Modulo.
Une introduction à l'informatique

Groupe de travail DGEP, EPFL, HEP-VD, UNIL

13 septembre 2024

CC BY NC

Table des matières

1 Algorithmique I **1**
1.0 Introduction . . . 1
1.0.1 Quoi ? . . . 1
1.0.2 Pourquoi ? . . . 1
1.0.3 Comment ? . . . 2
1.0.4 Objectifs d'apprentissage . . . 2
1.1 Les algorithmes . . . 3
1.1.1 Résolution d'un problème par étapes . . . 3
1.1.2 Les ingrédients d'un algorithme . . . 7
1.1.3 Exercices . . . 10
1.2 Trie, cherche et trouve . . . 13
1.2.1 Algorithmes de tri . . . 13
1.2.2 Tri par insertion . . . 14
1.2.3 Tri par sélection . . . 14
1.2.4 Tri à bulles . . . 15
1.2.5 Comparaison d'algorithmes . . . 18
1.2.6 Exercices . . . 19
1.3 Des algorithmes aux programmes . . . 21
1.3.1 Exercices . . . 24
1.4 Conclusion . . . 27
1.4.1 Les algorithmes et nous . . . 27
1.4.2 Focus sur l'automatisation . . . 27

CHAPITRE **1**

Algorithmique I

1.0 Introduction

1.0.1 Quoi ?

Nous avons tous entendu parler des algorithmes dans les médias. Normal, c'est le mot à la mode et que tout le monde utilise sans vraiment le comprendre. Ils sont partout, ils font toutes sortes de choses, même nous manipuler. Pourquoi en parle-t-on de la même manière que des extraterrestres ? Dans ce cours, nous allons tenter de revenir sur terre, parce que les algorithmes ce n'est pas si compliqué que ça. On apprendra à les définir, à les faire marcher et surtout à reconnaître la différence entre un programme et un algorithme, ainsi qu'entre un « bon » et un « mauvais » algorithme.

1.0.2 Pourquoi ?

Les algorithmes existent depuis des millénaires. On doit le nom d'algorithme à Al-Khwârizmî, mathématicien perse né en l'an 780 dont les ouvrages ont contribué à la popularisation des chiffres arabes en Europe, ainsi que la classification de plusieurs algorithmes connus à ce moment. D'ailleurs l'algorithme le plus connu, l'algorithme d'Euclide, date environ de l'an 300 av J.-C. et permet de calculer le plus grand diviseur commun de deux nombres. Si Euclide a bien laissé des traces écrites de cet algorithme, il est vraisemblable qu'il ait puisé cette connaissance auprès de disciples de Pythagore lui-même.

Les algorithmes sont devenus très populaires aujourd'hui grâce à la machine qui a permis de les automatiser. Que ce soit dans votre smartphone, sur un ordinateur ou dans un système embarqué, ils permettent de résoudre une quantité de problèmes, facilement et avec une rapidité impressionnante.

1.0.3 Comment ?

Dans un premier temps nous allons nous intéresser à la notion même d'algorithme : qu'est-ce qui caractérise un algorithme et comment le faire exécuter par une machine ? Nous allons voir que pour un problème donné il existe de nombreuses solutions, mais que toutes ces solutions ne sont pas de *bonnes* solutions, selon le contexte dans lequel on tente de résoudre le problème.

1.0.4 Objectifs d'apprentissage

À la fin de ce chapitre, vous saurez ce qu'est un algorithme et vous serez capable de transcrire des algorithmes en programmes. Vous saurez résoudre des problèmes, en décomposant leur solution en étapes à suivre. Vous verrez également que pour un même problème, on peut avoir plusieurs solutions avec des propriétés, avantages et désavantages différents.

— Se familiariser avec la notion d'algorithme.
— Savoir résoudre des problèmes, en décomposant leur solution en étapes à suivre.
— Savoir que pour un même problème, on peut avoir plusieurs solutions avec différents propriétés, avantages et désavantages.
— Être capable de transcrire un algorithme dans un programme.

Bienvenue dans le monde fascinant des algorithmes.

1.1 Les algorithmes

La première question que l'on va se poser est la suivante : qu'est-ce qu'un *algorithme* ? Est-ce la même chose qu'un programme informatique, ou s'agit-il d'autre chose ?

Un algorithme est en quelque sorte « une recette » que l'on peut suivre pour **résoudre un problème**. De nos jours, il existe énormément de problèmes que les algorithmes nous permettent de résoudre. Il existe des algorithmes pour calculer le trajet le plus rapide entre deux lieux ; d'autres algorithmes ont été imaginés pour détecter les visages dans nos photos ; une demande sur un moteur de recherche est analysée par de nombreux algorithmes afin de nous aider à mieux définir ce que l'on cherche ou afin de nous proposer des contenus publicitaires adaptés.

Ce n'est pas l'algorithme qui est exécuté sur une machine pour nous donner une solution concrète pour tous ces problèmes. *L'algorithme n'est donc pas un programme*. L'algorithme décrit plutôt un « mode d'emploi », qui permet de réfléchir à un problème de manière générale et ensuite de créer un *programme*. C'est le programme qui sera exécuté par un système informatique pour concrètement résoudre le problème. En d'autres mots, l'algorithme décrit l'idée humaine derrière la solution d'un problème, alors que c'est le programme qui permet à une machine de trouver une solution numérique dans des cas précis.

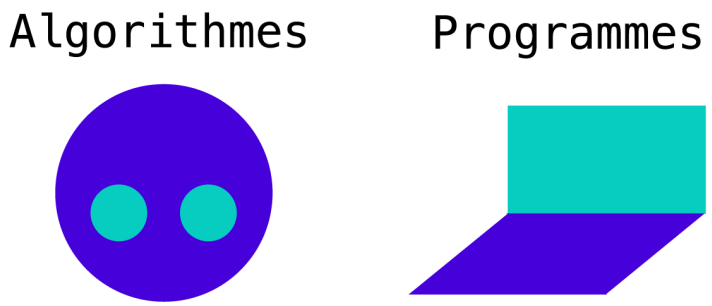


Fig. 1.1 – Différence entre un algorithme et un programme. Un algorithme doit être compréhensible par un humain, alors qu'un programme est écrit de façon à ce qu'il soit compréhensible par une machine.

1.1.1 Résolution d'un problème par étapes

Un mode d'emploi, ou une recette, décrit les **étapes** à suivre pour arriver à une solution. Dans le cas d'une recette de cuisine, la préparation des ingrédients, leur cuisson et leur présentation sont différentes étapes que l'on peut suivre pour réaliser un plat. Prenons un cas précis : *faire une omelette*. Pour chaque étape de la préparation de l'omelette, il faut prévoir une marche à suivre suffisamment détaillée, afin que la personne qui suit la recette arrive au résultat souhaité. Dans le cas de l'omelette, les opérations pourraient être (voir la figure suivante) :

1. Casser les œufs dans un bol.
2. Mélanger les œufs jusqu'à obtenir un mélange homogène.
3. Cuire le mélange d'œufs dans une poêle à température moyenne.
4. Lorsque cuite, glisser l'omelette dans une assiette.

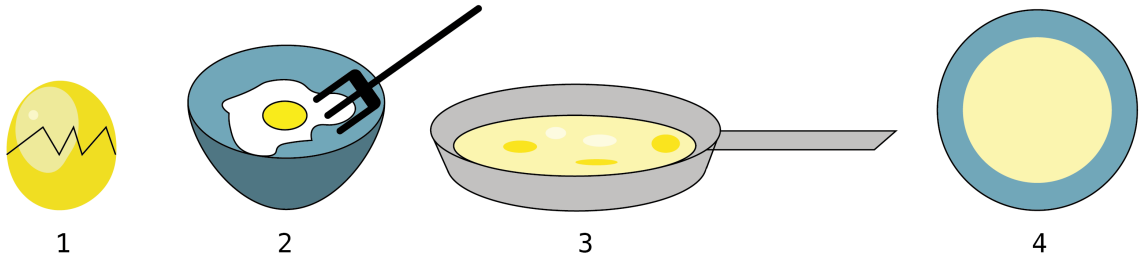


Fig. 1.2 – Un algorithme est un peu comme une recette de cuisine. Cet exemple illustre les opérations à suivre pour la réalisation d'une omelette.

Dans le cas de la recette d'une omelette, nous avons décomposé la marche à suivre en étapes à réaliser dans un certain ordre. Il en est de même pour un algorithme. Pour résoudre un problème, il faut d'abord **décomposer le problème en sous-problèmes** que l'on *sait résoudre*. La solution de chaque sous-problème donne lieu à une étape qu'il faudra exécuter pour arriver à un résultat. Voici les sous-problèmes que certaines étapes ci-dessus permettent de résoudre. Afin d'extraire le contenu comestible de l'œuf, il faut casser les œufs. Pour que l'omelette ait une jolie couleur uniforme, il faut mélanger le jaune et le blanc d'œuf. Cette étape ne serait pas du tout pertinente si le problème que l'on essaie de résoudre est la préparation d'un œuf au plat. *L'algorithme décrit donc toutes les opérations qu'il faut effectuer pour arriver à ce résultat*. Nous allons ainsi définir l'algorithme comme **une suite d'opérations qui permettent de résoudre un problème**.

Le langage utilisé pour écrire un programme doit être extrêmement précis, sans quoi une machine ne pourrait pas le comprendre. Nous avons vu qu'un algorithme n'a pas besoin d'être compris par une machine, mais seulement par les humains. Ainsi, le langage que l'on va utiliser pour exprimer un algorithme sera plus libre que celui utilisé pour coder un programme. Ce langage **peut varier d'une personne à l'autre** et se rapproche dans notre cas de la langue française, comme le montre cet exemple :

```
Liste Nombres           # la variable Nombres contient une liste de nombres
n ← longueur(Nombres)   # la variable n contient le nombre d'éléments dans Nombres
i ← 1                   # la variable i contient 1 pour commencer
Résultat ← 0            # la variable Résultat contient 0 pour commencer

Répéter Pour i ← 1 à n  # i prend la valeur de 1, puis 2, puis 3, jusqu'à n
    Résultat ← Résultat + Nombres[i]
                        # Résultat est incrémenté de l'i-ème élément de Nombres
Fin Répéter             # quand i vaut n l'algorithme se termine

Retourner Résultat      # la solution se trouve dans Résultat
```

Dans cet algorithme on mentionne le terme *variable*. Pour rappel, les variables associent un nom (ou un identifiant) à une valeur. Par exemple, ci-dessus on va utiliser une variable que l'on va appeler `i` et qui va stocker pour commencer la valeur `1`. Le terme variable prend tout son sens dans l'opération *Répéter*, lorsque `i` contient à tour de rôle des valeurs allant de `1` à `n`, car à ce moment-là la valeur stockée dans `i` **varie**.

Pour mieux vous représenter une variable, imaginez un grand meuble avec des tiroirs (voir la figure suivante). Les variables sont les tiroirs. Chaque tiroir comporte une étiquette, c'est le nom de la variable, et c'est grâce à ce nom que l'on sait quel tiroir ouvrir et quelle valeur utiliser. Le tiroir est petit et ne peut contenir qu'une valeur. Donc `i` peut valoir `1` ou `2`, mais pas `1` et `2` à la fois. Par contre *i* pourrait contenir une liste qui contient les valeurs `[1, 2]`. Cependant, *i* ne peut contenir qu'une seule liste à la fois et pas par exemple deux listes `[1, 2]` et `[3, 4]`.

Fig. 1.3 – Une variable est un tiroir avec une étiquette. Cela peut être utile de voir la variable comme un tiroir qui permet de stocker une valeur (contenu du tiroir) sous un nom (étiquette du tiroir). Attention, le tiroir est petit et ne peut contenir qu'une chose (valeur) à la fois. Deux tiroirs différents ne peuvent porter la même étiquette.

Lorsque l'on dit que `i ← 1`, ou que `i = 1` en Python, cela veut tout simplement dire que la variable `i` vaut maintenant `1`. Cette opération signifie que l'on va prendre le tiroir avec étiquette `i` dans la commode (s'il n'existe pas encore on va noter *i* sur l'étiquette d'un tiroir disponible) et on va mettre la valeur `1` dedans. Ce qui se trouvait dans le tiroir avant la valeur `1` ne s'y trouve plus, on dit que *la valeur précédente est écrasée*. A chaque fois que nous utilisons `i` dans l'algorithme ou dans le code, nous faisons référence à la valeur stockée dans le tiroir.

Exercice 1 – Algorithme mystère

Lisez bien l'algorithme présenté ci-dessus. Quel problème cet algorithme permet-il de résoudre ? Il est plus facile de répondre à cette question, si l'on imagine que la liste *Nombres* contient par exemple les nombres `4`, `5` et `6` (correspond à `[4, 5, 6]` en Python).

Solution 1 – Algorithme mystère

Pour répondre à cette problématique il faut se poser la question suivante : que contient la variable `Résultat` à la fin de l'algorithme ?

Pour commencer, la variable `Résultat` vaut 0. En effet, l'opération `Résultat ← 0` *initialise* `Résultat` à 0. Initialiser une variable veut dire qu'on lui assigne une toute première valeur (une valeur initiale). Dans le cas de `Nombres` qui contiendrait les nombres `4`, `5` et `6`, après le premier passage dans la boucle `Répéter`, `Résultat` vaut `4`. En effet, pour commencer `i` vaut `1` et donc `Nombres[i]` vaut `Nombres[1]`. `Nombres[1]` correspond au premier élément de la liste *Nombres* et vaut `4`. L'opération `Résultat ← Résultat + Nombres[i]`, additionne alors 0 et `4` (`Résultat + Nombres[i]`) et l'opérateur ← stocke cette valeur 4 dans la variable `Résultat`.

Au deuxième passage dans la boucle, `i` vaut `2`. On additionne à nouveau `Résultat`, qui maintenant vaut `4`, au 2ème élément de *Nombres*, qui vaut 5. Après ce deuxième passage de la boucle, `Résultat` contient `9` (`4 + 5`). Finalement, au troisième et dernier passage de la boucle, on additionne cette nouvelle valeur de `Résultat` (ou `9`) avec le 3ème élément de Nombres, qui vaut `6`. Il s'agit du dernier passage de la boucle, parce que lors de ce passage de la boucle `i` atteint la longueur de la liste `Nombres` (ou `3`). À la fin de l'algorithme, `Résultat` vaut ainsi `15`.

Il est plus facile de se représenter ces valeurs sous forme de tableau :

Passage dans la boucle	i	Nombres[i]	Résultat
avant	1	4	0
1	1	4	4
2	2	5	9
3	3	6	**15**

Cet algorithme permet de calculer la somme des nombres contenus dans une liste (ici la liste `Nombres`).

Pour comprendre ce que fait l'algorithme ci-dessus, il faut se mettre à la place de la machine. On parle de *simuler* un algorithme, de faire comme si l'algorithme s'exécutait sur une machine. Pour que ce soit plus concret, on peut imaginer des valeurs fictives pour les variables telles que `Nombres`. Dans la vie réelle, `Nombres` pourra contenir tous les nombres possibles, mais cela ne nous aide pas à comprendre. On imagine alors des nombres précis que `Nombres` pourrait contenir, comme par exemple `4`, `5` et `6`. Lorsqu'on exécute les opérations de l'algorithme l'une après l'autre, avec des valeurs concrètes, on comprend mieux quel effet ces opérations ont sur les valeurs contenues dans les variables. La simulation de l'algorithme nous permet de saisir **les calculs** réalisés par cet algorithme, ici une simple somme.

Exercice 2 – Machine mystère

Quel objet du quotidien (autre que la calculatrice) fait des additions et utilise cet algorithme pour résoudre un problème ?

Il y a-t-il des avantages à automatiser cette tâche, à demander à une machine de le faire à la place d'un humain ?

Il y a-t-il des désavantages à automatiser cette tâche ?

Solution 2 – Machine mystère

Une caisse enregistreuse ! La caisse enregistreuse calcule la somme des prix des produits contenus dans un panier (une liste de courses) et nous donne le prix total à payer. Il s'agit d'un exemple parmi d'autres.

Au niveau des avantages, la caisse enregistreuse fait bien moins d'erreurs qu'un humain, elle ne se fatigue pas, elle ne se plaint pas et elle est bien plus rapide.

Au niveau des désavantages, l'automatisation est en général énergivore (avec une empreinte environnementale significative) et provoque une certaine « obsolescence des humains » en les remplaçant dans leur travail pour un moindre coût financier.

« Chaque étape d'un algorithme doit être définie précisément » (Knuth, 2011). En effet, si on ne décompose pas suffisamment la solution du problème, on peut se retrouver face à une recette inutile, par exemple : prendre des œufs et cuire l'omelette. Cette recette ne nous dit pas vraiment comment procéder pour arriver à faire une omelette…

En lien

Lorsqu'on sauve un fichier dans un ordinateur, il est stocké dans une mémoire. La mémoire d'un ordinateur pourrait être comparée a une grande commode de tiroirs étiquetés. Ainsi, lorsqu'un fichier est stocké en mémoire, la taille du fichier correspond au nombre de tiroirs qu'il occupe. Si c'est un fichier de texte par exemple, on peut imaginer qu'un tiroir contient un caractère simple (un octet). Si c'est une image en couleur, un pixel de cette image occuperait 3 tiroirs (un octet par couleur rouge, vert et bleu).

1.1.2 Les ingrédients d'un algorithme

L'objectif d'un algorithme est de décrire la solution à un problème donné. Concrètement, pour résoudre un problème, l'algorithme va utiliser des **données** qu'il reçoit ***en entrée*** et va retourner un **résultat** ***en sortie***. Le résultat en sortie va être la solution au problème sur la base des calculs effectués sur les données en entrée. Un exemple d'algorithme qui détecte les visages reçoit en *entrée* une image (ce sont les *données*) et

retourne en sortie «oui» ou «non» (c'est le résultat) selon si l'image contient un visage ou pas. Les données en entrée d'un algorithme qui traduit pourraient être le mot à traduire et un dictionnaire. L'algorithme traiterait ces données pour retourner en *sortie* la traduction du mot dans une autre langue.

Entre l'entrée et la sortie, l'algorithme précise les **opérations** qu'il faut exécuter sur les données en entrée. Les opérations que l'on peut demander à un humain sont très différentes de celles que l'on peut demander à une machine. On peut demander à un humain de casser des œufs, mais un ordinateur ne peut pas comprendre et réaliser cette opération. Par contre on peut demander à un ordinateur de se souvenir de milliers de valeurs stockées dans des variables et de comparer les valeurs de toutes ces variables entre elles sans faire d'erreur. Pour résoudre un problème, l'humain cherche une solution sur la base des données à disposition, et la décrit sous la forme d'opérations dans un algorithme. Dans un deuxième temps, ces opérations sont retranscrites en une suite d'instructions élémentaires dans un programme informatique, exécutable par une machine. Dans un troisième temps on vérifie si la solution obtenue est correcte, et si besoin on corrige l'algorithme.

Le dernier ingrédient de l'algorithme, mais tout aussi important, est l'**ordre des opérations**. Dans l'exemple de l'omelette, on ne peut cuire les œufs avant de les avoir cassés, sinon on obtiendrait des œufs durs. De même, l'ordinateur a besoin de recevoir les instructions élémentaires à exécuter dans le bon ordre. Pour résumer, les ingrédients pour concevoir un algorithme sont les suivants :

1. Des données en entrée.
2. Des opérations, dans un ordre précis.
3. Un résultat en sortie.

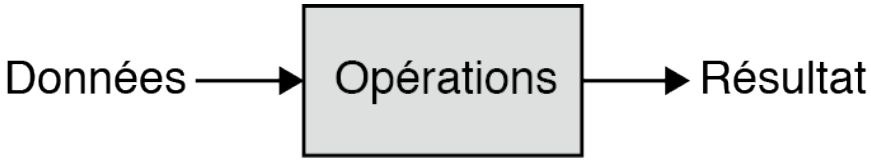


Fig. 1.4 – Schéma des ingrédients d'un algorithme. Un algorithme reçoit des données en entrée, qu'il traite selon des opérations dans un ordre précis, dans le but de produire un résultat en sortie. Ce résultat représente la solution à un problème donné.

Notez que les opérations d'un algorithme doivent être précises et ***non ambigües***. Il doit y avoir une seule interprétation possible de l'algorithme. Une recette de cuisine ne serait pas assez précise pour une machine, par exemple, il faudrait indiquer clairement ce que température moyenne et mélange homogène veulent dire. Les êtres humains peuvent interpréter, deviner et supposer, mais pas les machines (pour l'instant).

Le saviez-vous ? – Jeu d'instructions

Le jeu d'instructions élémentaires dépend du système informatique sur lequel elles s'exécutent. Nous avons vu qu'un algorithme spécifie des opérations à suivre dans un ordre donné afin de résoudre un problème. Ces opérations sont transcrites sous la forme d'un programme informatique en instructions élémentaires exécutables par une machine, qui peuvent être très différentes d'une machine à l'autre pour un même algorithme. Ainsi, l'algorithmique permet d'aborder la résolution de problèmes de manière générale, sans se préoccuper des détails d'implémentation sur différents systèmes.

Exercice 3 – Ingrédients de l'algorithme mystère

A quoi correspondent « les ingrédients d'un algorithme » dans l'exemple de la recette de l'omelette ?

Solution 3 – Ingrédients de l'algorithme mystère

Les données en entrée sont les œufs, les opérations sont les étapes 1 à 4 de la recette et finalement le résultat en sortie est l'omelette. On peut considérer le matériel culinaire (bol, fourchette, poêle, spatule) comme du matériel informatique à notre disposition, capable de traiter des données (œufs). En effet, on peut cuire plein d'autres aliments dans une poêle.

Exercice 4 – Échange de deux variables

Écrire un algorithme qui échange les valeurs de deux variables. Par exemple, si la première variable `X` contient `1` et la deuxième variable `Y` contient `2`, à la fin de l'algorithme `X` contient `2` et `Y` contient `1`. Pour rappel, une variable peut contenir une seule valeur à la fois.

Conseil : cela aide de se mettre à la place de la machine et de représenter le contenu de chaque variable sous la forme d'un tiroir, en la dessinant avec son étiquette et son contenu *après chaque opération de votre algorithme*.

Solution 4 – Échange de deux variables

Pour commencer, la variable `X` contient `1` et la variable `Y` contient `2`. Une solution naïve consisterait à écrire l'algorithme suivant :

```
X ← Y
Y ← X
```

Cet algorithme met la valeur de `Y` dans `X`, puis la valeur de `X` dans `Y`. Représentons maintenant ces deux variables par des tiroirs étiquetés. Le premier tiroir s'appelle `X` et contient `1`, le deuxième s'appelle `Y` et contient `2` :

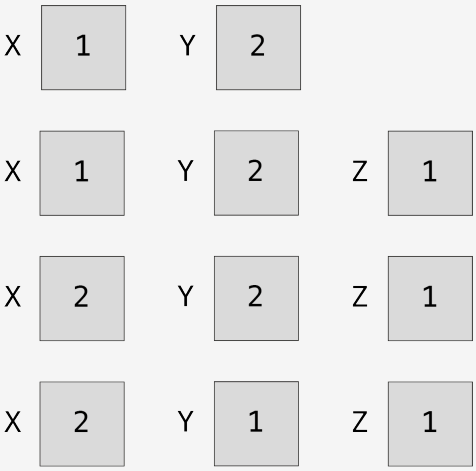
Après la première opération où on met la valeur de `Y` dans la variable `X` on se retrouve avec cette situation, où la valeur contenue dans `Y` écrase la valeur qui était contenue dans `X` :

En effet, un tiroir ne peut contenir qu'une seule valeur ! Nous n'avons donc plus accès à la valeur qui était stockée dans la variable `X` avant d'y mettre celle de `Y`. Pour remédier à ce problème, il faut penser à utiliser une variable temporaire `Z` qui permet de se souvenir de la valeur initiale de `X`.

Un algorithme correct pour échanger les valeurs de deux variables est :

```
Z ← X
X ← Y
Y ← Z
```

Si on dessine l'état des variables après chacune de ces opérations dans des tiroirs, voici ce qu'on obtient :



Nous avons donc la confirmation que la solution obtenue résout correctement notre problème d'échange des valeurs de deux variables.

1.1.3 Exercices

Exercice 5 – Forme mystère

L'algorithme suivant contrôle un crayon. Quelle forme dessine-t-il ?

```
Répéter 8 fois:
    Avance de 2 cm
    Tourne à droite de 60°
```

Exercice 6 – Nombre minimum

Ecrire un algorithme qui permet de trouver le plus petit nombre d'une liste. Penser à décomposer la solution en différentes étapes.

Appliquer l'algorithme à la liste `[3, 6, 2, 8, 1, 9, 7, 5]`.

L'algorithme trouve-t-il la bonne solution ? Si non, modifier l'algorithme afin qu'il trouve la bonne solution.

Exercice 7 – Le prochain anniversaire

On souhaite déterminer l'élève dont la date d'anniversaire est la plus proche de la date d'aujourd'hui, dans le futur. Ecrire un algorithme (en langage familier) qui permet de trouver cet élève. Penser à décomposer le problème en sous-problèmes.

Comparer la solution trouvée à celle de la personne à côté de vous. Avez-vous procédé de la même manière ? Si non, expliquer vos raisonnements.

Un ordinateur peut-il réaliser les opérations décrites par cet algorithme ?

Exercice 8 – Échange de trois variables

Écrire un algorithme qui effectue la permutation circulaire des variables `X`, `Y` et `Z` : à la fin de l'algorithme, `X` contient la valeur de `Z`, `Y` la valeur de `X` et `Z` la valeur de `Y`. Pour rappel, une variable ne peut contenir qu'une valeur à la fois.

Conseil : il est très utile de se mettre à la place de la machine et de représenter le contenu de chaque variable sous la forme d'un tiroir, en dessinant le tiroir avec son étiquette et son contenu *après chaque opération de l'algorithme*. Est-ce que votre algorithme donne le résultat attendu ? Si non, modifier l'algorithme pour qu'il résolve le problème correctement.

Exercice 9 – Affectations

Quel est le résultat de la suite des trois affectations suivantes ? On parle d'*affectation* lorsqu'on attribue une valeur à une variable.

```
X ← X + Y
Y ← X - Y
X ← X - Y
```

Vérifier la solution que vous avez trouvée en représentant chaque variable avec une valeur fictive. Suivre les opérations dans l'ordre et dessiner le contenu des variables après chaque étape.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je connais la différence entre un algorithme et un programme.
2. Je sais simuler un algorithme : je représente les valeurs des variables après chaque opération de l'algorithme.
3. Je sais formuler un algorithme : je décompose le problème en sous-problèmes et je décris les opérations qui permettent de résoudre chaque sous-problème.

1.2 Trie, cherche et trouve

Matière à réfléchir – Bibliothèque inutile

Imaginez une bibliothèque scolaire un peu spéciale : les livres n'y sont pas rangés par ordre alphabétique ! Ils sont bien rangés sur des étagères, mais sans aucune logique particulière. Vous entrez dans cette bibliothèque un peu spéciale et vous vous mettez à chercher l'ouvrage ***Le Guide du voyageur galactique.***

Pensez-vous pouvoir retrouver ce livre ? Combien de temps cela vous prendra-t-il ?

Y a-t-il des objets chez vous, que vous rangez dans un ordre bien particulier ?

Y a-t-il des objets chez vous, que vous feriez mieux de ranger dans un ordre bien particulier, parce que vous passez beaucoup de temps à les chercher ?

Pour l'instant il faut nous croire sur parole, mais si l'on veut pouvoir trouver une information parmi une avalanche d'informations, il faut que ces informations soient bien rangées. L'exemple de la bibliothèque ci-dessus illustre ce besoin de manière intuitive, mais vous allez pouvoir l'expérimenter concrètement dans le chapitre Algorithmique II.

Saviez-vous que le succès fulgurant de *Google* est surtout dû à sa capacité à bien ranger l'information disponible sur le Web ? Au moment où vous avez besoin d'une information particulière, leurs algorithmes sont capables de la retrouver parce qu'elle est bien rangée. Ce problème qui consiste à ranger les données a un nom, il s'agit du **problème du Tri**. Il est si important qu'il est un des problèmes les plus étudiés en algorithmique.

1.2.1 Algorithmes de tri

Un **algorithme de tri** est un algorithme qui permet de résoudre le problème du tri des données, donc d'organiser les données selon ***une relation d'ordre***. Dans la figure ci-dessous, les objets sont organisés soit par la luminosité de leur couleur (ligne du haut), soit par leur taille (lignes du bas), dans un **ordre croissant**.

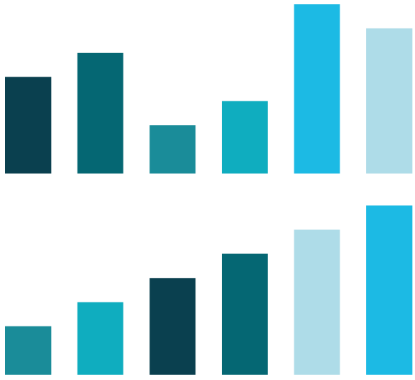


Fig. 1.5 – Problème du tri. Des objets sont triés selon une relation d'ordre, en lien avec une propriété. Sur la ligne du haut, les rectangles sont organisés selon leur couleur (de la plus sombre à la plus claire), alors que sur la ligne du bas, ils sont triés selon leur taille (du plus petit au plus grand).

Exercice 1 – Problème du tri

Trier les rectangles de la ligne du haut de la figure précédente en fonction de leur taille, pour arriver à la disposition de la ligne du bas. Noter toutes les étapes intermédiaires de vos actions et la disposition des rectangles avant d'arriver à la solution finale. Conseil : remplacer les rectangles par un nombre qui représente leur taille.

En lien avec les ingrédients d'un algorithme, déterminer les données en entrée et le résultat en sortie de l'algorithme.

Quels types d'opérations avez-vous effectuées ?

Solution 1 – Problème du tri

Si on remplace les rectangles de la ligne du haut par un nombre qui représente leur taille, on obtient la liste `[3, 4, 1, 2, 6, 5]`. Le plus important est que l'ordre des nombres conserve l'ordre de la taille des rectangles. Après le tri, si l'algorithme est correct, vous devriez vous retrouver avec la liste `[1, 2, 3, 4, 5, 6]`. Les opérations et les dispositions intermédiaires exactes dépendent de l'algorithme que vous avez utilisé.

Les données en entrée sont les rectangles sur la ligne du haut : leur taille et l'ordre de leur taille, ici `[3, 4, 1, 2, 6, 5]`. Le résultat en sortie correspond aux rectangles sur la ligne du bas : l'ordre croissant de leur taille, ici `[1, 2, 3, 4, 5, 6]`.

Les types d'opérations que vous avez effectuées sont des comparaisons de la taille de deux rectangles et des déplacements de rectangles.

Nous allons exposer ici **trois algorithmes de tri simple**, que l'on pourrait utiliser pour trier des objets dans la vie de tous les jours.

1.2.2 Tri par insertion

L'*algorithme* du **tri par insertion** parcourt la liste d'éléments à trier du deuxième au dernier élément. Pour chaque nouvel élément considéré, il l'insère à l'emplacement correct dans la liste déjà parcourue. A tout moment, la liste d'éléments déjà parcourus (jusqu'à l'élément que l'on considère à un moment donné) est toujours bien triée.

1.2.3 Tri par sélection

L'*algorithme* du **tri par sélection** commence par rechercher le plus petit élément de la liste et l'échange avec le premier élément de la liste. Il recherche ensuite le plus petit élément de la liste restante (sans le premier plus petit élément). Il sélectionne ainsi le deuxième plus petit élément de la liste et l'échange avec le deuxième élément de la liste. Et ainsi de suite : il recherche le plus petit élément de la liste restante, en excluant les éléments déjà triés, et échange ce plus petit élément avec le premier élément pas encore trié. Il continue de la sorte jusqu'à arriver au dernier élément de la liste.

1.2.4 Tri à bulles

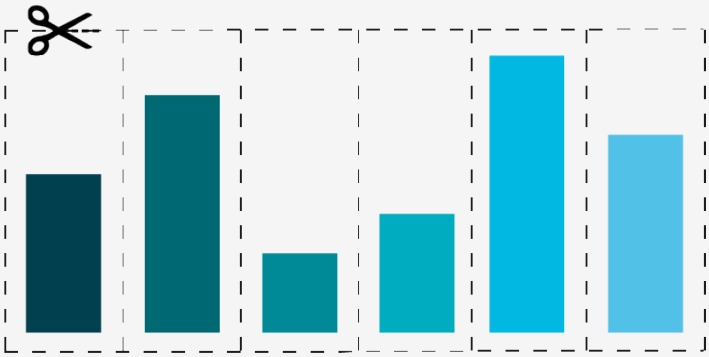
L'*algorithme* du **tri à bulles** compare les éléments voisins, deux par deux. Il commence par comparer les deux premiers éléments de la liste et les met dans le bon ordre (le plus petit des deux éléments précède le plus grand des deux). Il compare ensuite les deux éléments suivants (le nouveau deuxième et le troisième élément de la liste) et les met dans le bon ordre. Il continue de la sorte jusqu'à la fin de la liste. Après ce premier parcours de la liste, le plus grand élément se retrouve en dernière position de la liste. L'algorithme parcourt à nouveau la liste, en comparant et en déplaçant les éléments voisins deux par deux (en excluant également le dernier élément qui est déjà bien trié). Après le deuxième parcours de la liste, le deuxième plus grand élément se retrouve en avant-dernière position de la liste. L'algorithme parcourt la liste de la sorte, autant de fois qu'elle possède d'éléments, en excluant les éléments bien triés en fin de la liste.

Exercice 2 – Algorithme de tri

Il est fortement recommandé de résoudre cet exercice avant d'avancer dans le chapitre.

Appliquer au moins un des trois algorithmes ci-dessus (tri par insertion, tri par sélection et tri à bulles) pour trier les rectangles de la ligne du haut de la figure **Problème du tri** en fonction de leur taille (le résultat est illustré dans la ligne du bas).

Noter l'ordre des éléments à chaque fois qu'il change. Vous aurez besoin d'une grande feuille de papier. Vous pouvez aussi représenter la taille des rectangles par un nombre, cela permet de gagner de la place. Si cela vous aide, vous pouvez découper les rectangles ci-dessous et les manipuler.



Solution 2 – Algorithme de tri

La solution est donnée dans la suite du chapitre et est illustrée dans la figure **Algorithmes de tri**.

Remarque – Quand on cherche on trouve. Vraiment ?

Vous passez trop de temps à chercher vos affaires ? Pensez à mieux les trier. Le temps perdu à ranger vos affaires sera bien inférieur à celui que vous passerez à les chercher plus tard.

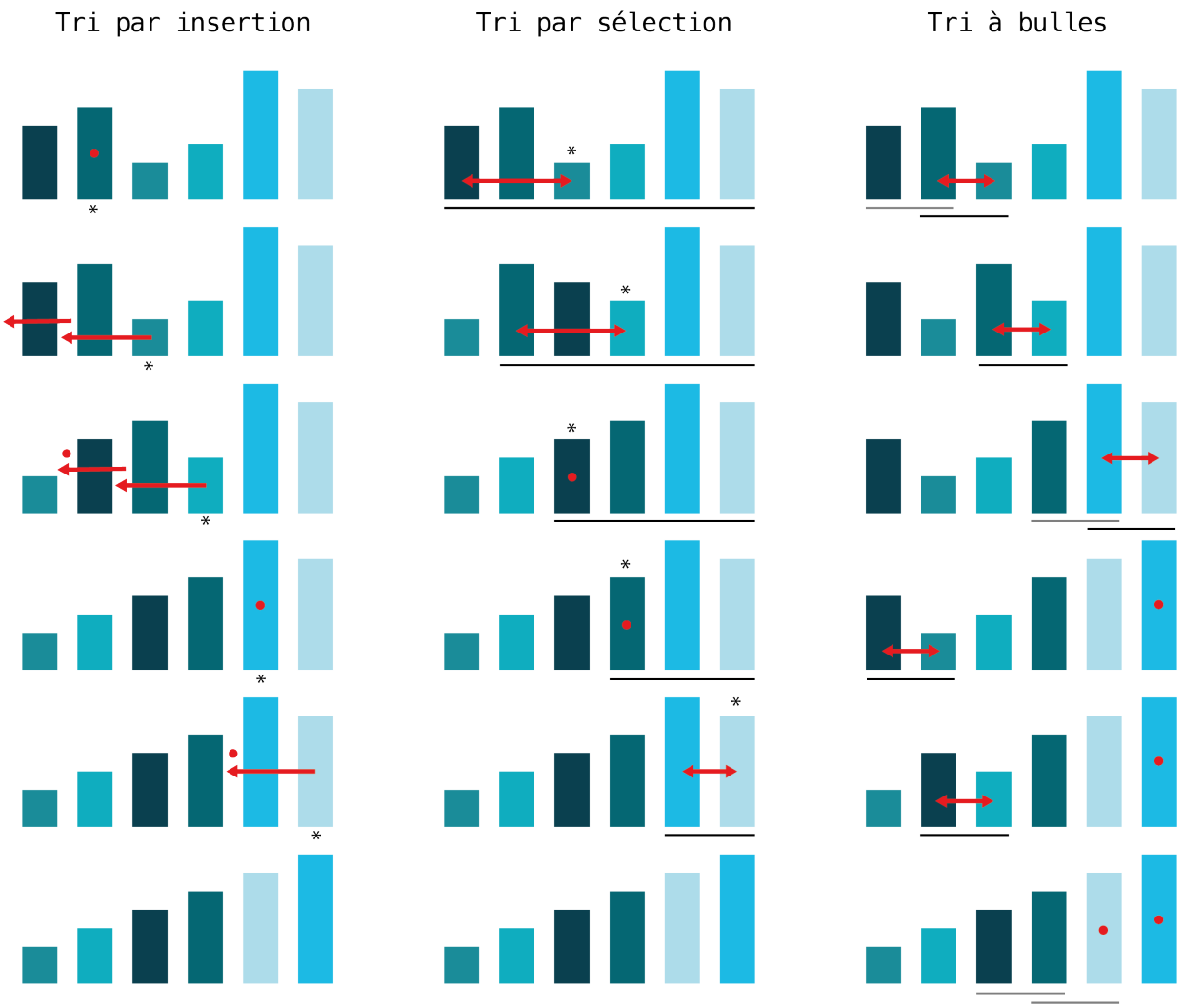


Fig. 1.6 – Algorithmes de tri. Etapes intermédiaires lors de l'application des différents algorithmes de tri. La flèche rouge montre les mouvements des éléments suite à une opération. Si l'élément ne bouge pas, la flèche rouge est remplacée par un point rouge. **A gauche**, le tri par insertion. L'étoile dénote l'élément considéré à un moment donné. **Au milieu**, le tri par sélection. L'étoile désigne le plus petit élément de la liste non triée. **A droite**, le tri à bulles. Ici le point rouge signale les éléments triés.

La figure ci-dessus détaille les étapes intermédiaires des trois *algorithmes* de tri vus précédemment. Dans le **tri par insertion** à gauche, on parcourt la liste dans l'ordre, un élément après l'autre (dénoté par une étoile). A chaque étape, on cherche à ***insérer*** le rectangle considéré à la bonne place dans la liste précédemment triée. La flèche rouge montre la position à laquelle le rectangle sera inséré après comparaison avec l'élément précédent. Si l'élément est déjà bien trié, aucune action n'est requise dans ce cas et la flèche est remplacée par un point rouge. Notez que la liste qui précède le rectangle considéré (celui avec l'étoile) est toujours bien triée.

Dans le **tri par sélection** au milieu, on parcourt la liste pour ***sélectionner*** son plus petit élément, et on le met à la bonne position. La ligne noire au-dessous des rectangles montre la liste parcourue pour rechercher le plus petit élément. Le plus petit élément de cette liste est désigné par l'étoile. Finalement, la flèche rouge montre les éléments échangés : le premier élément de la liste non triée et le plus petit élément. Ainsi, le

plus petit élément sélectionné (avec étoile) se retrouve à la fin de la liste déjà triée (liste non soulignée). Si l'élément est déjà bien trié et qu'aucune action n'est requise, la flèche bidirectionnelle est remplacée par un point rouge.

Dans **le tri à bulles** à droite, les lignes en dessous des rectangles montrent les éléments voisins qui sont comparés à chaque étape. Lorsque cette ligne est grise, les éléments sont déjà bien ordonnés et aucune action n'est requise. Lorsque la ligne est noire, les éléments ne sont pas dans le bon ordre et doivent être intervertis (flèche rouge). Après un passage complet de la liste, l'élément le plus grand se retrouve en dernière position, il remonte comme une ***bulle*** (voir la 4e ligne). Le point rouge ici indique les éléments triés. Dans ce cas, la liste est triée après deux parcours complets de la liste.

Notez que même si tous les *algorithmes* arrivent à la même solution finale, ils y arrivent de manière très différente et avec plus ou moins de calculs.

Exercice 3 – Votre algorithme de tri

Rappelez-vous quelle méthode vous avez utilisée pour résoudre le premier exercice. De quel algorithme de tri se rapproche-t-elle le plus ?

Solution 3 – Votre algorithme de tri

Cela dépend de votre solution du premier exercice. Vous avez probablement utilisé la méthode du tri par sélection ou du tri à bulles.

Le saviez-vous ? – Tri stupide

Il existe un algorithme, **Tri de Bogo** (ou *Bogosort*), aussi nommé le *tri lent* ou encore le *tri stupide*. C'est un tri qui génère différentes permutations des éléments de la liste et s'arrête lorsque la configuration obtenue par hasard est triée. A votre avis, combien d'opérations prend cet algorithme en moyenne ?

Exercice 4 – Opérations

Pour chaque algorithme de tri, compter le nombre de ***comparaisons*** de la taille de deux rectangles, ainsi que le nombre de ***déplacements*** (le nombre de fois que deux rectangles échangent leur place).

Imaginons que ce qui prend le plus de temps est une ***comparaison***. Dans ce cas précis, quel algorithme de tri parmi les trois algorithmes présentés est le plus lent ?

Imaginons que ce qui prend le plus de temps est un ***déplacement***. Dans ce cas précis, quel algorithme de tri est le plus lent ? Quel algorithme est le plus rapide ?

Solution 4 – Opérations

Le décompte des opérations effectuées, en se référant à la figure **Algorithmes de tri** est comme suit :

Tri par insertion : 9 comparaisons deux par deux (flèches et points rouges) et 5 déplacements deux par deux (flèches rouges). Notez que pour insérer un élément en première position, il faut tout d'abord l'échanger avec l'élément juste devant, puis avec l'élément avant, et ainsi de suite jusqu'à arriver à la première position.

Tri par sélection : 15 comparaisons deux par deux (lignes en dessous) et 3 déplacements deux par deux (flèches rouges).

Tri à bulles : 9 comparaisons deux par deux (lignes en dessous) et 5 déplacements deux par deux (flèches rouges).

Si ce qui prend beaucoup de temps est la comparaison de la taille de deux rectangles, il ne faudrait pas utiliser le tri par sélection, car il comporte le plus grand nombre de comparaisons et il serait le plus lent. Si c'est le déplacement de deux rectangles qui coûte beaucoup de temps, cette fois-ci le tri par sélection serait le plus rapide (avec 3 rectangles qui échangent leur position). Donc, selon l'implémentation sur la machine, le tri par sélection serait le plus lent ou le plus rapide des trois algorithmes.

Ces résultats sont valables pour cette configuration en particulier. Si on trie un autre tableau, la performance des trois algorithmes pourrait changer. Le choix du meilleur algorithme dépend donc de l'implémentation et de la situation initiale. Notez finalement qu'il existe des algorithmes de tri bien plus rapides que les trois algorithmes considérés ici.

1.2.5 Comparaison d'algorithmes

Toutes les recettes de cuisine ne se valent pas, de la même manière, un *algorithme* peut aussi être ***plus approprié*** qu'un autre algorithme pour résoudre le même problème. Il existe des dizaines d'*algorithmes* qui trient avec des approches différentes (nous en verrons encore quelques-uns). Certains algorithmes sont plus rapides, d'autres plus économes en mémoire ou encore plus simples à coder. Ainsi, selon la situation, il faut choisir le « bon » *algorithme*.

La qualité d'un *algorithme* dépend de la propriété que l'on souhaite optimiser (maximiser ou minimiser). Cela pourrait être de maximiser la **vitesse d'exécution** (mesurée par le nombre d'*instructions* élémentaires exécutées), de minimiser la place occupée en **mémoire**, de minimiser la **consommation d'énergie** ou de maximiser la **précision de la solution**. L'*algorithme* utilisé devrait être choisi en fonction de ce qui est important.

La vitesse d'un algorithme dépend également des données en entrée. Selon la configuration initiale des *données* en *entrée* (correspond à la ligne du haut de la figure **Algorithmes de tri**), un *algorithme* « rapide » peut devenir « lent » et *vice versa*. Il faut savoir que les *algorithmes* vus jusqu'ici sont tous des *algorithmes* lents (nous verrons un *algorithme* de tri rapide ultérieurement).

Le saviez-vous ? – Tri trop lent

Pour trier 1 million d'éléments, selon l'algorithme choisi, cela peut prendre de 20 millions à 1 billion d'opérations. Si chaque opération prenait 1 microseconde ($10^{-6}s$) à s'exécuter, il faudrait 20 secondes pour trier 1 million d'éléments si l'algorithme est efficace. Par contre, pour un des algorithmes ci-dessus, cela pourrait prendre 11 jours !

Pour aller plus loin

Imaginer que les quatre éléments d'une liste sont triés dans le sens inverse de ce que l'on souhaite (ils sont placés du plus grand au plus petit). Trier la liste selon les trois algorithmes de tri vus précédemment : le tri par insertion, le tri par sélection et le tri à bulles.

Dans cette configuration précise, quel algorithme est le plus rapide (présente le moins d'étapes intermédiaires) ?

Et quel algorithme est le plus lent ?

1.2.6 Exercices

Exercice 5 – L'algorithme de votre journée

Réfléchir à votre journée : y a-t-il des actions qui se retrouvent chaque jour ouvrable ? Arrivez-vous à esquisser un algorithme que vous suivez sans que vous en ayez conscience ?

Exercice 6 – Trois algorithmes de tri

Trier la liste [2, 5, 3, 4, 7, 1, 6] en utilisant les trois algorithmes de tri vus dans le cours. Représenter l'état de la liste après chaque étape.

Exercice 7 – Vérificateur de tri

Ecrire un algorithme qui vérifie si une liste est triée.

Que prend l'algorithme en entrée et que retourne-t-il en sortie ?

Demander ensuite à un autre élève de suivre les opérations décrites par votre algorithme. Est-ce que votre algorithme est correct ?

Comparer vos algorithmes. Sont-ils différents ?

Exercice 8 – Mondrian

Analyser les œuvres cubistes de Piet Mondrian. Trouver un algorithme qui permet de créer une œuvre qui pourrait être attribuée à Mondrian.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais qu'il existe plusieurs manières différentes de résoudre un problème.
2. Je sais qu'il faut choisir le meilleur algorithme en fonction de critères objectifs : vitesse de l'algorithme, qualité de la solution, espace utilisé en mémoire ou encore consommation d'énergie.
3. Je sais appliquer les trois algorithmes de tri vus dans le cours.

1.3 Des algorithmes aux programmes

Matière à réfléchir – Lieu mystère

Pensez à un lieu connu, qui se trouve à proximité. Ecrivez les étapes à suivre pour s'y rendre, sans mentionner le lieu. Vous ne pouvez utiliser que les instructions : **avancer**, **tourner à gauche** et **tourner à droite**.

Demandez à vos camarades de classe de suivre ces instructions. Sont-ils arrivés à deviner à quel lieu ils se sont rendus ?

Si non : essayez de comprendre à quel moment ils se sont perdus. Adaptez votre algorithme en fonction.

Si oui : reformulez vos instructions en utilisant les mots-clés **si (if)**, **sinon (else)** ou **tant que (while)**.

[Optionnel] Imaginez que votre camarade peut uniquement **avancer de 1m tout droit** et **tourner de 15 degrés**. Reformulez votre solution en utilisant en plus le mot clé **répéter (for)**.

Une fois que l'on a déterminé le meilleur *algorithme* à utiliser, pour l'automatiser, il faut le retranscrire dans un *programme* qu'une machine peut comprendre. Nous allons détailler ce processus pour l'algorithme du **tri par sélection**.

Cet algorithme consiste à parcourir la liste à trier plusieurs fois. A chaque *itération*, on sélectionne le plus petit élément et on l'échange avec le premier élément de la liste non triée. Comment pourrait-on traduire ceci en Python ? Comment représenter les rectangles dans un langage de programmation ?

Tout d'abord, il faut représenter la taille des rectangles par des nombres. On peut par exemple représenter l'ordre des rectangles de la première ligne de la figure **Trier** en fonction de leur taille, dans une liste nommée `rect` :

```
rect = [3, 4, 1, 2, 6, 5]
```

On doit ensuite ***parcourir la liste*** pour trouver le plus petit élément de la liste, qui correspond au rectangle le plus court. Nous allons commencer par ***déclarer une variable***, nommée `indice_min`, qui va se souvenir de la position du plus petit élément de la liste (équivalent à l'indice de l'élément à l'intérieur de la liste). Pour commencer, nous supposons que le plus petit élément de la liste est le premier élément, et nous initialisons la variable nommée `indice_min` à `0`.

```
# initialise une variable qui va se souvenir du plus petit rectangle de la liste
indice_min = 0
```

Nous allons ensuite parcourir la liste à partir du deuxième élément. Pour chaque nouvel élément, nous allons tester s'il est plus petit ou plus grand que le plus petit élément connu jusqu'alors. Si le nouvel élément est plus petit que l'élément désigné par `indice_min`, c'est l'indice du nouvel élément qui sera stocké dans `indice_min` à la place de l'ancien :

```
# for permet de parcourir la liste rect
for i in range(1,len(rect)):  # len(rect) donne la longueur de la liste rect

    # identifie l'indice du plus petit élément de la liste
    if rect[i] < rect[indice_min] :
        indice_min = i
```

Pour faire plus simple, nous pouvons également utiliser la *fonction* Python **min()** qui retourne directement le plus petit élément d'une liste. Nous avons aussi besoin de la fonction **index()** afin d'accéder à la position (ou l'indice) du plus petit élément.

```
# identifie l'indice du plus petit élément de la liste
indice_min = rect.index(min(rect))
```

Grâce à ces fonctions Python préexistantes, nous avons remplacé les 3 lignes du code au-dessus par une seule ligne de code. Après cette opération, `indice_min` contient l'indice du plus petit élément de la liste. On doit à ce stade, échanger cet élément et le premier élément. Comme nous avons pu le voir avant, pour échanger les valeurs de deux variables, nous avons besoin d'une ***variable temporaire***. En effet, si on met la valeur du plus petit élément directement à la position 0, nous perdons la valeur contenue à la position 0 à ce moment-là. Il faut donc la stocker temporairement dans une autre variable :

```
# échange le plus petit élément avec le premier élément
rect_temp = rect[0]
rect[0] = rect[indice_min]
rect[indice_min] = rect_temp
```

Là encore, Python permet d'écrire ces trois lignes de manière beaucoup plus compacte. En affectant les deux variables simultanément, c'est Python qui se charge de créer la variable temporaire :

```
# échange le plus petit élément avec le premier élément
rect[0], rect[indice_min] = rect[indice_min], rect[0]
```

On doit ensuite refaire exactement les mêmes opérations (parcourir à nouveau la liste pour trouver le plus petit élément et échanger sa position), mais en excluant le premier élément de la liste qui est maintenant bien trié. Donc on va rechercher le plus petit élément de la liste restante, et l'échanger cette fois-ci avec le deuxième élément de la liste (attention, il s'agit de la position 1 et non 2 en Python). On adapte le code précédent :

```
# trouve le plus petit rectangle de la liste rect[1:] (à partir du 2e élément)
indice_min = rect.index(min(rect[1:]))

# échange le plus petit élément avec le deuxième élément
rect[1], rect[indice_min] = rect[indice_min], rect[1]
```

La suite de l'algorithme consiste à nouveau à rechercher le plus petit élément de la liste restante (en excluant cette fois-ci le premier et deuxième élément, qui sont bien triés) et l'échanger avec le troisième élément (premier élément non trié). À nouveau on peut reprendre le même code, mais on incrémente tous les indices de 1 :

```
# trouve le rectangle le plus petit de la liste rect[2:] (à partir du 3e élément)
indice_min = rect.index(min(rect[2:]))

# échange le plus petit élément avec le troisième élément
rect[2], rect[indice_min] = rect[indice_min], rect[2]
```

On détecte un motif qui se répète. On fait toujours les mêmes opérations, mais en commençant à une position différente. On peut réécrire le même code autant de fois que d'éléments dans la liste, mais ce n'est pas optimal si la liste est longue et si on veut pouvoir réutiliser ce code pour une liste de longueur différente. Il vaut mieux remplacer l'indice qui change par une variable que l'on *incrémente* (augmente). Notez que ce code est répété `len(rect)-1` fois et pas autant de fois qu'il y a d'éléments de la liste, car on doit pouvoir comparer et échanger deux éléments.

```
# pour tous les éléments de la liste non triée
for j in range(0,len(rect)-1):

    # trouve le rectangle le plus petit de la liste rect[j:] (à partir de l'élément j)
    indice_min = rect.index(min(rect[j:]))

    # échange le plus petit élément avec le j-ième élément
    rect[j], rect[indice_min] = rect[indice_min], rect[j]
```

Le principal avantage de cette **factorisation** (réécriture) est que maintenant notre code fonctionne pour toutes les longueurs de listes. Nous n'avons plus besoin de savoir à l'avance combien d'éléments sont contenus dans la liste (combien de fois répéter les opérations). Au lieu de répéter le code un nombre prédéterminé de fois, le code s'exécute autant de fois qu'il y a d'éléments dans la liste (moins 1, car on compare toujours 2 éléments).

L'étape suivante consiste à encapsuler tout le code dans une **fonction** qui reçoit la liste comme ***paramètre***, afin de le rendre utilisable par d'autres programmes sans avoir à copier-coller le code. Cela permet aussi en cas d'erreur de facilement corriger la fonction, plutôt que de corriger le code partout il a été copié-collé. Pour que la fonction soit utilisable, il ne faut pas oublier de rajouter le `return` qui retourne le résultat.

```
# Tri par sélection
def tri_selection(rect) :

    # pour tous les rectangles de la liste non triée
    for j in range(0,len(rect)-1):

        # trouve le rectangle le plus petit de la liste rect[j:]
        indice_min = rect.index(min(rect[j:]))

        # échange le plus petit élément et le j-ième élément
```

(suite sur la page suivante)

(suite de la page précédente)

```
        rect[j], rect[indice_min] = rect[indice_min], rect[j]

    return(rect)
```

Finalement le terme `rect` n'est pas assez général, car le tri par sélection peut être utilisé pour trier toutes sortes d'éléments et pas seulement des rectangles. Ainsi on peut renommer la *variable* `rect` par le terme plus général `liste`, partout où `rect` apparait dans le code ci-dessus :

```
# Tri par sélection
def tri_selection(liste) :

    # pour tous les éléments de la liste non triée
    for j in range(0,len(liste)-1):

        # trouve l'élément le plus petit de liste[j:]
        indice_min = liste.index(min(liste[j:]))

        # échange le plus petit élément et le j-ième élément
        liste[j], liste[indice_min] = liste[indice_min], liste[j]

    return(liste)
```

Pour trier la liste `rect` définie au tout début, il suffit d'appeler la fonction `tri_selection` avec la liste `rect` en *argument*. La fonction **print()** permet d'afficher la liste triée :

```
# trier la liste de rectangles par sélection
rect = [3,4,1,2,6,5]
print(tri_selection(rect))
```

En traduisant les étapes intermédiaires du tri par sélection en des lignes de code, nous avons automatisé l'algorithme. Nous l'avons transcrit en un programme informatique qui peut être exécuté sur une machine.

1.3.1 Exercices

Exercice 1 – Jeu de la devinette

Ecrire le programme suivant : le programme pense à un nombre au hasard. Lorsque vous lui proposez un nombre, il vous dit si « c'est plus » ou si « c'est moins » jusqu'à ce que vous trouvez le bon nombre. Conseil : utiliser le module Python *random*.

Y a-t-il une stratégie gagnante ?

Exercice 2 – Plus petit nombre

Transcrire l'algorithme de l'exercice qui permet de déterminer le plus petit nombre d'une liste, en un programme Python.

Exercice 3 – Programmes de tri

Implémenter le tri à bulles et/ou le tri par insertion vus au cours.

Créer une liste qui contient les valeurs de `1` à `n` dans un ordre aléatoire, où `n` prend la valeur 10, par exemple. Vous pouvez utiliser la fonction `shuffle()` du module random.

Pour aller plus loin.

A l'aide du module `time` et de sa fonction `time()`, chronométrer le temps prend le tri d'une liste de 100, 500, 1000, 10000, 20000, 30000, 40000 puis 50000 nombres.

Noter les temps obtenus et les afficher sous forme de courbe dans un tableur. Ce graphique permet de visualiser le temps d'exécution du tri en fonction de la taille de la liste. Que constatez-vous ?

Sur la base de ces mesures, pouvez-vous estimer le temps que prendrait le tri de 100000 éléments ?

Lancer le programme avec 100000 éléments et comparer le temps obtenu avec votre estimation.

Exercice 4 – Tri de Bogo

Coder l'algorithme du tri de Bogo en Python (voir chapitre précédent : Le saviez-vous ?).

Relancer l'algorithme plusieurs fois, en notant le nombre d'itérations nécessaires pour qu'il termine.

A partir de quelle taille de liste cet algorithme est-il inutilisable ?

Exercice 5 – Fibonacci

Ecrire un algorithme qui calcule la suite des nombres de Fibonacci.

Traduire l'algorithme en une fonction Python.

Comparer avec les solutions trouvées par vos camarades de classe.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais lire et appliquer un algorithme, c'est-à-dire que je peux déduire le résultat que me donnera un algorithme à partir d'un jeu de données particulier.
2. Je sais retranscrire un algorithme en un programme, je sais traduire les opérations d'un algorithme en instructions élémentaires if, else, while et for.

1.4 Conclusion

1.4.1 Les algorithmes et nous

Sans le développement de l'algorithmique, de nombreux problèmes n'auraient pas pu eêtre résolus par les ordinateurs dans un temps raisonnable.

L'étude des algorithmes a un effet bénéfique sur notre manière de réfléchir et de résoudre des problèmes dans notre vie quotidienne. L'étude de l'algorithmique permet de structurer notre pensée et de prendre des décisions fondées sur une réflexion argumentée.

Les algorithmes sont omniprésents. « Ils » décident de ce que nous voyons sur les réseaux sociaux, ils influencent nos choix quand nous cherchons une personne qui nous correspond, ils nous suggèrent des livres à lire et des films à regarder, corrigent nos textes, les traduisent ou encore embellissent nos photos en un clic. Ils font la pluie et le beau temps en bourse, décident si un prévenu doit être emprisonné, rédigent des articles de journaux, conduisent des voitures autonomes. Cette liste s'allonge chaque jour…

Comprendre le fonctionnement de base des algorithmes permet de mieux appréhender ce qu'il se passe dans toutes ces situations. Nous y reviendrons plus en détail dans la deuxième partie du cours d'algorithmique.

1.4.2 Focus sur l'automatisation

Grâce aux algorithmes, la machine a pu remplacer l'homme dans de nombreuses tâches (comparez les deux images ci-dessous), en allant des robots constructeurs d'automobiles aux pilotes automatiques dans les avions, ou encore aux logiciels de trading. L'automatisation permet aux employés de se concentrer sur des tâches plus valorisantes et permet aux entreprises de réaliser des économies.

Usine du début du siècle dernier. Les machines dans cette usine de métallurgie à Vallorbe dans le canton de Vaud sont au service des ouvriers. Source : https ://wikivaud.ch/metallurgie-vaudoise/

Usine du début de ce siècle. Les machines dans cette usine de montage Mistubishi en Chine ont remplacé les ouvriers. Source : https ://www.lemonde.fr/blog/fredericjoignot/2015

Selon un rapport publié en 2017 par DELL et le Think Tank californien « L'institut du futur », cité par la chasseuse de têtes Isabelle Rouhan dans sans livre « Les métiers du futur », ***85% des métiers*** qui seront exercés en 2030 par les écoliers d'aujourd'hui n'ont pas encore été inventés.

À retenir

Un algorithme est **une suite d'instructions dans un ordre bien précis** qui permet de résoudre un problème. L'algorithme va produire un résultat en sortie, en fonction de données reçues en entrée.

Pour arriver à résoudre un problème, il faut commencer par le **décomposer en sous-problèmes**.

Afin de pouvoir rechercher de manière efficace, **les données doivent impérativement être triées** en utilisant un algorithme de tri.

Il existe de multiples manières de résoudre un problème, par exemple différents algorithmes de tris. Toutes ces manières ne se valent pas. Il faut **choisir l'algorithme en fonction de ce qui doit être optimisé** : le temps de résolution, l'espace de stockage, la consommation d'énergie, la précision de la solution, etc.

L'algorithme n'est pas un programme. Pour être exécuté sur un système informatique, l'algorithme doit être transcrit en un programme, pour résoudre le problème concrètement et de manière automatisée.

Remarque – Souhaiteriez-vous devenir neuro-manageur.euse ou éducateur.rice de robots ?

Extrait. *Intelligence artificielle. Enquête sur ces technologiques qui changent nos vies. Les algorithmes vont-ils tuer l'emploi ?*, éd. Flammarion, 2008, pp. 72-73.

Résultat, ce ne sont pas seulement les cols-bleus qui sont touchés, mais également les cols blancs, cadres de professions intermédiaires et même supérieures, comptables, traducteurs ou encore traders, etc. Pour l'essayiste Hakim El Karoui, la robotisation pourrait être à ces derniers ce que la mondialisation a été aux premiers. Goldman Sachs, la star des banques d'affaires new-yorkaises, a provoqué une onde de choc dans la profession lorsqu'un de ses responsables a déclaré début 2017 que son *desk* de *trading actions*, qui employait 600 traders à son pic en 2000, n'en comptait plus que deux ! « Ces 600 traders, ils occupaient beaucoup d'espace », a-t-il lâché, sûr de son effet, lors d'un colloque à l'université de Harvard. Des bataillons de traders disparus pour cause de bascule vers le trading électronique à haute fréquence, qui représente aujourd'hui 99% des ordres d'achat et de vente chez Goldman Sachs, 75% chez les concurrents et 45% toutes classes d'actifs confondus dans le secteur bancaire. Ce géant de la finance les a avantageusement remplacés par... 200 ingénieurs. Ce sont eux qui pilotent désormais, pour des salaires jusqu'à cinq fois moins importants, des algorithmes programmés pour gagner des sommes certes infinitésimales, mais sur des millions d'opérations quotidiennes en limitant au maximum le risque. Au suivant ?