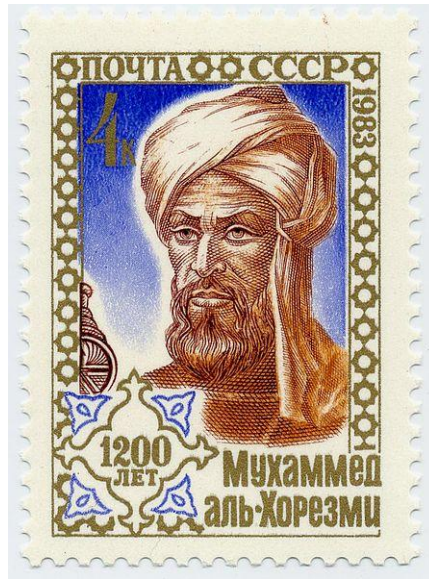


Algorithmes :



Définition

Il existe beaucoup de définitions de ce qu'est un algorithme. En voici une :

« Un algorithme, c'est tout simplement une façon de décrire dans ses moindres détails comment procéder pour faire quelque chose. [...] Le but est d'évacuer la pensée du calcul, afin de le rendre exécutable par une machine numérique (ordinateur...). »

[Gérard Berry](#) (1948-)

En voici une autre :

«Un algorithme est une suite finie et non ambiguë d'instructions et d'opérations permettant de résoudre une classe de problèmes.»

modulo-info.ch

Un algorithme permet de faire quelque chose. Il a des données d'**entrée** et produit une **sortie**.

L'algorithme effectue donc un **traitement** des données d'entrée pour produire une sortie.

Exemple :

On donne parfois comme exemple simpliste d'un algorithme la recette de cuisine en supposant qu'il n'y ait aucune ambiguïté dans son énoncé.

Dans ce cas, les ingrédients constituent l'entrée et le plat cuisiné constitue la sortie. La recette décrit tous les traitements que nous devons effectuer sur les ingrédients pour obtenir le plat.

Terminaison

Si un algorithme ne termine jamais lors de l'exécution, on n'obtient jamais de résultat !

Il est important de savoir si un algorithme va terminer ou pas.

Exemples :

Dans certains cas, la terminaison est évidente :

tant que i est supérieur à 0 :

remplacer i par i+1

va boucler à l'infini si i est positif au départ et ne terminera donc jamais !

Par contre :

tant que i est supérieur à 0 :

remplacer i par i-1

va forcément s'arrêter au bout d'un certain nombre de répétitions de la boucle.

Pour certains algorithmes, on ne peut pas prouver sa terminaison.

Exemple : «[Conjecture de Syracuse](#)»

tant que n est supérieur à 1 :

si n est divisible par 2 :

$n \leftarrow n / 2$

sinon :

$n \leftarrow 3 \cdot n + 2$

Cet algorithme termine pour toutes les valeurs de n qu'on a testées jusqu'à présent mais il est possible que certaines valeurs fassent boucler l'algorithme indéfiniment...

Correction

On dit d'un algorithme qu'il est correct s'il donne la sortie correcte par rapport à l'entrée qu'il reçoit. Il se peut que certaines entrées posent problème et qu'alors, l'algorithme ne termine pas (comme pour la conjecture de Syracuse) : on parle alors de correction partielle.

Si l'algorithme termine toujours et qu'il est correct, on parle de correction totale.

Complexité

Pour un problème donné, il y a souvent plusieurs algorithmes qui peuvent le résoudre. Pour savoir quel est l'algorithme le plus efficace, on peut étudier sa complexité.

La complexité permet d'estimer le temps, l'espace mémoire ou même l'énergie nécessaire pour exécuter un algorithme sur des données d'entrée de taille «n».

La complexité qui nous intéressera particulièrement est celle qui estime le temps nécessaire à l'exécution de l'algorithme sur un ordinateur. On nomme cela la complexité temporelle.

Comme on veut qu'elle soit indépendante du type d'ordinateur utilisé, on ne comptera que le nombre d'instructions nécessaire à la terminaison de l'algorithme. C'est donc un calcul théorique.

Le comptage du nombre d'instructions est un peu fastidieux (voir exemples plus bas) mais on s'intéressera finalement à l'ordre de grandeur et non pas au nombre exact.

C'est pourquoi on parlera de complexité linéaire notée $O(n)$ lorsque le nombre d'instructions est linéaire par rapport à la taille d'entrée. Dans ce cas, si la taille de l'entrée double, le temps double également.

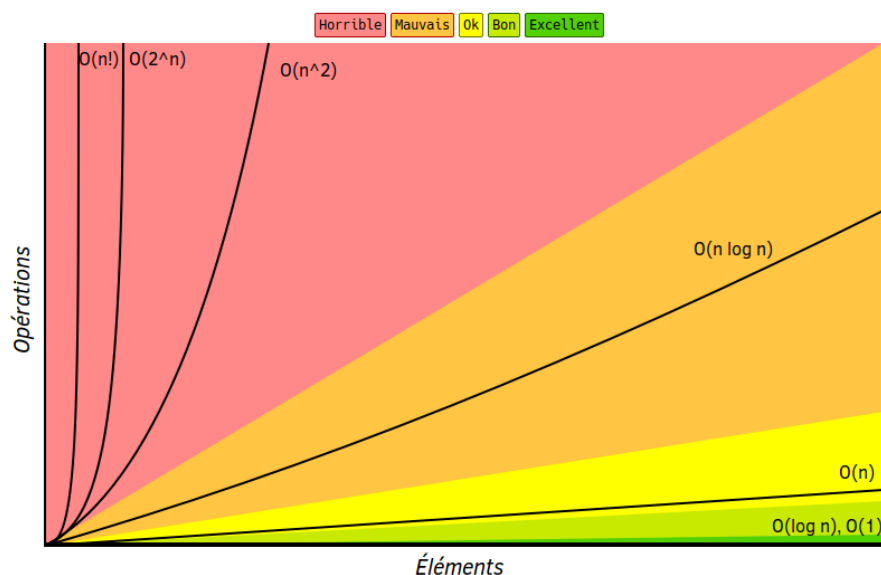
On parlera de complexité quadratique notée $O(n^2)$ lorsque le nombre d'instructions quadruple pour une entrée de taille double.

Il existe encore d'autres complexités comme l'exponentielle notée $O(2^n)$ (horrible) ou la factorielle notée $O(n!)$ (encore pire).

Il existe également la complexité logarithmique notée $O(\log n)$ qui est très favorable voire la complexité constante notée $O(1)$ (qui ne dépend pas du tout de la taille d'entrée!)

Une complexité hybride nommée linéarithmique correspond à $O(n \cdot \log n)$. C'est le mieux qu'on peut faire pour un algorithme de tri (tri par fusion ou par pivot).

Voici un graphique montrant le nombre d'instructions (opérations) nécessaires (sur l'axe vertical) nécessaires pour exécuter l'algorithme sur une entrée de taille n (sur l'axe horizontal).



On y voit bien que la complexité est cruciale pour savoir si un algorithme est utilisable pour des entrées de tailles moyennes ou grandes car les complexités horribles (comme l'exponentielle ou la factorielle) conduisent à des nombres d'instructions si énormes qu'aucun ordinateur au monde n'est capable de résoudre ces problèmes en un temps raisonnable dès que la taille d'entrée est grande.

Exemple : «[le problème du voyageur de commerce](#)» consiste à trouver le chemin le plus court pour passer par n villes différentes. Il faut passer une seule fois dans chaque ville.

Ce problème à une complexité factorielle qui rend sa solution optimale impossible à trouver dès que le nombre de villes dépasse quelques dizaines.

Exemples :

- **Détermination du minimum dans une liste:** (linéaire)
 - Pour trouver le minimum dans une liste, on choisit le premier élément de la liste comme premier candidat puis on parcourt le reste de la liste. Pour chaque élément, on le compare au candidat actuel et s'il lui est inférieur, on écrase la valeur du candidat par cette nouvelle valeur.
 - On remarque facilement qu'il nous faudra une boucle parcourant toute la liste une seule fois. La complexité sera donc linéaire. $O(n)$
- **Tri d'une liste par sélection:** (quadratique)
 - On doit imbriquer deux boucles qui vont parcourir tout ou partie de la liste à trier, raison pour laquelle on obtient une complexité quadratique. $O(n^2)$.
 - La première boucle parcourt tous les éléments pour savoir quelle position (i) de la liste triée on cherche. La deuxième boucle parcourt tous les éléments du reste de la liste (sous-liste non triée) pour en trouver le minimum qui sera ensuite échangé avec l'élément i .
 - Les mêmes raisonnements conduisent également à une complexité quadratique pour les autres algorithmes simples de tri (insertion et bulles).
- **Recherche linéaire d'un nombre dans une liste:** (linéaire)
 - On parcourt toute la liste, élément par élément, pour voir si on trouve le nombre.
 - On boucle sur tous les éléments une seule fois, la complexité est linéaire.
- **Recherche dichotomique dans une liste triée:** (logarithmique)
 - On part du principe que la liste est ordonnée. On commence donc par chercher au milieu de la liste. Si l'élément central de la liste est plus grand que le nombre cherché, on écarte la deuxième moitié de la liste pour se concentrer sur la première moitié (on fait le contraire si l'élément est plus petit). On répète ce processus jusqu'à ce qu'on ait trouvé le nombre ou qu'on ait un seul élément et qu'il ne soit pas égal à notre nombre.
 - Le fait qu'on divise par deux (environ) la taille de la liste à chaque étape rend cette méthode beaucoup plus efficace - $O(\log n)$ - que la recherche linéaire.
 - C'est une stratégie Diviser pour régner. Elle est détaillée plus bas.

Stratégies

Il existe diverses stratégies pour résoudre un problème donné. Nous allons voir ci-dessous 3 familles importantes de stratégies: La "force brute", la méthode "gloutonne" (greedy en anglais) et "Diviser pour régner" (Divide and conquer en anglais)

- **Force brute**

- On essaie toutes les possibilités d'un problème pour en dégager la meilleure solution.
- Exemple: pour le tri, on essaie toutes les possibilités d'arrangement des nombres d'une liste pour chercher ensuite l'une d'entre elles qui soit triée.

- **Glouton**

- Cette stratégie consiste à essayer de trouver un choix optimal à chaque étape de l'algorithme. Ceci rend l'algorithme plus performant car on n'envisage pas toutes les possibilités. Chaque choix élimine en effet tous les autres qu'on a écartés.
- Cette stratégie n'est pas toujours optimale car elle peut mener à une solution non optimale. Voir exemples pour le rendu de monnaie, pour le problème du voyageur de commerce, pour le remplissage de sac à dos...
- En résumé, cette stratégie permet d'être plus rapide car on ne teste pas toutes les possibilités (comme on le ferait en stratégie force brute) mais en revanche, elle peut être non optimale pour certains problèmes.
- Exemple: les algorithmes simples de tri qu'on a envisagés sont, en quelque sorte, des algorithmes gloutons puisqu'à chaque étape, ils améliorent l'ordre global de la liste en faisant des échanges optimaux et non aléatoires.

- **Diviser pour régner**

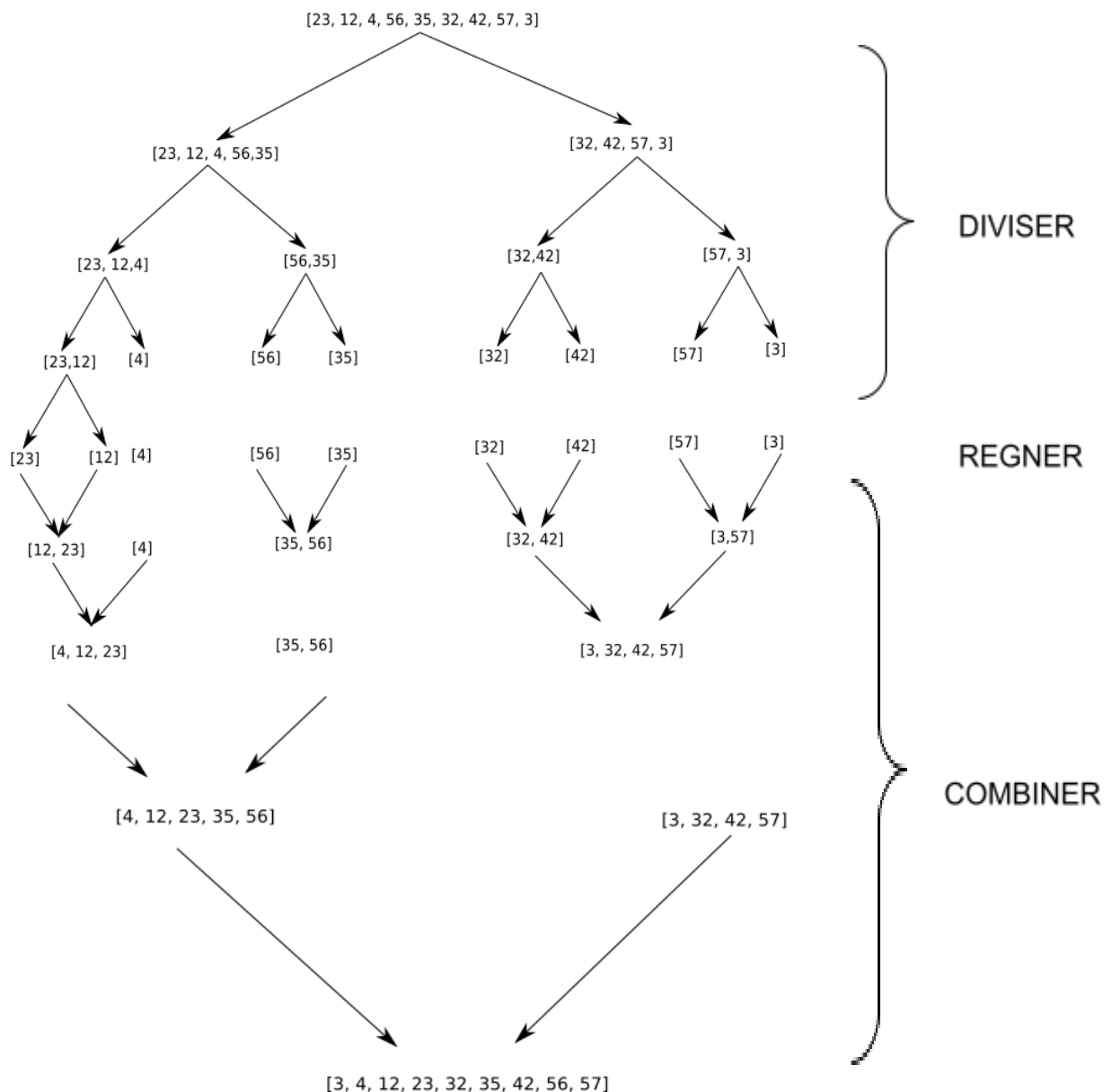
- Cette stratégie consiste à diviser le problème initial en plusieurs sous-problèmes plus simples (de taille plus petite). On résout ensuite ces sous-problèmes, puis on finit par recombinaison des solutions des sous-problèmes pour trouver la solution globale.
- On retrouve toujours les trois phases:
 - Diviser (rendre le problème plus simple)
 - Régner (résoudre le problème pour les problèmes simples)
 - Combiner (combiner les solutions des problèmes simples pour fabriquer la solution globale)
- Exemple: Les algorithmes de "tri par fusion" ou "tri par pivot" (aussi appelé "tri rapide") utilisent la stratégie "Diviser pour régner". Ils sont plus efficaces que les algorithmes simples que nous avons vus auparavant (sélection, insertion ou bulles)

Algorithmes avancés de tri:

1. Tri par fusion

Dans cet algorithme de tri avancé, on commence par diviser la liste que l'on souhaite trier en deux parties de tailles (environ) égales: une sous-liste avec la moitié des éléments de gauche et une autre sous-liste avec la moitié de droite (au pire, il y a une différence de taille d'un élément si le nombre d'éléments d'origine est impair). On divise de nouveau ces deux sous-listes pour en obtenir de plus petites et ainsi de suite jusqu'à obtenir uniquement des sous-listes de taille 1. Elles sont naturellement triées (puisque une liste d'un seul élément ne peut pas être désordonnée). On procède alors à la fusion qui est la partie la plus intéressante de cet algorithme (d'où son nom). On fusionne donc les sous-listes triées deux par deux jusqu'à incorporer tous les éléments de la liste originelle.. Pour chacune de ces fusions, on procède de la manière suivante: on prend chaque fois l'élément le plus petit entre le premier élément de chacune des deux sous-listes qu'on fusionne jusqu'à les incorporer tous. Le fait qu'on n'ait pas besoin de parcourir tous les éléments des sous-listes à fusionner mais seulement le premier de chaque sous-liste est ce qui permet de gagner en efficacité par rapport aux algorithmes simples vu précédemment.

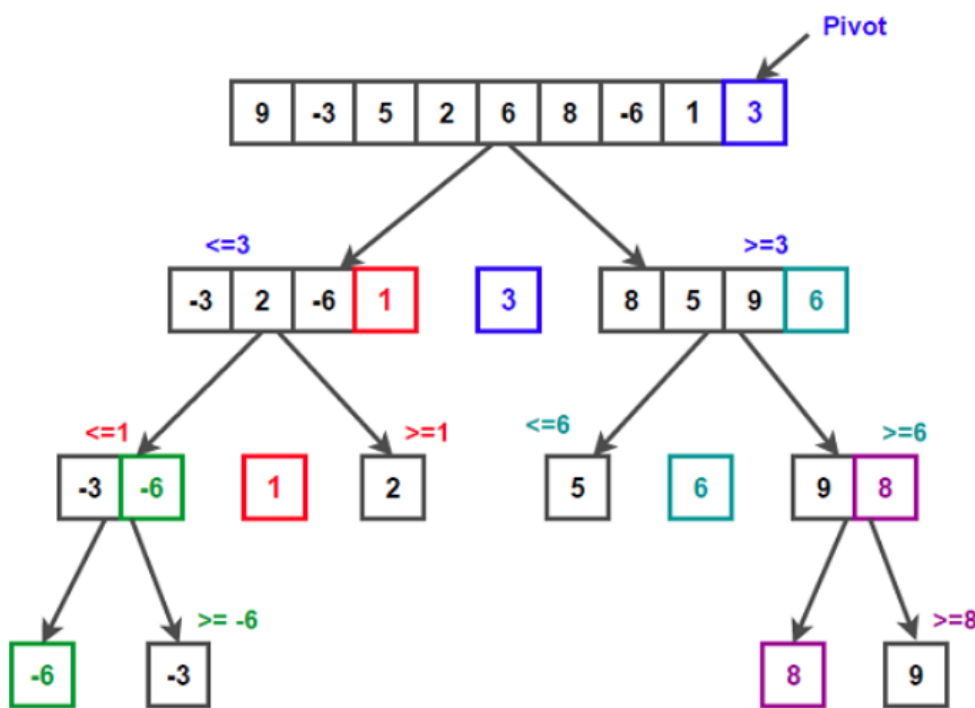
Exemple:



2. Tri par pivot (ou quick sort)

Cet algorithme divise également la liste initiale mais selon un autre procédé. Elle utilise un pivot qui est l'un des éléments de la liste qu'on souhaite diviser. On a plusieurs choix possibles: on peut choisir de toujours prendre le premier élément comme pivot, ou le dernier, ou celui du milieu. On peut même décider de prendre un élément tiré au hasard. Pour l'exemple ci-dessous, on a choisi de prendre toujours le dernier élément comme pivot. Ce pivot va permettre de placer tous les éléments plus petits que le pivot à sa gauche et tous les éléments plus grands que le pivot à sa droite. Dans chaque sous-liste (celle qui est à la gauche du pivot et celle qui est à sa droite), on recommence: on choisit comme pivot le dernier élément puis on déplace les éléments plus petits à leur gauche et les éléments plus grands à leur droite. On recommence ce processus jusqu'à ce qu'il ne reste aucune liste de plus d'un élément. En les recombinaison dans l'ordre où ils apparaissent dans la décomposition, on obtient la liste triée.

Exemple:



La complexité de ces deux algorithmes (tri par fusion et tri par pivots) est **$O(n \log n)$** . C'est une complexité qui se situe entre $O(n)$ et $O(n^2)$. Elle est donc bien meilleure que celle des algorithmes simples ($O(n^2)$). Dès que le nombre d'éléments à trier devient conséquent (des milliers, des millions, voire plus), cet algorithme trie dans des temps raisonnables alors que les algorithmes simples sont inutilisables car beaucoup trop lents.

Ce sont les algorithmes réellement implémentés dans les fonctions de tris des langages de programmation tels que Python.

On peut comparer visuellement ces algorithmes de de tri dans des sites comme

<https://www.toptal.com/developers/sorting-algorithms>

ou

<https://www.sortvisualizer.com>