Définition L'*algorithmique* est l'étude des algorithmes. C'est l'art d'analyser des algorithmes afin d'en mesurer l'efficacité et la fiabilité (l'algorithme répond-t-il vraiment au problème posé ?).

1 Pourquoi étudier l'algorithmique ?

Etudier l'algorithmique permet de mieux comprendre les mécanismes utilisés par un ordinateur et de répondre à des questions du type :

- L'algorithme proposé se termine-t-il ?
- Le résultat donné par l'ordinateur est-il conforme à la réponse attendue ?
- Combien de temps vais-je devoir attendre la fin de l'exécution de l'algorithme ?
- Mon ordinateur possède-t-il assez de mémoire pour une bonne exécution de l'algorithme ?
- Est-il possible d'écrire un programme plus rapide ou moins gourmand en mémoire ?

En pratique l'algorithmique va nous permettre de prouver qu'un algorithme est correct (se termine et produit une réponse conforme à ce qui est attendu), de comparer différents algorithmes avec les mêmes spécifications (complexité) et nous fournir des exemples d'algorithmes et de modélisation de problèmes.

2 Qu'est-ce qu'un algorithme ?

Un algorithme est une suite finie d'instructions permettant de résoudre un problème. Il est structuré en trois parties :

1. l'entrée des données ;
2. le traitement des données ;
3. la sortie des données.

Les algorithmes existaient bien avant l'informatique. Par exemple : l'algorithme d'Euclide a plus de 2000 ans. Le nom algorithme vient du savant arabe Al-Kwarizmi (IXème siècle) à qui l'on doit notamment des méthodes pour le calcul effectif des racines d'une équation du second degré.

En pratique on rédige des algorithmes en « pseudo-code » car l'objectif est de donner les principales étapes d'un programme en vue de résoudre un problème mais indépendamment des particularités du langage utilisé. Il en donne la structure logique (comme le plan pour une dissertation).

Donald Knuth, dans un ouvrage nommé « The Art of Computer Programming » donne en 1968 cinq caractéristiques fondamentales aux algorithmes :

- ⋆ un algorithme doit toujours se terminer après un nombre fini d'étapes,
- ⋆ chaque étape de l'algorithme doit être clairement spécifiée (les instructions doivent être claires et sans ambiguïté),
- ⋆ un algorithme peut avoir des données en entrée (quelquefois aucune),
- ⋆ un algorithme produit une ou plusieurs sorties en relation avec les éventuelles entrées,
- ⋆ les instructions sont suffisamment simples pour pouvoir être exécutées par une personne en temps fini et de manière exacte avec un papier et un crayon.

Remarque 2.1 Pour simplifier la compréhension d'un algorithme, les programmeurs peuvent utiliser certaines conventions de nommage pour les variables. Par exemple, pour une variable comptant des itérations on pourra choisir de nommer sa variable `cpt`, pour une variable réalisant une somme de plusieurs valeurs on pourra choisir `acc` (pour accumulateur), pour une variable temporaire : `temp`, ...

3 Écrire un algorithme : spécification et tests

3.1 Spécification

Avant d'écrire un algorithme il faut bien définir ce que l'on veut faire et à partir de quoi, il s'agit de donner une *spécification* au problème. Pour cela on doit :

- donner un nom explicite à l'algorithme,
- décrire les conditions d'utilisation (ou la *précondition)* : nature des données d'entrée (par exemple : un entier plus grand que 0),
- décrire le résultat attendu (ou la *postcondition*) : quelles sont les données renvoyées et à quoi correspondent-elles.

Définition 3.1 (Précondition) Une *précondition* est une condition à vérifier avant le début d'un calcul ou l'appel d'une fonction afin de pouvoir en garantir le résultat. Par exemple : avant une division, s'assurer que le dénominateur est non nul.

Définition 3.2 (♦ Postcondition) Une *postcondition* est une condition à vérifier à la fin d'un calcul ou d'une fonction. Sa vérification implique que le calcul est correct ; et réciproquement, une postcondition non satisfaite correspond à une erreur lors du calcul, généralement due à une erreur de programmation. Exemple : à la fin d'une fonction retournant la valeur absolue d'un nombre, vérifier que le résultat est supérieur ou égal à 0.

Exercice 3.3 On donne la fonction suivante, codée en `Python`. La documentation fournie (ou *docstring*) donne une spécification de la fonction.

```
def médiane(num1,num2,num3):
    """Les paramètres sont de type numérique (int ou float).
    La fonction renvoie la valeur médiane (ni la plus grande, ni la plus petite).
    Ex : médiane(1,2,3) renvoie 2."""
    return min(max(num1,num2),max(num2,num3),max(num1,num3))
```

Précisez quelle est la précondition et quel est le résultat attendu.

Remarque 3.4 Le programmeur qui prend le temps de bien documenter ses fonctions réalise en fait l'étape de spécification.

Exemple 3.5 Problème : calculer la somme des n premiers entiers, n étant un entier donné.

Spécification : l'algorithme se nomme `SommePremiersEntiers` et prend en entrée un entier strictement positif (précondition) n. Il renvoie en sortie un entier S contenant la somme des entiers allant de 1 à n (postcondition).

L'algorithme ci-contre répond à cette spécification :

```
fonction SommePremiersEntiers(n)
  S ← 0
  k ← 1
  tant que k ⩽ n faire
     S ← S + k
     k ← k+1
  fin tant que
  renvoyer S
fin fonction
```

◅ **Méthode** Dans un premiers temps il faut identifier quelles sont les données nécessaires à la résolution du problème. C'est l'*entrée des données*. Celles-ci pourront être de différentes formes : numériques, sous forme de texte (chaînes de caractères), logiques (deux valeurs possibles : vraie ou faux), graphiques… Dans cette première partie on trouve les instructions de saisie (récupération de données entrées par l'utilisateur, position d'un clic de souris, lecture d'un fichier de données…). On doit définir ici des *préconditions*

On peut ensuite décider sous quelle forme le résultat de l'algorithme sera donné. C'est la *sortie des données*. Elles pourront être affichées à l'écran, stockées dans des variables réutilisées par la suite, traduites dans un fichier…

Le gros du travail reste alors de décrire précisément la suite d'instructions permettant de transformer les données d'entrée en données de sortie. C'est l'étape du *traitement des données*. Les instructions utilisées sont de trois types :

- les affectations ou transfert de données (lecture, stockage, copie),
- les instructions arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière) et
- les instructions de contrôle (structure alternative (si … alors… sinon), structures répétitives (les boucles tant que et pour) et appels de fonctions).

Exercice 3.6 Charles souhaite écrire un algorithme pour calculer sa moyenne en NSI, composée de trois notes. La première note est coefficient 1, la deuxième coefficient 2 et la troisième coefficient $0{,}5$.
Donner la spécification de cet algorithme et écrire un algorithme répondant au problème de Charles.

3.2 Tests

Même si on a bien spécifié sa fonction, il est encore possible de faire une erreur en écrivant son algorithme. Pour détecter ces éventuelles erreurs, on peut **tester** son algorithme sur quelques cas concrets et vérifier qu'il produit effectivement le résultat attendu. On procèdera alors à l'étape de débogage (moment où l'on essaie de corriger les erreurs de l'algorithme).

Exemple 3.7 On reprend la fonction `médiane` présentée dans l'exercice 3.3. L'instruction `médiane(1,2,3)` est un test : il s'agit de vérifier qu'elle renvoie effectivement le nombre 2.

Un test permet de vérifier que l'algorithme fonctionne sur une donnée précise. Pour programmer efficacement il faut concevoir des *jeux de tests* permettant de vérifier que l'algorithme renvoie, dans des cas particuliers bien choisis, ce que l'on attend de lui. Il est cependant impossible d'écrire un ensemble de tests permettant d'exclure toutes les erreurs possibles, mais on peut cependant essayer d'en construire un en respectant déjà les règles suivantes (on pourra en ajouter au cas par cas) :

- si la spécification de l'algorithme mentionne plusieurs cas possibles, les tester tous ;
- si l'algorithme doit renvoyer une valeur booléenne, construire des tests permettant d'obtenir les deux valeurs de vérité ;
- si l'algorithme s'applique à un tableau (une liste en `Python`, voir chapitre 2), effectuer un test avec un tableau vide ;
- si l'algorithme s'applique à un nombre, effectuer des tests avec des valeurs positives, négatives et avec zéro (ou d'autres valeurs remarquables pour le problème comme les valeurs limites de l'intervalle de spécification).

Exercice 3.8 Proposer deux tests pour la fonction `SommePremiersEntiers` de l'exemple 3.5.

4 Preuve d'algorithmes

Lorsque l'on écrit un algorithme, il faut pouvoir s'assurer qu'il s'exécute correctement dans toutes les situations. **Un simple jeu de tests ne suffit pas assurer que le programme se termine et produit le résultat attendu à tous les coups**. Dans certaines applications industrielles il est même vital de démontrer qu'un programme est correct (ex : programmes contrôlant la conduite des lignes de métro automatiques).

Par analogie avec les mathématiques, on peut assimiler respectivement l'entrée et la sortie d'un algorithme à l'hypothèse (les conditions d'application ou précondition) et la conclusion d'un théorème (la postcondition). C'est la *correspondance de Curry-Howard*, appelée également correspondance preuve/programme ou correspondance formule/type. Elle établit des relations entre les démonstrations formelles d'un système logique et les programmes d'un modèle de calcul. Les premiers exemples de correspondance de Curry-Howard remontent à 1958.

En pratique, une preuve d'algorithme se décompose en deux parties :

- la *terminaison* : on montre que l'algorithme se termine.
- la *correction partielle* : il s'agit de savoir si l'algorithme produit bien la réponse attendue.

Quand on a démontré la terminaison et la correction partielle de l'algorithme on obtient ainsi la *correction totale*. Autrement dit, on peut affirmer que quelles que soient les données fournies, l'algorithme s'arrête **et** donne une réponse correcte.
Nous verrons plus tard des outils pour prouver la terminaison et la correction de certains algorithmes (voir chapitre 3 sur les tris).

5 Comparer des algorithmes : la complexité

On utilise des critères de comparaison indépendants du langage de programmation choisi et des performances de la machine : le nombre d'opérations élémentaires effectuées et la place mémoire utilisée.

La complexité d'un algorithme se calcule en tenant compte de la taille des données d'entrée. En effet, trier une liste de 3 éléments est naturellement plus court que trier une liste de 1000 éléments. La notion de taille des données est intrinsèque au problème étudié.

Le temps d'exécution pour une entrée particulière est le nombre d'opérations élémentaires, ou « étapes » effectuées. On considère que chaque instruction a un temps d'exécution constant. Pour calculer la complexité d'un algorithme, il s'agit alors d'additionner le temps d'exécution des instructions de l'algorithme en fonction de la taille des données. On se contente d'avoir un ordre de grandeur par excès de cette complexité en considérant généralement le cas le plus défavorable.

Le but principal d'un calcul de complexité est de pouvoir comparer l'efficacité entre différents algorithmes répondant à un même problème. Plus précisément, il permet de répondre au problème fondamental suivant :

« Sur toute machine, et quel que soit le langage utilisé,
l'algorithme α est-il meilleur que l'algorithme β pour des données de grande taille ? »

Définition 5.1 La complexité d'un algorithme est une *mesure intrinsèque* à l'algorithme qui est *indépendante* de toute implémentation. Elle est calculée en fonction d'un *paramètre représentatif des entrées* (la taille) et à l'aide d'une *mesure élémentaire* (nombre de comparaisons, nombre d'opérations arithmétiques, nombre d'affectations, etc.).

Le choix de la mesure élémentaire ne donne pas la même information sur la complexité d'un algorithme. Par exemple, l'utilisation du nombre de comparaisons ou d'opérations arithmétiques donne une information sur la vitesse d'exécution de l'algorithme, alors que l'utilisation du nombre d'affectations donne une information sur la place mémoire utilisée par l'algorithme. Ainsi, on distingue deux types de complexité :

- *la complexité en temps*, c'est-à-dire comment évolue le temps d'exécution d'un algorithme lorsque la taille des données augmente ?
- *la complexité en espace*, c'est-à-dire comment évolue l'espace mémoire requis lors de l'exécution d'un algorithme lorsque la taille des données augmente ?

Dans la majorité des cas, quand on parle de complexité d'un algorithme, on s'intéresse plutôt à la complexité en temps. En effet, compte-tenu des mémoires RAM des ordinateurs actuels, la place mémoire n'est généralement pas un problème. Toutefois, pour certaines applications, comme les technologies embarquées, l'étude de la complexité en espace s'avère également très importante.

Exemple 5.2 (Somme des premiers entiers) La fonction `SommePremiersEntiers` est basée sur une boucle : en prenant comme mesure les opérations arithmétiques élémentaires, son exécution nécessite $2n$ additions (on ajoute 1 à la variable k et k à S dans le corps de la boucle, ce que l'on répète n fois). Cependant, il existe une formule mathématique permettant de calculer la sommes des entiers de 1 à n :

$$1 + 2 + ... + n = \frac{n(n+1)}{2}.$$

Un algorithme basé sur cet formule n'utilise alors que 3 opérations (une addition, une multiplication et une division). : il est donc clairement meilleur que `SommePremiersEntiers` en terme de nombre d'opérations arithmétiques.