $O(n^3)$ | Cubique | $5n^3 + 2n^2 + 2n + 60$ | 1h 23min 22s 2ms 60µs |
| $O(2^n)$ | Exponentiel | $2^n$ | $10^{86}$ siècles |
| O(n!) | Factoriel | n! | $10^{2552}$ siècles |

## 7.3 OPTIMISATION DE L'OCCUPATION D'ESPACE EN MÉMOIRE CENTRALE

De la même manière qu'on le fait pour la complexité temporelle, on peut calculer **l'ordre de complexité spatiale (OCS)** et la **complexité spatiale théorique (CST)** d'un algorithme, qui évaluent la manière dont l'espace mémoire nécessaire augmente quand la taille des données à traiter (et donc aussi à mémoriser) croît. La CST est le plus souvent linéaire (on doit stocker toutes les données en entrée, plus il y en a, plus l'espace mémoire requis augmente).

## 7.4 OPTIMISER LE TEMPS D'EXÉCUTION OU L'ESPACE REQUIS ?

**Généralement, on ne saura pas trouver un algorithme qui minimise à la fois les complexités temporelles et spatiales !** Il faudra alors réaliser un compromis entre le temps d'exécution et la taille mémoire nécessaire. Vu la diminution constante des coûts des mémoires des ordinateurs personnels, la tendance a longtemps visé la minimisation de la complexité temporelle. Cependant, avec l'avènement des smartphones et tablettes, la montée de la domotique et l'arrivée de l'IoT (*Internet of Things*) et autres équipements « intelligents », le programmeur d'applications dites « mobiles » ou « embarquées » doit tenir compte des capacités mémoires réduites de ce type d'équipement et il redevient crucial de veiller à réduire la complexité spatiale.

# 8 TYPES DE PROBLÈMES

La notion de complexité temporelle théorique permet de définir deux grandes classes de problèmes :

- Les **problèmes « faciles »**, c'est à dire pour lesquels il existe des algorithmes de complexité (dans le pire des cas) bornée par un polynôme en la taille des entrées, ou encore pour lesquels il existe des algorithmes de **complexité polynômiale**, ces algorithmes étant alors appelés **des algorithmes efficaces.**
- Les **problèmes « difficiles »**, c'est à dire pour lesquels la complexité dans le pire des cas n'est pas polynômiale. Ce sont donc des **problèmes non polynômiaux**, et les algorithmes éventuels permettant malgré tout de les résoudre sont qualifiés d'**inefficaces.**

Cette distinction est en fait d'une grande importance pratique !

- Les problèmes polynomiaux correspondent à des problèmes pour lesquels on a de bonnes chances de trouver une méthode de résolution, (soit dès aujourd'hui, soit dans un proche avenir) avec un ordinateur, pour toutes les tailles raisonnables des entrées.
- Les problèmes non polynomiaux correspondent à des problèmes que l'on a de bonnes chances de ne jamais pouvoir résoudre avec un ordinateur, pour toutes les tailles raisonnables des entrées! Il faudra donc envisager des solutions approchées ou euristiques[14]…

[14] On qualifie d'euristique une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème difficile

Ainsi, considérons un problème nécessitant de parcourir toutes les permutations possibles de n objets. Ce problème a une complexité temporelle factorielle : O(n!). Pour le résoudre avec un nombre d'objets n égal à 25, il faudrait près de 5 millions de siècles à une machine effectuant 1 milliard d'instructions par secondes !

# TABLE DES MATIÈRES DE LA PARTIE 1


**1 AVANT-PROPOS À L'ÉDITION 2024 ..... 1**

**2 L'UE « PROGRAMMATION INTERMÉDIAIRE » ..... 1**

2.1 THÈMES DU COURS D'ALGORITHMIQUE ..... 2

2.2 MODALITÉS D'ÉVALUATION ..... 2

2.3 SUPPORTS DE COURS ..... 2

2.4 RÉFÉRENCES PRINCIPALES ..... 3

**3 INTRODUCTION – NOTION D'ALGORITHME ..... 4**

3.1 RAPPELS DES ÉTAPES DE « FABRICATION » D'UN PROGRAMME ..... 4

3.2 DÉFINITIONS ..... 6

*3.2.1 Définitions informelles ..... 6*

*3.2.2 Définitions formelles ..... 7*

3.3 EXEMPLE DU LABYRINTHE DE PLEDGE ..... 8

**4 ÉLABORATION DE SOLUTIONS ALGORITHMIQUES: APPROCHES LES PLUS FRÉQUENTES ..... 9**

**5 TYPES D'ALGORITHMES ET PRINCIPAUX DOMAINES D'UTILISATION ..... 11**

**6 QUALITÉS OU CARACTÉRISTIQUES D'UN ALGORITHME ..... 15**

6.1 DOMAINE ..... 15

6.2 EXACTITUDE ..... 15

*6.2.1 Les tests ..... 16*

*6.2.2 Utilisation d'une référence ..... 18*

6.3 EFFICACITÉ ..... 21

*6.3.1 Nombre d'opérations ..... 21*

*6.3.2 Espace mémoire ..... 21*

*6.3.3 Transferts ..... 21*

6.4 EXEMPLE : RECHERCHE PAR DICHOTOMIE ..... 22

*6.4.1 Jeu de tests à utiliser : ..... 22*

*6.4.2 Caractéristiques/Qualités ..... 23*

**7 NOTION DE COMPLEXITÉ ..... 24**

7.1 INTRODUCTION ..... 24

7.2 CALCUL DE LA COMPLEXITÉ TEMPORELLE THÉORIQUE D'UN ALGORITHME ..... 29

*7.2.1 Exemple ..... 34*

*7.2.2 On retiendra : ..... 35*

*7.2.3 Quelques chiffres ..... 36*

7.3 OPTIMISATION DE L'OCCUPATION D'ESPACE EN MÉMOIRE CENTRALE ..... 36

7.4 OPTIMISER LE TEMPS D'EXÉCUTION OU L'ESPACE REQUIS ? ..... 37

**8 TYPES DE PROBLÈMES ..... 37**