1NSI	Algorithmique Les algorithmes	TP/Cours – **Les algorithmes**

Objectifs :

- ⇨ Définir ce qu'est un algorithme
- ⇨ Représenter un algorithme
- ⇨ Connaitre les différents types d'algorithmes
- ⇨ Définir la complexité d'un algorithme
- ⇨ Esquisser les méthodes d'études des algorithmes
- ⇨ Programmer quelques algorithmes simples et classiques

I – Qu'est-ce qu'un algorithme ?

1) Définition

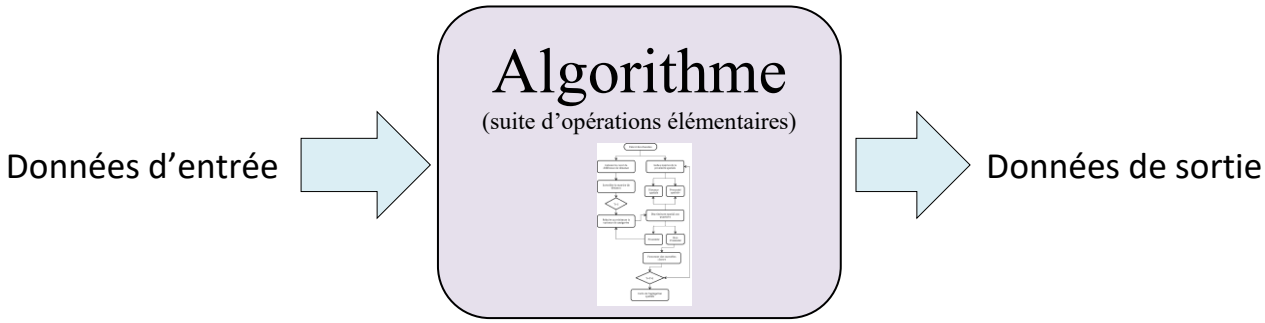

Al Khwarizmi (780 – 850)

Le mot « algorithme » vient du nom du mathématicien perse Al Khwarizmi (780 – 850), qui, au IX° siècle écrivit le premier ouvrage systématique sur la solution des équations linéaires et quadratiques. La notion d'algorithme est donc historiquement liée aux manipulations numériques, mais elle s'est progressivement développée pour porter sur des objets de plus en plus complexes : des textes, des images, des formules logiques, des objets physiques, etc.

Définition

L'algorithmique désigne l'étude des algorithmes.

Un algorithme traite des *données d'entrée* pour fournir des *données de sortie* (la solution du problème) :



Exemples :

2) Catégories d'algorithmes

Il existe plusieurs façons de catégoriser les algorithmes.

Sur la façon de les programmer / exécuter :

- Algorithme ..... ou ..... :
  Un algorithme récursif s'appelle lui-même répétitivement, jusqu'à ce qu'une condition soit remplie. Un algorithme itératif utilise des boucles et ne s'appelle pas lui-même. Jusqu'ici nous n'avons utilisé que des algorithmes itératifs (basés sur des boucles).
- Algorithmes série / parallèles / distribués :
  En général on suppose que les instructions des algorithmes sont exécutés une par une, et on parle d'algorithme en série, mais ils peuvent aussi être parallèles, quand l'architecture de l'ordinateur permet de traiter plusieurs instructions en même temps. Dans ce cas le problème est divisé en sous-problèmes affectés à des processeurs différents. Les algorithmes itératifs sont parallélisables, et surtout les algorithmes de tri. Les algorithmes distribués s'exécutent en parallèle sur des machines différentes.

Selon le type de solution :

- Algorithme ..... :
  On évalue les données en entrée pour donner une réponse de type vrai/faux. Exemple : Y a-t-il un nombre par dans le tableau en entrée ?
- Algorithme ..... :
  On cherche à partir des données en entrée la meilleure solution au problème. Exemple : Meilleur itinéraire pour aller d'un point A à un point B.
- Algorithme ..... :
  On cherche une solution approchée à un problème d'optimisation dont la solution serait trop difficile à trouver en un temps raisonnable.

Selon le paradigme de conception (la méthode générale de résolution) :

- .....
  Le principe "diviser pour régner", de façon récursive réduit un problème à un cas plus simple ou un ensemble de sous-problèmes, jusqu'à atteindre un niveau de simplicité suffisant pour pouvoir le résoudre facilement.
- .....
  Quand la solution optimale d'un problème s'obtient à partir des solutions optimales de sous-problèmes, la programmation dynamique permet d'éviter de recalculer les solutions déjà traitées en mémorisant le résultat.
- .....
  Proche de la programmation dynamique avec cette différence que les solutions des sous-problèmes n'ont pas besoin d'être connues à chaque étape du traitement. Au contraire un choix "glouton" est fait qui consiste à prendre la meilleure décision possible à cet instant.
- .....
  Le problème est exprimé sous forme d'un ensemble d'inégalités linéaires, puis on en fait un traitement mathématique.
- .....
  L'algorithme explore les différents choix possibles, les évalue au fur et à mesure et revient sur ses choix si ceux-ci se révèlent trop mauvais ou mènent à une impasse.
- .....
  Ils font des choix aléatoires.

- .....
  Recherche la solution d'un problème en imitant les processus d'évolution biologiques, avec des générations successibles qui sont des solutions intermédiaires. En fait cela reproduit le principe de survie des meilleurs.
- .....
  Ils passent en revue un par un tous les cas possibles sans hiérarchisation ou optimisation.
- .....
  On réduit le problème a un problème beaucoup plus simple à résoudre en appliquant sur les données un autre algorithme. Par exemple pour la recherche de la médiane d'un tableau, le problème devient très simple si le tableau est trié : il suffit de prendre l'élément du milieu du tableau. On va donc réduire le problème à quelque chose de trivial en appliquant au préalable un algorithme de tri.

3) Description d'un algorithme

On peut décrire un algorithme de plusieurs façons différentes.

a. Par des phrases

Quelques phrases permettent souvent de donner l'idée générale d'un algorithme.

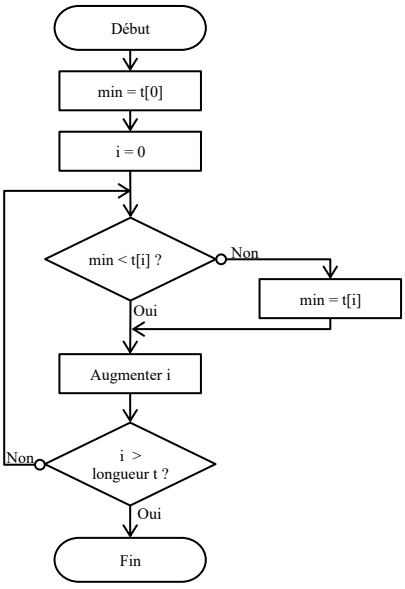
Exemple :
« On note comme minimum le premier élément du tableau, puis on parcourt le tableau en comparant chaque nombre à notre minimum. Si ce nombre est plus petit alors il devient notre nouveau minimum et on passe au suivant. Arrivé à la fin du tableau le dernier minimum noté est le minimum du tableau »

b. Par un algorigramme

Un algorigramme est une représentation graphique d'un algorithme utilisant des symboles normalisés.
Voir https://troumad.developpez.com/C/algorigrammes/ pour davantage de précisions.

Il permet de repérer facilement les itérations d'un algorithme.

Voir l'exemple ci-contre :



c. Pseudo-code

Le pseudo-code est un langage pour exprimer clairement et formellement un algorithme. Ce langage est près d'un langage de programmation comme Python ou javascript, sans être identique à l'un ou à l'autre. Il exprime des idées formelles dans une langue près du langage naturel de ses usagers (pour nous, le français) en lui imposant une forme rigoureuse. Il n'y a toutefois pas de standard normalisé mais seulement des conventions partagées par un plus grand nombre de programmeurs.

Quelques règles :

- Le nom d'une variable ou d'une constante doit être significatif. On devrait savoir immédiatement, à partir de son nom, à quoi sert la variable ou la constante, et quel sens donner à sa valeur.

- Les majuscules et les minuscules sont des symboles distincts dans la plupart des langages de programmation, mais pas tous. Ainsi, pour éviter les ennuis, ne donnez pas à deux entités des noms qui ne différeraient que sur cet aspect
- Les instructions se font une ligne à la fois (pas de ';' en pseudo-code)
- On ne se préoccupe pas des types des variables et des constantes (ce principe n'est pas universel).
- Les opérations de base sont **LIRE, ÉCRIRE et** ← (affectation d'une valeur à une variable). Les opérations de base et/ou mots clés doivent être écrits en gras ou en majuscules.

Plus d'information sur http://info.blaisepascal.fr/pseudo-code#Algorithmique_ecriture_en_pseudo-codes.
Exemple :

```
ALGORITHME : Recherche du minimum d'un tableau
Entrée : tableau de nombres t
Sortie : valeur du plus petit nombre contenu dans le tableau t

DEBUT
min ← t[0]
POUR i VARIANT DE 0 A longueur de t FAIRE :
      SI t[i] < min ALORS :
            min ← t[i]
      FIN SI
FIN POUR
RETOURNER min
FIN
```

II - Analyse d'un algorithme

1) Trace d'exécution

Il existe différentes manières de réaliser une trace de programme et/ou d'algorithme. Une trace :

- permet de **suivre pas à pas** l'algorithme;
- permet de **détecter des erreurs**;
- permet de **contrôler** que l'algorithme fait bien ce que l'on avant prévu;
- permet de **comprendre** ce que fait un algorithme.

Dans la mesure du possible, on peut organiser une trace d'exécution d'un algorithme en constituant un tableau avec toutes les variables de l'algorithme. Il faut numéroter toutes les lignes de l'algorithme. En colonne, il faut indiquer le nom des variables et en ligne les numéros de ligne.

Exemple :

```
1 leplusgrand = 0
2 entree = 0
3 while entree >= 0:
4     entree = int(input("Prochain nombre?"))
5     if entree > leplusgrand:
6           leplusgrand = entree
7
8 print("Nombre le plus grand:",leplusgrand)
```

	Variables		
Ligne	leplusgrand	entree	Commentaire
1	0	/	
2	0	0	
3	0	0	La condition est vraie
4	0	3	L'utilisateur rentre « 3 »
5	0	3	La condition est vraie
6	3	3	
3	3	3	La condition est vraie
.	.	.	.
.	.	.	.
.	.	.	.

2) Complexité

a. Complexité temporelle

La **complexité temporelle** d'un algorithme est une estimation de son temps d'exécution, exprimée comme fonction de la taille de l'entrée. Le temps d'exécution brut n'est pas une grandeur fiable car il dépend de la puissance de la machine sur laquelle on exécute l'algorithme. Pour mesurer la vitesse on compte le nombre d'étapes de calcul avant d'arriver à un résultat. La durée d'exécution d'une étape de calcul dépend de la machine qui l'exécute tandis que le nombre d'étapes à faire ne dépend que de l'algorithme (et éventuellement des données en entrée).
Lorsqu'on étudie un algorithme on essaye souvent d'estimer sa complexité en « nombre d'opérations ». Le « nombre d'opération » dépend du problème et de l'architecture sur laquelle on compte utiliser l'algorithme. On peut compter la complexité par exemple en nombre d'opération mathématique élémentaire ou en nombre de comparaisons ou encore en nombre d'entrées/sorties. Suivant la façon de compter, la complexité peut varier légèrement mais on restera dans la même classe de complexité.

On calcule presque toujours de la complexité *dans le pire des cas*, c'est-à-dire le cas où la valeur des données à traiter entraine le temps de calcul le plus long (ex : tri d'un tableau complétement en désordre au lieu d'un tableau presque déjà trié). On note alors cette complexité dans le pire des cas avec un O(*fonction de n*). Le nombre d'actions à réaliser pour parvenir au bout de l'algorithme est alors proportionnel à ce O().

Liste de complexités en temps classiques

Temps de calcul ↓

Nom	Complexité	Exemple de temps de calcul (nombre d'opérations en fonction de n)	Exemple d'algorithmes
	O(1)	1, 3, 230	Algo simple ne dépendant pas de la taille des données (ex : `x = len(tab)`)
	O(log n)	log n ; log(n^2)	Recherche dichotomique
	O(n)	n ; 4·n ; 0,5·n	Recherche séquentielle, algorithme simple de recherche du plus petit élément d'un tableau
	O(n log n)	n log n, log n!	Tri fusion
	O(n^2)	n^2 ; $2n^2$-1	Tri par insertion
	O(n^3)	n^3	Algorithme naïf de multiplication matricielle.
	O(n^α)	$n^{1,5}$: n^3 ; $3n^6$	Avec $\alpha > 1$
	O(a^n)	$1{,}1^n$; 10^n	Algorithme en force brute, par exemple pour la résolution d'un sudoku.
	O(n!)	3·(n)! ; 2(n+1)!	Problème du voyageur de commerce[1]

On remarque que multiplier par une constante ou rajouter un nombre constant ou de plus petit ordre ne change pas la classe de complexité. Un algorithme en $2n^2 + 1$ sera plus rapide qu'un algorithme en $4n^2 + 2n$ mais ils font tous les deux partie des algorithmes quadratiques et leurs performances seront comparables.

Dans les cas des algorithmes polynomiaux (O(n^2), O(n^3), …) le temps de calcul peut augmenter très vite avec la taille des données. Pour les algorithmes exponentiels, l'augmentation est tellement rapide qu'elle rend impossible les calculs sur des tailles de données vraiment importante ($n > 10^3$).

[1] Le problème du voyageur de commerce est un problème d'optimisation qui, étant donné une liste de villes, et des distances entre toutes les paires de villes, détermine un plus court circuit qui visite chaque ville une et une seule fois.

Pour se donner une idée de l'importance de la complexité dans les temps de calculs, voici un tableau du temps d'exécution en fonction de la complexité d'un algorithme et de la taille des données.
Le temps est estimé pour un ordinateur capable de réaliser 10^{12} opérations par secondes (puissance des tous meilleurs ordinateurs personnels en 2020).

		Complexité						
		1	log(n)	n	n·log(n)	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 ps	6,64 ps	0,1 ns	0,66 ns	10 ns	1 µs	4×10^{10} a
	$n = 10^3$	1 ps	9,96 ps	1 ns	9,96 ns	1 µ s	1 ms	∞
	$n = 10^4$	1 ps	13,28 ps	10 ns	133 ns	0,1 ms	1 s	∞
	$n = 10^5$	1 ps	16,6 ps	0,1 µs	1,6 µs	10 ms	> 16 min	∞
	$n = 10^6$	1 ps	19,93 ps	1 µs	19,9 µs	1 s	> 11 j	∞

∞ = « > 10^{25} années »

Question 1 : Calculs de complexité
Calculer la complexité en nombre d'opérations (mathématiques, affectations, comparaisons) des fonctions ci-dessous.

1)
```
def conversion(n):
    h = n // 3600
    m = (n - 3600*h) // 60
    s = n % 60
    return h,m,s
```

2)
```
def puissanceMoinsUn(n):
    if n%2==0:
        res = 1
    else:
        res = -1
    return res
```

3)
```
def sommeEntiers(n):
    somme = 0
    for i in range(n+1):
        somme += i
    return somme
```

4)
```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

5) (Pour celle-ci faire la complexité en nombre de comparaisons)
```
def triSelection(l):
    for i in range(len(l)-1):
        indMini = i
        for j in range(i+1, len(l)):
            if l[j] < l[indMini]:
                indMini = j
        l[i], l[indMini] = l[indMini], l[i]
```

b. Complexité spatiale

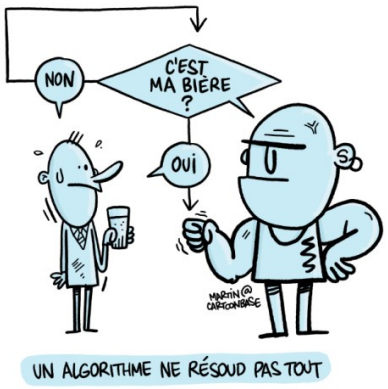
La complexité spatiale étudie la quantité de mémoire dont un algorithme a besoin au cours de son exécution. On ne s'y intéresse qu'assez peu car dans la plupart des cas ce n'est pas un critère pertinent.

3) Terminaison

On doit s'assurer que l'algorithme qu'on programme doit se terminer avec une réponse et de forme pas de boucle infinie.
Pour cela on utiliser différentes techniques comme le **variant de boucle** que nous verrons plus tard plus en détail.

4) Preuve de correction

On souhaite également s'assurer que notre algorithme est **correct**, c'est-à-dire qu'il fournit une réponse valide (par exemple qu'un algorithme de tri renverra toujours un tableau parfaitement trié, qu'un algorithme de recherche trouvera bien la donnée si elle existe, etc…).
Pour ce faire, on ne peut pas se contenter de montrer que « ça marche » avec certaines valeurs. Comme en math il faudra *démontrer* la correction (ou validité) de l'algorithme.
Cela peut se faire notamment avec un **invariant de boucle** : une propriété qui reste toujours vraie du début à la fin d'une boucle de l'algorithme. Nous détaillerons également cela par la suite.



III – Ecriture d'un algorithme

1) Quelques conseils

L'écriture d'un algorithme (à part les cas élémentaires) ne se fait jamais directement dans un langage de programmation. Il y a toujours une étape de conception qui se déroule souvent sur papier.
Voici une technique possible :

- Prendre une feuille et un crayon
- Imaginer la résolution par un humain d'un problème similaire mais avec une très grande quantité de données à traiter (par exemple recherche d'un nom dans une liste d'une centaine de pages).
- Ecrire alors sur sa feuille sous forme de quelques phrases la description de la méthode employée par l'humain.
- Noter sur la feuille l'enchaînement des opérations que réaliserait l'humain résolvant le problème, notamment tout ce qu'il doit mémoriser ou noter (utilisation de variables pour notre algorithme) et toutes les parties qu'il doit répéter (qui formeront les boucles).
- Si besoin prendre un cas exemple et simuler pas à pas la résolution par un humain en consignant bien toutes les étapes puis repérer dans ces étapes les répétitions et les réutilisations de certaines valeurs.
- Passer ensuite à l'écriture en algorigramme ou en pseudo-code.
- Si besoin on peut utiliser la technique des tables d'exécution pour vérifier que l'algorithme imaginé fonctionne comme prévu.
- Transcrire l'algorigramme ou le pseudo-code en langage de programmation.

2) Algorithmes élémentaires au programme de NSI

a. Recherche d'un minimum/maximum dans un tableau

Cet algorithme a déjà été traité de nombreuses fois. Se reporter aux exemples précédents.

b. Recherche séquentielle dans un tableau

On souhaite chercher un élément dans un tableau. L'algorithme reçoit en entrée un tableau et un élément et doit valider sa présence dans le tableau (renvoie VRAI si l'élément est dans le tableau et FAUX s'il n'y est pas).

Question 2 : Recherche séquentielle

1) Ecrire une fonction `recherche_sequentielle` qui prend en argument un tableau `t` et un élément `element` et qui renvoie `True` si `element` est dans le tableau `t` et `False` s'il n'y est pas.

Attention : Il est interdit d'utiliser le mot-clé « in » de python : vous devez programmer l'algorithme avec les éléments de base.

2) **Variante :** Ecrire une fonction `recherche_sequentielle_indice` qui prend en argument un tableau `t` et un élément `element` et qui renvoie l'indice de l'élément où se trouve la première occurrence de l'élément si ce dernier est présent dans le tableau et `False` s'il n'y est pas.
3) **Variante 2 :** Ecrire une fonction `recherche_sequentielle_indices` qui prend en argument un tableau `t` et un élément `element` et qui renvoie la liste des indices où se trouvent l'élément dans le tableau. La liste est une liste vide si l'élément n'est pas dans le tableau.

Remarque : On n'a fait aucune hypothèse sur les données contenues dans le tableau. Celles-ci pourront être de type numérique, texte ou autre sans que cela change l'algorithme.

3) Recherche dichotomique

La recherche précédente peut être grandement accélérée si le tableau est trié.

Question 3 : Recherche dichotomique

1) Imaginer un algorithme de recherche plus efficace du fait que le tableau soit trié (pensez par exemple à une recherche d'un mot dans un dictionnaire) et écrivez la description (quelques phrases) de cet algorithme.
2) Ecrire une fonction `recherche_dichotomique` qui prend en argument un tableau `t` et un élément `element` et qui renvoie `True` si `element` est dans le tableau `t` et `False` s'il n'y est pas.
3) Quelle est la complexité de cet algorithme ?

Quelques ressources à consulter pour approfondir sur la complexité :
https://www.lemonde.fr/blog/binaire/2021/04/16/henri-potier-a-lecole-de-la-complexite/
https://yewtu.be/watch?v=AgtOCNCejQ8&