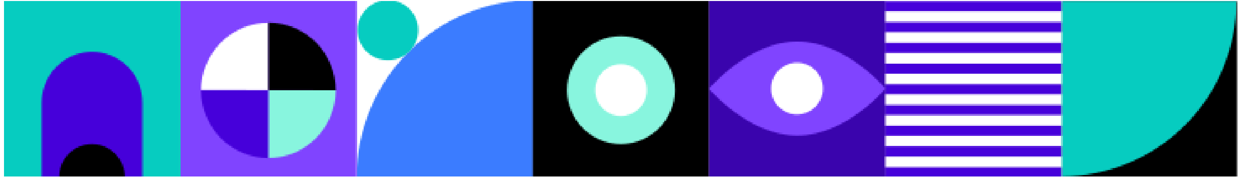


Modulo.

Une introduction à l'informatique

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

17 juillet 2023

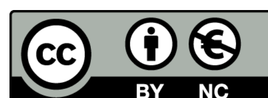


Table des matières

1 Algorithmique II **1**

1.0 Introduction . . . 1
1.0.1 Quoi ? . . . 1
1.0.2 Pourquoi ? . . . 2
1.0.3 Comment ? . . . 2
1.0.4 Objectifs . . . 2
1.1 Terminaison et complexité . . . 3
1.1.1 Principe de terminaison . . . 3
1.1.2 Principe de complexité . . . 5
1.2 Algorithmes de recherche . . . 7
1.2.1 Recherche linéaire . . . 7
1.2.2 Recherche binaire . . . 13
1.2.3 Exercices . . . 19
1.3 Algorithmes heuristiques . . . 21
1.3.1 Complexité exponentielle . . . 21
1.3.2 Exercices . . . 25
1.4 Algorithmes de tri [niveau avancé] . . . 27
1.4.1 Tri par sélection . . . 27
1.4.2 Tri rapide . . . 29
1.4.3 Exercices . . . 34
1.5 Récursivité [niveau avancé] . . . 37
1.5.1 Tri par fusion . . . 37
1.5.2 Focus sur la récursivité . . . 39
1.5.3 Exercices . . . 45
1.6 Conclusion . . . 49

CHAPITRE **1**

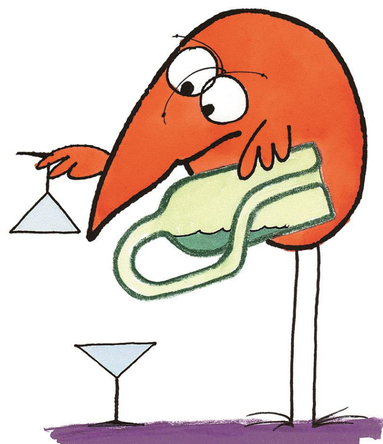
Algorithmique II

1.0 Introduction

1.0.1 Quoi ?

Pour résoudre un problème, il faut commencer par le décomposer en sous-problèmes. Pour chaque sous-problème à résoudre, on décrit les opérations à réaliser sous la forme d'un *algorithme*. Il existe une multitude d'*algorithmes* pour résoudre un problème, mais ils ne se valent pas tous.

L'**algorithmique** étudie les propriétés de ces *algorithmes*. Cette analyse est nécessaire pour nous aider à décider quel *algorithme* utiliser. On se propose à présent de passer en revue quelques propriétés importantes des *algorithmes*.



1.0.2 Pourquoi ?

Si tous les chemins mènent à Rome, on ne peut en emprunter qu'un. Lorsqu'on est face à plusieurs chemins pour arriver au même résultat, il est important de choisir le chemin le plus optimal.

Vous avez déjà rencontré plusieurs algorithmes pour arriver jusqu'ici. Encore plus fort, vous avez rencontré plusieurs algorithmes pour résoudre un même problème, ce qui nous met face à un dilemme : quelle algorithme choisir ? Et y a-t-il une solution à tout problème ?

1.0.3 Comment ?

Dans un premier temps nous allons nous intéresser à la notion de complexité : comment déterminer la vitesse d'un algorithme ? Si plusieurs *bonnes* solutions existent, alors il faut choisir la plus rapide. Mais sera-t-elle toujours la solution la plus rapide ?

Dans un deuxième temps, si vous le souhaitez, vous pouvez ouvrir la porte merveilleuse de la récursivité, à la manière des *Infinity Mirror Room* de Yayoi Kusama.

1.0.4 Objectifs

A la fin de ce chapitre, vous saurez ce qui fait qu'un algorithme est un bon algorithme et quels critères prendre en considération pour choisir le meilleur algorithme. Vous verrez également qu'il existe des problèmes relativement simples que l'on n'arrive toujours pas à résoudre.

— Pouvoir déterminer quelle est la meilleure solution pour un problème donné, en fonction de critères objectifs.
— Comprendre pourquoi certains problème simples n'ont pas de solution exacte.
— [Optionnel] Créer des fonctions récursives, qui s'appellent elles-mêmes.

1.1 Terminaison et complexité

Matière à réfléchir – Compte à rebours

Voici une version naïve du compte à rebours des secondes pour le passage du Nouvel An :

```
# Compte à rebours
def compte_a_rebours(nb_secondes) :
    while True :
        print(nb_secondes)
        nb_secondes = nb_secondes - 1

compte_a_rebours(10)
```

Qu'arrive-t-il lorsqu'on exécute ce *programme* ?

Corriger le programme pour qu'il s'arrête à 0.

Qu'arrive-t-il lorsque l'on exécute la nouvelle version du programme avec la valeur -10 en entrée ou `compte_a_rebours(-10)` ?

1.1.1 Principe de terminaison

La **terminaison** est une propriété essentielle des *algorithmes*, qui garantit que les calculs de l'algorithme finiront par s'arrêter. Lorsque l'on conçoit un algorithme, il est important de faire en sorte que les calculs s'arrêtent à un moment donné. Cette garantie doit tenir pour toutes les entrées possibles. Voici un exemple d'algorithme qui compte et qui ne se termine pas :

```
# Algorithme qui compte infini
Variable i : numérique
i ← nombre donné par l'utilisateur
Tant que i > 0
       i  ← i + 1
        Afficher i
Fin Tant que
```

Si on exécute cet *algorithme*, le *programme* ne s'arrête jamais : `i` est *incrémenté* de `1` indéfiniment. En pratique, si on retranscrit cet algorithme en programme et que l'on exécute le programme, le programme finira par s'arrêter lorsque les nombres représentés seront trop grands pour être représentés.

Exercice 1 – L'infini en programme

Retranscrire l'algorithme infini en programme. Après combien de boucles le programme s'arrête ?

Solution 1 – L’infini en programme

La solution de l’exercice est donnée directement dans le texte qui suit.

Pour faire en sorte que le programme finisse par s’arrêter, nous pouvons le modifier ainsi :

```
# Algorithme qui compte toujours infini
Variable i : numérique
i ← nombre donné par l’utilisateur
Tant que i != 10000
       i  ← i + 1
        Afficher i
Fin Tant que
```

Exercice 2 – L’infini ne finit plus de finir

L’algorithme ci-dessus est appelé «Algorithme qui compte toujours infini». Pourquoi est-il toujours infini ? Dans quel cas cet algorithme ne s’arrête jamais ?

Solution 2 – L’infini ne finit plus de finir

La solution de l’exercice est donnée directement dans le texte qui suit.

Dans la version ci-dessus, si l’utilisateur entre une valeur plus grande que 10000, ou encore une valeur à virgule, l’algorithme ne s’arrête pas. Il peut être implicite pour la personne qui programme qu’un décompte se fait toujours avec des nombres entiers, mais il doit prendre des précautions face aux utilisateurs. Voici une version de l’algorithme de décompte qui s’arrête dans tous les cas :

```
# Algorithme qui compte et qui s’arrête
Variable i : numérique
i ← nombre donné par l’utilisateur
Tant que i < 10000
       i  ← i + 1
        Afficher i
Fin Tant que
```

En programmant, nous devons nous assurer que nos programmes se terminent dans tous les cas, autrement il ne seront pas utilisables. Il ne suffit pas de compter sur la bienveillance des utilisateurs.

Le saviez-vous ? – Conjecture de Syracuse

De nos jours, on ne sait toujours pas si ce programme termine pour chaque entrée n. Ce problème est connu sous le nom la ***conjecture de Collatz*** ou ***la conjecture de Syracuse*** :

```
def Collatz(n) :
    while n > 1 :
        if n % 2 == 0 :
            n = n / 2
        else :
            n = 3 * n + 1
        print(n, '\n')

Collatz(4)
```

1.1.2 Principe de complexité

Matière à réfléchir – Record de vitesse

On souhaite comparer deux algorithmes qui permettent de résoudre le même problème, afin d'utiliser l'algorithme qui permet de résoudre le problème plus rapidement. Mais comment pourrait-on calculer la vitesse d'un algorithme ?

Il est important lorsqu'on utilise un *algorithme* de nous préoccuper de son *efficacité*. Mais comment calculer l'efficacité d'un algorithme, comment calculer sa vitesse ?

Est-ce qu'on peut utiliser la taille de l'algorithme pour prédire le temps qu'il va prendre à s'exécuter ? En d'autres termes, est-ce qu'un algorithme de 10 lignes est toujours plus lent qu'un algorithme de 5 lignes ? Nous avons vu que l'algorithme infini du chapitre précédent est très court (seulement 5 lignes), mais en théorie il ne s'arrête jamais. Une *boucle* rallonge le code de seulement 2 lignes, mais rallonge le temps d'exécution de manière importante.

On pourrait croire qu'il suffit de programmer un algorithme et de chronométrer le temps que ce programme prend à s'exécuter. Cette métrique est problématique, car elle ne permet pas de comparer différents algorithmes entre eux lorsqu'ils sont exécutés sur différentes machines. Un algorithme lent *implémenté* sur une machine dernière génération pourrait prendre moins de temps à s'exécuter qu'un algorithme rapide implémenté sur une machine datant d'une dizaine d'années.

Pour mesurer le temps d'exécution (ou la vitesse) d'un algorithme, il existe un critère plus objectif : le **nombre d'instructions élémentaires**. De manière formelle et rigoureuse, on ne parle pas d'efficacité, mais plutôt de la **complexité d'un algorithme**, qui est en fait contraire à son efficacité. L'analyse de la complexité d'un algorithme étudie la quantité de ressources, par exemple de temps, nécessaires à son exécution.

Le saviez-vous ? – Compliqué

Est-ce que *complexe* veut dire la même chose que *compliqué* ? Une chose compliquée est difficile à saisir ou à faire, alors qu'une chose complexe est composée d'éléments avec de nombreuses interactions imbriquées.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que l'on doit garantir la terminaison d'un algorithme.
2. Je sais que la complexité d'un algorithme peut nous donner une indication de sa vitesse.
3. Je sais que la complexité est une fonction du nombre d'instructions élémentaires.

1.2 Algorithmes de recherche

Les ordinateurs passent la majorité de leur temps à chercher : ils cherchent des fichiers, ils cherchent des sites Web, ils cherchent des informations dans un site Web, ils cherchent votre compte lorsque vous vous loguez sur un site Web, ils cherchent des *posts* à vous montrer et des personnes à qui vous connecter. Nous allons commencer par estimer la complexité de deux algorithmes de recherche importants.

La complexité ne reflète pas la difficulté à implémenter un algorithme, comme on pourrait le croire, mais à quel point l'algorithme se complexifie au fur et à mesure que le nombre des entrées augmente. La complexité mesure le temps d'exécution d'un algorithme (ou sa rapidité) en termes du nombre d'instructions élémentaires exécutées en fonction de la taille des données.

1.2.1 Recherche linéaire

La manière la plus simple pour rechercher un élément dans un tableau (une liste en Python) consiste à parcourir le tableau et à comparer l'élément recherché à tous les éléments du tableau :

```
# Algorithme de recherche linéaire

Tableau Eléments            # données stockées dans un tableau (une liste en Python)
n ← longueur(Eléments)      # la variable n contient le nombre d'éléments
elément_recherché ← entrée  # l'élément recherché est un paramètre de l'algorithme
i ← 1                       # index pour parcourir la liste

Répéter Pour i = 1 à n
    si Eléments[i] == élément_recherché
        Retourner « Oui »
Fin Répéter

Retourner « Non »
```

Quelle est la complexité de cet *algorithme* de **recherche linéaire** ? Pour répondre à cette question, il faut estimer le nombre d'*instructions* élémentaires nécessaires pour rechercher un élément dans un tableau. Cenombre dépend naturellement de la taille du tableau n : plus le nombre d'éléments dans le tableau est grand, plus on a besoin d'instructions pour retrouver un élément.

Supposons que le tableau contienne 10 éléments. Pour trouver l'élément recherché, il faut au moins deux *instructions* par élément du tableau : une instruction pour accéder à l'élément du tableau (ou `Elements[i]`) et une autre instruction pour le comparer à `élément_recherché`. Dans le cas du tableau à 10 éléments, cet algorithme prendrait 20 instructions élémentaires, 2 (instructions) multiplié par le nombre d'éléments dans le tableau. Mais si le tableau contient 100 éléments, le nombre d'instructions élémentaires monte à 200. De manière générale, si le nombre d'éléments dans le tableau est n, cela prend $2n$ instructions pour le parcourir et pour comparer ses éléments.

Cette estimation n'est pas exacte. Nous n'avons pas pris en compte les instructions élémentaires qui permettent d'incrémenter `i` et de vérifier si `i == longueur(Nombres)`. Lorsqu'on prend en compte ces 2 instructions supplémentaires liées à l'utilisation de `i`, le nombre d'instructions passe de 200 à 400 (ce qui correspond à $4n$). Il faut également y ajouter les 4 instructions d'initialisation avant la *boucle*, plus l'instruction de retour à la fin de l'algorithme, ce qui fait monter le nombre d'instructions de 400 à 405

ou $(4n + 5)$. Attention, le nombre exact peut être différent, car il dépend de l'implémentation sur la machine. Mais, ce qui nous intéresse ici n'est pas tant le nombre exact d'instructions, si c'est 205 ou 410 ou 815. Ce qui nous intéresse ici est plutôt l'**ordre de grandeur** de l'algorithme ou comment le nombre d'instructions élémentaires grandit avec la taille du tableau n.

Cet algorithme est de complexité **linéaire**. Une complexité linéaire implique que le nombre d'instructions élémentaires croît linéairement en fonction du nombre d'éléments des données : $cn + a$, ou c et a sont des *constantes*. Dans ce cas précis, c vaut 4 et a vaut 5. Si le tableau contient 10 éléments, il faut environ 45 instructions ; pour 100 éléments il faut environ 405 instructions ; pour 1000 éléments il faut environ 4005 instructions et ainsi de suite. Le nombre d'instructions grandit de manière linéaire en fonction de la taille des données n, et cette relation est représentée par une ligne lorsqu'on la dessine (voir la figure ci-dessous).

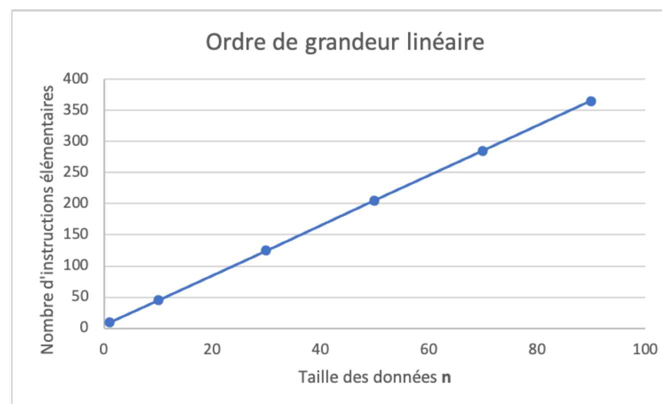


Fig. 1.1 – Complexité linéaire. La complexité de l'algorithme de recherche linéaire, comme son nom l'indique, est linéaire. La relation entre la taille du tableau n et le nombre d'instructions nécessaires pour retrouver un élément dans ce tableau représente une ligne.

Exercice 1 – Compter jusqu'à n

Ecrire un algorithme qui affiche tous les nombres de 1 à n.

Combien d'instructions élémentaires sont nécessaires lorsque n vaut 100 ?

Quelle est la complexité de cet algorithme ?

Solution 1 – Compter jusqu'à n

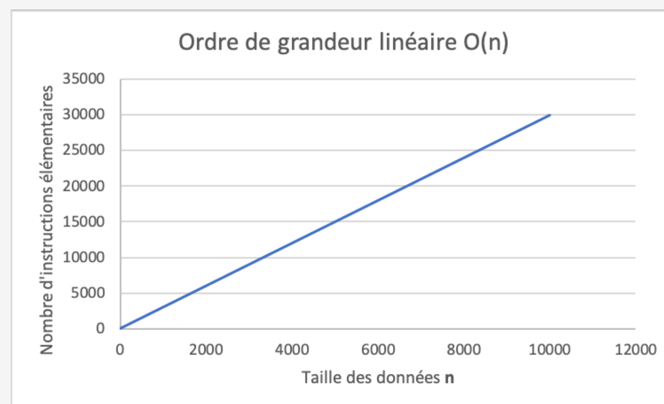
```
Variable n : numérique
Variable i : numérique

Répéter Pour i = 1 à n
    Afficher(i)
Fin Pour
```

L'initialisation des variables `n` et `i` compte pour 2 instructions élémentaires. Chaque passage de la boucle correspond à trois instructions élémentaires : 1 instruction qui affiche `i`, 1 instruction qui incrémente `i` de 1 et finalement une instruction qui compare `i` à `n` (pour savoir si la boucle s'arrête ou si elle continue). Le total d'instructions élémentaires pour le cas où `n` vaut 100 est $3 * 100 + 2$ ou 302 instructions élémentaires.

Il faut se rendre compte que cette estimation du nombre d'instructions élémentaires est approximatif, et non pas exact. Par exemple, l'instruction élémentaire `Afficher(i)` englobe certainement plusieurs instructions à l'exécution et prend de plus en plus de temps à mesure que `i` grandit (affiche de plus en plus de caractères).

La complexité (ou l'ordre de grandeur) de cet algorithme est linéaire, comme illustré dans ce graphique :



Exercice 2 – Compter par pas de 2

Ecrire un algorithme qui affiche tous les nombres *pairs* de 1 à n.

Combien d'instructions élémentaires sont nécessaires lorsque n vaut 100 ?

Quelle est la complexité de cet algorithme ?

Solution 2 – Compter par pas de 2

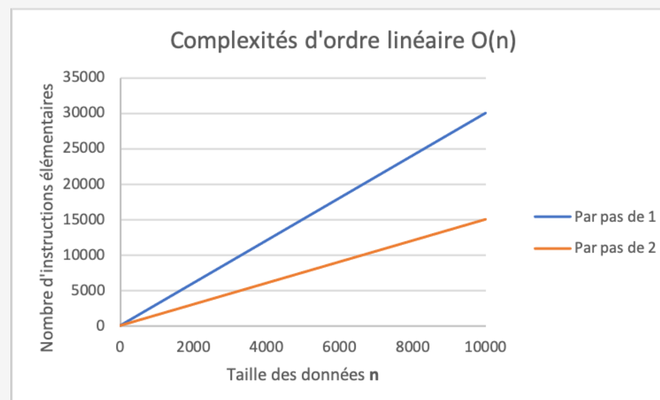
```
Variable n : numérique
Variable i : numérique

Répéter Pour i = 2 à n, par pas de 2
    Afficher(i)
Fin Pour
```

La seule ligne qui change par rapport à la solution de l'exercice précédent est l'incrément de la boucle par pas de 2.

L'initialisation des variables `i` et `n` compte pour 2 instructions élémentaires. Chaque passage de la boucle correspond à trois instructions élémentaires : 1 instruction qui affiche `i`, 1 instruction qui incrémente `i` de 2 et finalement 1 instruction qui compare `i` à `n` (pour savoir si la boucle s'arrête ou si elle continue). Pour le cas où `n` vaut 100, la boucle sera parcourue 50 fois (par pas de 2). Le total d'instructions élémentaires est donc $3 * 50 + 2$ ou 152 instructions élémentaires.

La complexité (ou l'ordre de grandeur) de cet algorithme est également linéaire, comme illustré dans le graphique. Il faut noter que l'ordre de grandeur est le même que pour l'exercice précédent, seule la vitesse de croissance change.



La différence de croissance se cache dans la constante c de l'ordre de grandeur $cn + a$. La valeur de c dans l'exercice précédent est supérieure à la valeur de c dans cet exercice. Dans un premier temps, on peut ignorer la valeur de cette constante c. Cependant elle peut devenir importante lorsque l'on doit comparer des algorithmes du même ordre entre eux.

Exercice 3 – Recherche linéaire

Programmer l'algorithme de recherche linéaire en Python. Rechercher une valeur entre 1 et 1000000 dans un tableau qui contient les valeurs allant de 1 à 1000000.

Pour quelle valeur l'algorithme est le plus rapide ? Le plus lent ? Indice : utiliser le module `time` pour chronometrer le temps d'exécution.

Solution 3 – Recherche linéaire

Voici un programme possible pour l'algorithme de recherche linéaire. Si on chronomètre le temps d'exécution, la valeur 1 est trouvée très rapidement en comparaison à la valeur 1000000. C'est beaucoup plus lent de rechercher une valeur qui se trouve à la fin du tableau, qu'au début du tableau.

```
# algorithme de recherche linéaire
def search_lin(search_list, search_element, verbose=0) :

    # boucle pour parcourir la liste
    for element in search_list :
```

```python
        if verbose :
            print("L'élément comparé est : " + str(element) + "\n")

        # l'élément de la liste correspond à l'élément recherché
        if element == search_element :
            return True

    # aucun élément ne correspond
    return False


import time

last = 1000000
ma_liste = list(range(1,last+1))

# mettre verbose à 1 pour avoir une vue de ce qui se passe
# attention, cela fausse les temps de calcul (plus temps d'affichage)
verbose = 0

# chronomètre le temps de recherche de l'élément 1000000
time_start = time.time()
search_lin(ma_liste, last, verbose)
time_1000000 = time.time() - time_start
print("Recherche linéaire 1000000 : " + str(round(time_1000000,7)) + "
    secondes.\n")

# chronomètre le temps de recherche de l'élément 1
time_start = time.time()
search_lin(ma_liste, 1, verbose)
time_1 = time.time() - time_start
print("Recherche linéaire 1 : " + str(round(time_1,7)) + " secondes.")
```

Exercice 4 – Recherche linéaire (dans le désordre)

On a vu dans l'exercice précédent que cela prend moins de temps pour trouver un élément au début de la liste qu'un élément de la fin de la liste. Qu'est-ce qui arrive si les éléments de la liste ne sont pas dans l'ordre ? Essayer en utilisant la fonction `shuffle()` du module `random`.

Solution 4 – Recherche linéaire (dans le désordre)

Si on mélange l'ordre des éléments, la valeur 1 pourrait se retrouver en fin de tableau et pourrait prendre très longtemps à retrouver. Au contraire, la valeur 1000000 pourrait se retrouver en début de tableau et pourrait prendre très peu de temps à retrouver. La situation est donc beaucoup moins prévisible. Le code ci-dessous est à utiliser avec la fonction `search_lin()` de la solution précédente.

```
import time
import random

last = 1000000
ma_liste = list(range(1,last+1))
# mélange les éléments du tableau au hasard
random.shuffle(ma_liste);

# mettre verbose à 1 pour avoir une vue de ce qui se passe
# attention, cela fausse les temps de calcul (plus temps d'affichage)
verbose = 0

# chronomètre le temps de recherche de l'élément 1000000
time_start = time.time()
search_lin(ma_liste, last, verbose)
time_1000000 = time.time() - time_start
print("Recherche linéaire 1000000 : " + str(round(time_1000000,7)) + "
    secondes.\n")

# chronomètre le temps de recherche de l'élément 1
time_start = time.time()
search_lin(ma_liste, 1, verbose)
time_1 = time.time() - time_start
print("Recherche linéaire 1 : " + str(round(time_1,7)) + " secondes.")
```

Pour aller plus loin

Pour décrire mathématiquement les ordres de complexité, on utilise la notation « Grand O ». Pour dire qu'un ordre de complexité est linéaire, on écrit par exemple qu'il est $O(n)$.

1.2.2 Recherche binaire

Matière à réfléchir – Le dictionnaire

Comment faites-vous pour rechercher un mot dans un dictionnaire ?

Utilisez-vous l'algorithme de recherche linéaire, parcourez-vous tous les éléments un à un ?

Quelle propriété du dictionnaire nous permet d'utiliser un autre algorithme de recherche que l'algorithme de recherche linéaire ?

Si on doit rechercher un élément dans un tableau qui est **déjà trié**, l'*algorithme* de recherche linéaire n'est pas optimal. Dans le cas de la recherche d'un mot dans un dictionnaire, la recherche linéaire implique que l'on va parcourir tous les mots du dictionnaire dans l'ordre, jusqu'à trouver le mot recherché. Mais nous ne recherchons pas les mots dans un dictionnaire de la sorte. Nous exploitons le fait que les mots du dictionnaire sont triés dans un ordre alphabétique. On commence par ouvrir le dictionnaire sur une page au hasard et on regarde si le mot recherché se trouve avant ou après cette page. On ouvre ensuite une autre page au hasard dans la partie retenue du dictionnaire. On appelle cette méthode la **recherche binaire** (ou la recherche dichotomique), car à chaque itération elle ***divise l'espace de recherche en deux***. En effet, à chaque nouvelle page ouverte, nous éliminons environ la moitié du dictionnaire qui nous restait à parcourir. Voici une description de l'algorithme de recherche binaire :

```
# Algorithme de recherche binaire

Tableau Eléments             # les données du tableau (liste en Python) sont triées
n ← longueur(Eléments)       # la variable n contient le nombre d'éléments
elément_recherché ← entrée   # l'élément recherché est un paramètre de l'algorithme
trouvé ← Faux                # indique si l'élément à été retrouvé
index_début ← 0              # au début on cherche dans tout le tableau
index_fin ← n                # au début on cherche dans tout le tableau

# tant que l'élément n'est pas trouvé et que le sous-tableau retenu n'est pas vide
Tant que trouvé != Vrai et n > 0 :

    # on identifie le milieu du sous-tableau retenu
    index_milieu = (index_fin - index_début)/2 + index_début

    # si l'élément recherché correspond à l'élément du milieu du sous-tableau retenu
    if Eléments[index_milieu] == elément_recherché :
        trouvé = Vrai
    else :

        # si l'élément du milieu est plus grand que l'élément recherché
        # on retient la première moitié comme sous-tableau
        if Eléments[index_milieu] > elément_recherché :
            index_fin = index_milieu

        # si l'élément du milieu est plus petit que l'élément recherché
```

(suite sur la page suivante)

(suite de la page précédente)

```
        # on retient la deuxième moitié comme sous-tableau
        else :
            index_début = index_milieu+1

    # calcule le nombre d'éléments du sous-tableau retenu
    n = index_fin - index_début

Fin Tant que

Retourner trouvé
```

Prenons le temps d'étudier cet *algorithme*. Que fait-il ? La *boucle Tant que* permet de parcourir le tableau `Eléments` et d'y rechercher `elément_recherché` tant qu'il n'est pas trouvé (tant que `trouvé != Vrai`). A la première itération (au premier passage dans la *boucle*, on vérifie si l'élément au milieu du tableau `Eléments` n'est justement pas l'élément recherché. Les éléments de la liste étant triés, si l'élément au milieu du tableau est plus grand que l'élément recherché, cela indique que `elément_recherché` se trouve dans la première partie du tableau. Si l'élément au milieu du tableau est plus petit que l'élément recherché, cela indique que l'élément recherché se trouve au contraire dans la deuxième moitié du tableau. Pour la suite de la recherche, on retient soit la première, soit la deuxième moitié du tableau, selon si l'élément recherché est plus grand ou plus petit que l'élément du milieu. A chaque prochaine itération (à chaque passage dans la *boucle*), on vérifie si l'élément au milieu du nouveau sous-tableau retenu n'est pas l'élément recherché.

Dans la **recherche linéaire**, chaque passage de la *boucle* permettait de comparer un élément à l'élément recherché et l'espace de recherche diminuait seulement de 1 (l'espace de recherche correspond aux nombre d'emplacements encore possibles). Dans la **recherche binaire**, chaque passage de la *boucle* divise l'espace de recherche par deux et nous n'avons besoin de parcourir plus qu'une moitié (de la moitié, de la moitié, de la moitié, etc.) du tableau.

Le nombre d'étapes nécessaires pour rechercher un élément dans un tableau de taille 16 de façon dichotomique correspond donc au nombre de fois que le tableau peut être divisé en 2 et correspond à 4 (comme on peut le voir sur la figure ci-dessous), parce que :

$16/2/2/2/2 = 1$ donc

$2 * 2 * 2 * 2 = 16$ ou

$2^4 = 16$

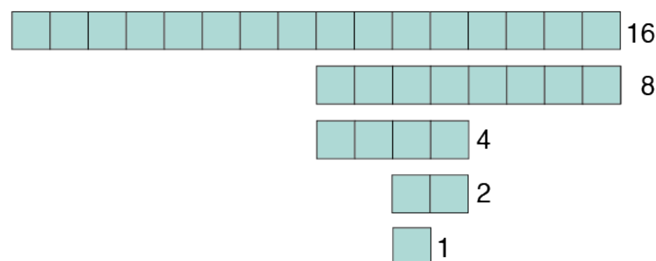


FIG. 1.2 – Exemple de parcours d'un tableau trié. Seulement 4 étapes sont suffisantes pour parcourir un tableau trié de 16 éléments à la recherche d'un élément qui se trouve à la onzième position. A chaque étape, l'espace de recherche est divisé par 2. Si le tableau n'était pas trié, 11 étapes seraient nécessaires (on doit vérifier chaque élément dans l'ordre).

Si on généralise, le nombre d'étapes x nécessaires pour parcourir un tableau de taille n est :

2^x = n par conséquent

$x = log_2(n)\ log(n)$ la simplification peut être faite car l'ordre de grandeur est le même

La complexité de l'algorithme de recherche binaire est donc **logarithmique**, lorsque n grandit nous avons besoin de $log(n)$ opérations. La figure ci-dessous permet de comparer les ordres de grandeur logarithmique et linéaire. On remarque qu'un algorithme de complexité logarithmique est beaucoup plus rapide qu'un algorithme de complexité linéaire, car il a besoin de beaucoup moins d'instructions élémentaires pour trouver une solution.

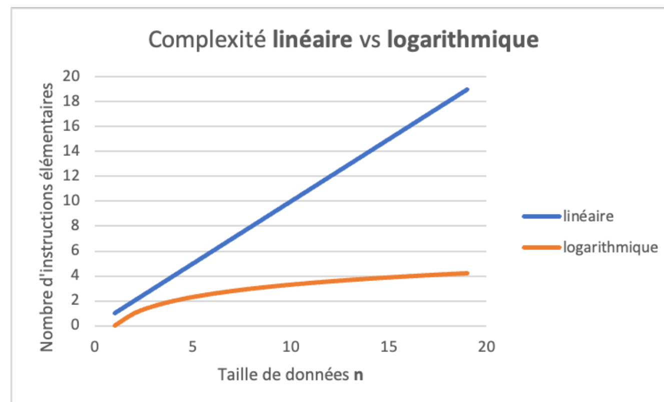


FIG. 1.3 – Comparaison de complexités logarithmique et linéaire. Un algorithme de complexité logarithmique est plus rapide qu'un algorithme de complexité linéaire.

L'algorithme de la recherche binaire se base sur une stratégie algorithmique de résolution de problèmes très efficace, que l'on appelle la stratégie « **diviser pour régner** ». Cette stratégie qui consiste à :

Diviser : décomposer le problème initial en sous-problèmes ;

Régner : résoudre les sous-problèmes ;

Combiner : calculer une solution au problème initial à partir des solutions des sous-problèmes.

Les sous-problèmes étant plus petits, ils sont plus faciles et donc plus rapides à résoudre. Les algorithmes de ce type en plus d'être efficaces, se prêtent à la résolution en parallèle (par exemple sur des multiprocesseurs).

Exercice 5 – Recherche binaire

Programmer l'algorithme de recherche binaire en Python. Rechercher une valeur entre 0 et 100 dans un tableau qui contient les valeurs allant de 1 à 99.

Pour quelle valeur l'algorithme est le plus rapide ? Pour quelle valeur l'algorithme est le plus lent ? Indice : mettre un mode verbose pour afficher ce que fait l'algorithme.

Solution 5 – Recherche binaire

Voici un programme possible pour l'algorithme de recherche binaire. L'algorithme est le plus rapide si on recherche la valeur qui se trouve au milieu du tableau (la valeur 50) et il est le plus lent lorsque l'on recherche la première ou dernière valeur du tableau (la valeur 1 ou 99).

```python
# algorithme de recherche binaire
def search_bin(search_list, search_element, verbose = 0) :
    # détermine les limites de la liste considérée
    start = 0
    end = len(search_list)
    # tant que la liste
    while end-start :
        # accède l'élément du milieu, division entière, il faut un index
        middle = (end-start) // 2 + start
        if verbose :
            print("Elément comparé : " + str(search_list[middle]) +"\n")
        # compare l'élément au milieu de la liste
        if search_element == search_list[middle] :
            if verbose :
                print("Elément retrouvé !\n")
            return True
        # l'élément est plus petit que l'élément du milieu
        elif search_element < search_list[middle] :
            # search_list devient dans la première moitié de search_list
            end = middle
            if verbose :
                print("1ère moitié de la liste : " + str(search_list[start:end])
                     + "\n")
        # l'élément est plus grand que l'élément du milieu
        else :
            # search_list devient la deuxième moitié de search_list
            start = middle + 1
            if verbose :
                print("2ème moitié de la liste : " + str(search_list[start:end])
                     + "\n")
    # aucun élément ne correspond
    if verbose :
        print("L'élément " + str(search_list[middle]) + " n'a pas été retrouvé
             ...\n")
    return False

ma_liste = list(range(1,100))
mon_element = 25
# recherche de l'élément mon_element
search_bin(ma_liste, mon_element, 1)
```

Exercice 6 – Recherche binaire (dans le désordre)

Est-ce qu'on peut utiliser l'algorithme de recherche binaire si le tableau n'est pas trié ? Essayer avec la fonction `shuffle()` du module `random`.

Solution 6 – Recherche binaire (dans le désordre)

Si le tableau n'est pas trié, l'algorithme n'est pas garanti de trouver l'élément recherché, car il peut facilement passer à côté. Le code ci-dessous est à utiliser avec la fonction `search_bin()` donnée dans la solution précédente.

```
import random

last = 99
ma_liste = list(range(1,last+1))

mon_element = random.randint(1,last)
print("L'élément recherché est : " + str(mon_element) + "\n")

# recherche de l'élément mon_element
search_bin(ma_liste, mon_element, 1)

random.shuffle(ma_liste)
print("Tableau mélangé... : " + str(ma_liste) + "\n")
search_bin(ma_liste, mon_element, 1)
```

Exercice 7 – Recherche linéaire versus binaire

Reprendre les programmes de recherche linéaire et recherche binaire en Python et les utiliser pour rechercher un élément dans un tableau à 100 éléments : quel algorithme est le plus rapide ?

Augmenter la taille du tableau à 1000, 10000, 100000, 1000000 et 10000000. Rechercher un élément avec vos deux programmes. Quel algorithme est plus rapide ? Est-ce significatif ?

Est-ce que **un million** vous semble être un grand nombre pour une taille de données ?

Solution 7 – Recherche linéaire versus binaire

Comme prévu par les estimations de complexité, avec sa complexité logarithmique, c'est l'algorithme de la recherche binaire qui est plus rapide. Le gain de temps devient de plus en plus important au fur et à mesure que le nombre d'éléments dans le tableau grandit. Pour cent éléments, la recherche binaire est environ ***2 fois*** plus rapide que la recherche linéaire, alors que pour un million d'éléments, elle est plus de ***1000 fois*** plus rapide.

On peut remarquer que le temps pris par la recherche binaire change peu avec la taille du tableau, ce qui n'est pas le cas de la recherche linéaire. Il faut environ ***10 secondes*** pour trier un million d'éléments avec l'algorithme linéaire, alors que moins de ***10 millisecondes*** suffisent à l'algorithme binaire. Les systèmes actuels traitent des données bien plus volumineuses qu'un million, pensez à toutes les vidéos sur le Web ou tous les utilisateurs d'un réseau social. Tout serait très lent, trop lent si on n'avait pas pensé à diviser pour régner.

```
import time
import random

# stocke les résultats
resultat_lin, resultat_bin = [], []
# longueurs des listes
nb = [100, 1000, 10000, 100000, 1000000, 1000000]

# pour toute les longueurs des listes
for last in nb :

    # créer la liste
    ma_liste = list(range(1,last+1))
    # rechercher un élément au hasard
    mon_element = random.randint(1,last)
    print("L'élément recherché est : " + str(mon_element))
    # recherche linéaire
    time_1 = time.time()
    search_lin(ma_liste, mon_element, 0)
    time_algo_lin = round(time.time() - time_1, 7)
    resultat_lin.append(time_algo_lin)
    # recherche binaire
    time_1 = time.time()
    search_bin(ma_liste, mon_element, 0)
    # des fois, c'est tellement rapide que le temps pris vaut 0
    time_algo_bin = round(max(time.time() - time_1, 0.000001), 7)
    resultat_bin.append(time_algo_bin)

print("Linéaire (secondes) : " + str(resultat_lin))
print("Binaire (secondes) : " + str(resultat_bin))
```

Le saviez-vous ? – Espace-temps et énergie

Nous allons surtout étudier la complexité des algorithmes en rapport avec le temps. Mais la complexité d'un algorithme peut également être calculée en rapport avec toutes les ressources qu'il utilise, par exemple des ressources d'**espace en mémoire** ou de **consommation d'énergie**.

1.2.3 Exercices

Exercice 8 – Recherche binaire aléatoire

Modifier votre programme de recherche binaire : au lieu de diviser l'espace de recherche exactement au milieu, le diviser au hasard. Cette recherche avec une composante aléatoire s'apparente plus à la recherche que l'on fait lorsque l'on cherche un mot dans le dictionnaire.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que la complexité temporelle donne une indication sur la vitesse d'un algorithme, en mesurant le nombre d'instructions élémentaires.
2. Je sais qu'un algorithme de complexité linéaire est plus lent qu'un algorithme de complexité logarithmique.
3. Je peux utiliser la stratégie « diviser pour régner » pour rechercher rapidement avec l'algorithme de recherche binaire.

1.3 Algorithmes heuristiques

Matière à réfléchir – Tour du monde

Vous avez décidé de faire le tour du monde. Choisissez cinq pays que vous souhaitez visiter et placez-les sur une carte. Essayez de trouver le meilleur itinéraire pour visiter ces cinq pays.

Quels critères avez-vous pris en compte pour décider du meilleur itinéraire ? Avez-vous essayé de trouver la plus petite distance à parcourir ?

Vous avez décidé de visiter dix pays. Est-ce qu'il est aussi facile de trouver un itinéraire optimal ?

Imaginez que vous souhaitez visiter tous les pays du monde (un peu moins de 200). Combien y a-t-il d'itinéraires possibles ? Comment s'appelle ce nombre ?

Si le calcul d'un itinéraire prenait 1 milliseconde, combien de temps faudrait-il pour trouver la meilleure solution en énumérant toutes les solutions possibles ? Pour comparaison, le nombre d'atomes dans l'univers est d'ordre 10^{80}.

1.3.1 Complexité exponentielle

Il existe des problèmes difficiles à résoudre. Nous allons nous pencher sur un problème qui s'appelle le **problème du sac à dos**. Prenons un sac à dos et une multitude d'objets qui ont chacun un poids. Notre objectif est de choisir les objets à mettre dans le sac à dos pour le remplir au maximum, mais sans dépasser sa capacité. Donc la question que l'on se pose est la suivante : quels objets devrions-nous emporter, sans dépasser le poids maximal que le sac à dos peut contenir ?

Exercice 1 – Le problème du sac à dos

Comment procéderiez-vous pour résoudre ce problème du sac à dos ? Prenez le temps d'imaginer un *algorithme* qui puisse résoudre ce problème ?

Appliquer cet algorithme pour 4 objets de poids 1, 3, 5 et 7 kg et un sac de capacité de 10 kg.

Est-ce que votre algorithme donne toujours la meilleure solution ?

Solution 1 – Le problème du sac à dos

La solution est donnée dans le texte qui suit.

L'algorithme le plus simple pour résoudre ce problème est un **algorithme de force brute** (ou un algorithme exhaustif), qui consiste à énumérer toutes les combinaisons d'objets que pourrait contenir le sac à dos, l'une après l'autre, et de calculer le poids total pour chaque combinaison. Après avoir calculé toutes les combinaisons, il suffit de sélectionner la combinaison dont le poids se rapproche le plus de la

capacité du sac à dos, sans la dépasser. Vous trouverez ci-dessous la solution pour l'exemple de l'exercice précédent (« oui » signifie que l'on met l'objet dans le sac à dos et « non » signifie que l'on ne met pas l'objet dans le sac à dos).

+ Combinaison	$1kg$	$3kg$	$5kg$	$7kg$	Poids total +
1	non	non	non	non	$0kg$
2	oui	non	non	non	$1kg$
3	non	oui	non	non	$3kg$
4	oui	oui	non	non	$4kg$
5	non	non	oui	non	$5kg$
6	oui	non	oui	non	$6kg$
7	non	oui	oui	non	$8kg$
8	oui	oui	oui	non	$9kg$
9	non	non	non	oui	$7kg$
10	oui	non	non	oui	$8kg$
11	non	oui	non	oui	$10kg$
12	oui	oui	non	oui	$11kg$
13	non	non	oui	oui	$12kg$
14	oui	non	oui	oui	$13kg$
15	non	oui	oui	oui	$15kg$
16	oui	oui	oui	oui	$16kg$

La meilleure solution se trouve à la 11ème ligne, la capacité du sac à dos (10 kg) est atteinte lorsqu'on y met le deuxième et le quatrième objet.

Exercice 2 – Le problème du sac à dos avec 10 objets

Combien de combinaisons possibles existent pour le problème du sac à dos avec 10 objets ?

Solution 2 – Le problème du sac à dos avec 10 objets

La solution est donnée dans le texte qui suit.

Mais, combien y a-t-il de combinaisons possibles si on a 10 objets ? Pour chaque objet, on a deux choix possibles : le mettre dans le sac à dos ou ne pas le mettre dans le sac à dos ***(to take or not to take, that is the question)***. Comme ces deux possibilités existent pour chacun des 10 objets, le nombre de combinaisons possibles vaut :

$$2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 2^{10}$$

Pour n objets, le nombre de solutions possibles est 2^n. Si on a 2 objets, il y a donc 4 combinaisons différentes d'objets dans le sac à dos (aucun objet, le premier objet, le deuxième objet et les deux objets ensemble). Pour 3 objets, le nombre de combinaisons est 8. Pour 5 objets, nous avons 32 possibilités à explorer. Mais déjà pour 10 objets, ce nombre dépasse les 1000 combinaisons possibles. Pour 100 objets, ce nombre devient prohibitif et vaut 10^{30}. Si on doit résoudre ce problème avec 270 objets sous la main, le nombre de

combinaisons possibles dépasse le nombre d'atomes dans l'univers, c'est-à-dire 10^{80}. Si le calcul du poids d'une combinaison prenait une microseconde, il nous faudrait pour résoudre ce problème bien plus que le temps de l'existence de l'univers, plus de 14 milliards d'années. Ces nombres sont réellement vertigineux. Cela va de soi, nous n'avons pas tout ce temps à disposition…

L'ordre de complexité de type 2^n est un ordre de **complexité exponentielle**. Cela vaut aussi pour d'autres constantes que 2, par exemple 10^n ou 1.1^n. Lorsqu'un algorithme est d'ordre de complexité exponentielle, cela veut dire que le temps nécessaire pour résoudre le problème croît exponentiellement en fonction de la taille des données n (voir figure ci-dessous). Les problèmes de complexité exponentielle ne peuvent être résolus dans un temps raisonnable, pour des données à partir d'une certaine taille.

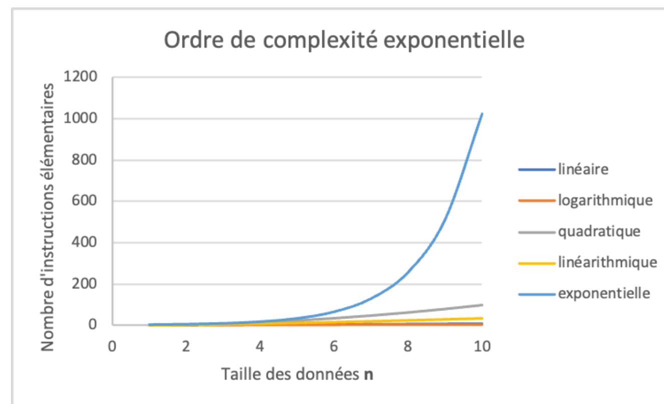


Fig. 1.4 – Complexité exponentielle. Comparaison de l'ordre de complexité exponentielle avec les ordres de complexité vus jusqu'ici. Dans un ordre de complexité exponentielle, le nombre d'instructions élémentaires grandit très rapidement avec la taille des données, et l'algorithme est très lent.

Lorsqu'il est trop difficile de trouver une solution exacte à un problème, nous ne devons pas nous avouer vaincus. Dans ce cas, nous pouvons tout de même rechercher une solution inexacte, mais qui se rapproche autant que possible de la solution optimale. Les algorithmes qui aboutissent à des solutions non optimales ou inexactes, sont appelés des **heuristiques**.

Un algorithme heuristique pour le problème du sac à dos pourrait être l'algorithme suivant : prendre les objets du plus petit au plus grand poids jusqu'à remplir le sac à dos, ce qui nous permettrait de mettre le plus d'objets possible. En suivant cet algorithme heuristique, dans l'exemple du premier exercice, on prendrait les trois premiers objets et on aurait un sac à dos rempli à 9 kg au lieu des 10 kg de capacité maximale du sac à dos. Cette solution est suffisamment proche de la meilleure solution, mais elle n'est pas la meilleure solution.

Une solution heuristique est donc une solution intuitive, qui se base sur une ***stratégie d'essais et d'erreurs***, qui en quelque sorte repose sur la chance. Un algorithme heuristique est plus rapide que l'algorithme de force brute qui énumérerait toutes les solutions possibles afin de trouver la meilleure solution, mais on paie le prix de cette efficacité par de la précision. Un algorithme heuristique aboutit à une solution moins précise et moins complète, à une solution sans garantie. Quand un problème est trop complexe, il ne peut être résolu que par des algorithmes heuristiques, aboutissant dans certains cas à des mauvaises solutions.

Le saviez-vous ? – Que veut dire heuristique ?

Le mot **heuristique** nous vient du grec ancien, plus précisément du terme *heuriskêin*, qui veut dire trouver, inventer, découvrir.

Ce même terme a donné un autre mot bien connu *eurêka*.

L'algorithme heuristique qu'on vient de voir est en fait un **algorithme glouton**, un algorithme qui choisit une solution *localement optimale* (qui choisit la meilleure solution en apparence à un moment donné) sans se préoccuper de toutes les solutions possibles. On espère ainsi que toutes ces décisions localement optimales mènent vers une très bonne solution. C'est un peu comme si on cherchait à atteindre le plus haut sommet d'une montagne, entourés de brouillard, et qu'on prenait une décision sur le chemin à emprunter uniquement en fonction de ce que l'on peut voir juste autour de nous. On pourrait prendre le chemin le plus pentu en espérant qu'il nous mène à un sommet très haut, mais une fois arrivés en haut d'un sommet, on ne peut savoir si notre sommet est bien le plus haut. On l'espère…

Il n'y a pas que des *heuristiques* gloutonnes. Un autre exemple de solution heuristique, très utilisée dans les jeux vidéos, est le calcul de distance entre deux objets. Ce calcul est très important par exemple lorsque l'on souhaite détecter si deux objets sont en collision. Pythagore nous dit que cette distance vaut la racine carrée de la somme de a et b au carré. Mais ce calcul est difficile, et même si on peut le calculer de manière exacte, il prend beaucoup de temps à calculer s'il y a beaucoup d'objets affichés à l'écran. On préfère ainsi estimer cette distance par un calcul bien plus simple $a + b$, que l'on sait faux, mais qui est suffisamment proche lorsque les objets sont alignés (voir la figure ci-dessous).

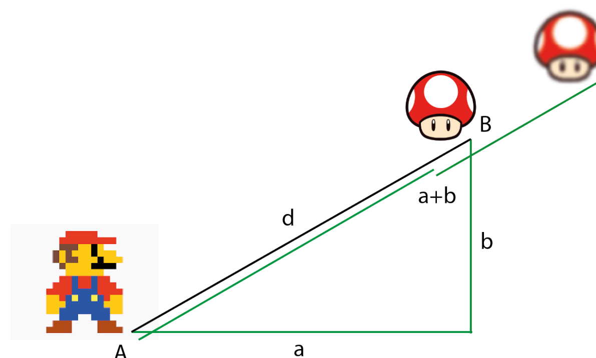


Fig. 1.5 – Exemple d'heuristique. Dans les jeux vidéos, on préfère estimer la distance d entre deux objets A et B par la somme des longueurs des côtés de l'angle droit $a + b$, plutôt que de calculer la racine carrée de la somme des carrés des longueurs des côtés de l'angle droit $d^2 = a^2 + b^2$ (théorème de Pythagore). Même si ce calcul est inexact, il est beaucoup plus rapide à calculer quand il y a beaucoup d'objets à afficher à l'écran, et il est suffisamment précis lorsque les deux objets sont alignés.

Il existe encore d'autres types d'algorithmes *heuristiques*, plus lents, mais qui permettent de s'approcher davantage de la solution optimale. Ils utilisent par exemple des stratégies de résolution statistiques, génétiques ou neuronales. L'apprentissage automatique à qui l'on doit les succès récents de l'intelligence artificielle repose sur des algorithmes heuristiques. La majorité des problèmes que l'on tente de résoudre aujourd'hui sont difficiles et leurs algorithmes de résolution ne trouvent pas la meilleure solution.

Pour aller plus loin

Voici un problème à un million de dollars, un parmi les sept problèmes mathématiques du prix du millénaire qui rapporteront de l'argent à la personne qui les résoudra.

On appelle la classe des problèmes qui sont faciles à résoudre la classe des problèmes P. Ces algorithmes peuvent être résolus en un temps polynomial en fonction de la taille des données n, ou $O(n^a)$.

Une autre classe de problèmes sont les problèmes difficiles à résoudre qui sont d'ordre de complexité exponentielle. Lorsqu'on arrive à vérifier rapidement (en temps polynomial) si une solution proposée permet de résoudre le problème, il s'agit d'une classe de problèmes appellée NP ou « non déterministe polynomial ».

On souhaite savoir si les problèmes NP peuvent être résolus en un temps P ou non, ou en d'autres termes : est-ce que $P = NP$?

S'il s'avérait que c'est bien le cas (ce qui est tout de même peu probable), beaucoup de problèmes difficiles à résoudre deviendraient d'un seul coup plus faciles à résoudre. Un des ces problèmes est le **problème de repliement des protéines** en biologie qui cherche de nouveaux médicaments. Cela pourrait également signifier la fin de la cryptographie telle qu'elle existe actuellement.

1.3.2 Exercices

Exercice 3 – L'univers dans un sac à dos

L'âge estimé de l'univers est de 14 milliards d'années. Si le calcul d'une combinaison d'objets dans le problème du sac à dos prenait une microseconde, pour quel nombre d'objets serait-il possible de trouver une solution exacte sans dépasser l'âge de l'univers ?

Exercice 4 – Parcours du parcours du parcours de listes

Quelle est la complexité d'un algorithme qui pour chacun des éléments d'une liste de n éléments, doit parcourir tous les éléments d'une autre liste de n éléments, puis pour chacune des combinaisons de deux éléments doit encore parcourir une troisième liste de n éléments ?

Si vous avez besoin de travailler sur un exemple plus concret, quelle est complexité de l'algorithme qui calcule tous les menus possibles d'un restaurant à partir d'une liste de n entrées, une liste de n plats et une liste de n desserts ?

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais reconnaître un algorithme de force brute.
2. Je sais reconnaître un algorithme heuristique.
3. Je sais reconnaître un algorithme glouton.
4. Je comprends pourquoi un algorithme de complexité exponentielle est lent.

1.4 Algorithmes de tri [niveau avancé]

Nous venons de voir que pour rechercher de manière efficace, les données doivent être triées. Mais quelle est la complexité de l'algorithme du Tri par sélection vu dans le chapitre Trie, cherche et trouve ? C'est sa complexité qui nous donnera une indication sur sa rapidité.

1.4.1 Tri par sélection

Pour rappel, l'*algorithme* du **Tri par sélection** parcourt le tableau à la recherche des plus petits éléments. Afin de trouver le plus petit élément du tableau, il faut commencer par parcourir tous les éléments du tableau. Cette opération prend cn *instructions* : c instructions pour l'accès et la comparaison des éléments du tableau, multiplié par le nombre d'éléments n. Il faut ensuite trouver le plus petit élément des éléments restants $n-1$, et ainsi de suite. Concrètement, on va parcourir jusqu'à n éléments, n fois (pour chacun des éléments). La complexité du Tri par sélection est donc proportionnelle à $n * n$ (n^2), on parle de complexité **quadratique**.

Si on compare les complexités vues jusqu'ici pour un tableau de 1000 éléments on obtient :

Complexité	Instructions élémentaires pour 1000 éléments
Linéaire	1000
Logarithmique	10
Quadratique	1000000

Avec une complexité quadratique, le Tri par sélection est un algorithme relativement lent.

Exercice 1 – Complexité du Tri par insertion

Quelle est la complexité de l'algorithme de **Tri par insertion** ? En d'autres termes, si le tableau contient n éléments, combien faut-il d'instructions pour trier ce tableau ? Pour rappel, le Tri par insertion parcourt le tableau dans l'ordre et pour chaque nouvel élément, l'insère à l'emplacement correct des éléments déjà parcourus.

Solution 1 – Complexité du Tri par insertion

Dans le pire cas, lorsque les éléments sont dans l'ordre inverse, on doit comparer chacun des n éléments avec 1 à n éléments. La complexité de l'algorithme du Tri par insertion est donc $n * n = n^2$ ou **quadratique**.

Pour aller plus loin – Calcul de complexité

Si vous souhaitez connaître les détails du calcul de complexité, lisez ce qui suit.

Pour calculer la somme totale d'instructions nécessaires, il faut additionner les termes qui permettent de retrouver le plus petit élément. La première fois que l'on recherche le plus petit élément il faut parcourir n éléments. La deuxième fois, il reste à parcourir $n-1$ éléments. La troisième fois, il faut parcourir les $n-2$ éléments restants. Et ainsi de suite, jusqu'à ce qu'il ne reste plus qu'un élément.

Par exemple, si le tableau contient les éléments `[5,2,3,6,1,4]`, pour trouver le plus petit élément 1 à la première itération on doit parcourir tout le tableau, ou 6 éléments. A la deuxième itération, on met l'élément 1 de côté et on parcourt le tableau `[5,2,3,6,4]`, ce qui fait 5 éléments. On met le plus petit élément 2 de côté et dans la troisième itération on parcourt le tableau `[5,3,4,6]`, ce qui fait 4 éléments. On met 3 de côté et on parcourt encore le tableau `[5,4,6]`, ce qui fait 3 éléments. Finalement on se retrouve avec les tableaux `[5,6]` à 2 éléments et `[5]` à 1 élément. La somme totale d'éléments parcourus est $6+5+4+3+2+1=21$.

Si on généralise on obtient :

$$n + (n-1) + (n-2) + ... + (n/2+1) + n/2 + ... + 3 + 2 + 1$$

En réarrangeant les termes deux par deux de l'extérieur vers l'intérieur on obtient plusieurs fois le même terme :

$$(n+1) + ((n-1)+2) + ((n-2)+3) + ... + ((n/2+1)+n/2)$$

$$(n+1) + (n+1) + (n+1) + ... + (n+1) =$$

$$(n+1) * n/2 =$$

$$n * n/2 + n/2 =$$

$$n^2/2 + n/2$$

Si on reprend l'exemple ci-dessus, on aurait :

$$6+5+4+3+2+1 =$$

$$(6+1) + (5+2)(4+3) =$$

$$7 * (6/2) = 7 * 3 = 21 \quad \text{ou}$$

$$6^2/2 + 6/2 = 36/2 + 3 = 18 + 3 = 21$$

Le terme dominant dans la somme $n^2/2 + n/2$ est $n^2/2$, plus n grandit plus $n/2$ est insignifiant par rapport à $n^2/2$. Par exemple, pour $n = 1000$, $n^2/2$ vaut 500000, alors que $n/2$ vaut seulement 500.

Cette somme nous donne le nombre d'éléments parcourus. Mais pour chacun de ces éléments, plusieurs instructions sont exécutées, comme l'accès aux éléments et leur comparaison. Ces instructions et le terme qui divise par 2 peuvent être absorbés dans une *constante* c qui multiplie le terme quadratique n^2. En ajoutant une constante a pour prendre en compte le nombre d'instructions qui ne dépendent pas de la taille des données (comme les initialisations au début de l'algorithme), on obtient l'ordre de grandeur $cn^2 + a$. L'ordre de grandeur est donc **quadratique**.

Exercice 2 – Complexité du Tri à bulles

Quelle est la complexité de l'algorithme de **Tri à bulles** ? En d'autres termes, si le tableau contient n éléments, combien faut-il d'instructions pour trier ce tableau ? Pour rappel, le Tri à bulles compare les éléments deux par deux en les réarrangeant dans le bon ordre, afin que l'élément le plus grand remonte vers la fin du tableau tel une bulle d'air dans de l'eau. Cette opération est répétée n fois, pour chaque élément du tableau.

Solution 2 – Complexité du Tri à bulles

Dans le cas du Tri à bulles, pour chacun des n éléments on parcourt jusqu'à n éléments de la liste, ce qui nous donne une complexité $n * n = n^2$ ou une complexité quadratique.

1.4.2 Tri rapide

Les trois *algorithmes* de tri vus dans le chapitre précédent – le Tri par sélection, le Tri par insertion et le Tri à bulles – sont des algorithmes de complexité quadratique. Si on devait utiliser ces tris dans les systèmes numériques en place, on passerait beaucoup de notre temps à attendre. Il existe d'autres algorithmes de tri qui sont bien plus rapides. Nous allons voir un *algorithme* de tri tellement rapide, qu'on lui a donné le nom **Tri rapide**.

On commence par prendre un élément du tableau que l'on définit comme ***élément pivot***. Cet élément pivot est en général soit le premier ou le dernier élément du tableau, soit l'élément du milieu du tableau ou encore un élément pris au hasard. On compare ensuite tous les autres éléments du tableau à cet élément pivot. Tous les éléments qui sont plus petits que le pivot seront mis à sa gauche et tous les éléments qui sont plus grands que le pivot seront mis à sa droite, tout en conservant leur ordre (voir la deuxième ligne de la figure ci-dessous).

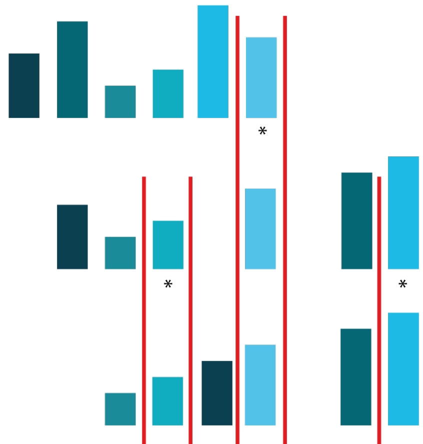


Fig. 1.6 – Tri rapide. Illustration du tri rapide sur les mêmes données que celles utilisées pour illustrer les algorithmes de tri du chapitre précédent. On choisit comme élément pivot le dernier élément des tableaux à trier. Tous les éléments qui sont plus petits que le pivot se retrouvent à sa gauche, tous les éléments plus grands que le pivot se retrouvent à sa droite. L'ordre est conservé.

Après la répartition des éléments autour de l'élément pivot en fonction de leur taille, on se retrouve avec deux tableaux non triés, un sous-tableau à chaque côté de l'élément pivot. On répète les mêmes opérations que pour le tableau initial. Pour chaque sous-tableau, celui de gauche et celui de droite du pivot, on détermine un nouvel élément pivot (le dernier élément du sous-tableau). Pour chaque nouvel élément pivot, on met à gauche les éléments du sous-tableau qui sont plus petits que le pivot et on met à droite les éléments du sous-tableau qui sont plus grands que le pivot. **On répète** ces mêmes opérations jusqu'à ce qu'il ne reste plus que des tableaux à 1 élément (plus que des pivots). A ce stade, le tableau est trié.

Intéressons-nous maintenant à la complexité de cet algorithme. A chaque étape (à chaque ligne de la figure ci-dessus), on compare tout au plus $n$éléments avec les éléments pivots, ce qui nous fait un multiple de n *instructions* élémentaires. Mais combien d'étapes faut-il pour que l'algorithme se termine ?

Dans le meilleur cas, à chaque étape de l'algorithme, l'espace de recherche est divisé par 2. Nous avons vu dans le chapitre Recherche binaire que lorsqu'on divise l'espace de recherche par deux, on obtient une complexité logarithmique. Le nombre d'étapes nécessaires est donc un multiple de $log(n)$.

Pour obtenir le nombre total d'instructions élémentaires on multiplie le nombre d'instructions par étape avec le nombre d'étapes. La complexité que l'obtient est une fonction de $nlog(n)$, il s'agit d'une complexité **linéarithmique**. Un algorithme avec une complexité linéarithmique est plus lent qu'un algorithme de complexité linéaire (la recherche linéaire) ou de complexité logarithmique (la recherche binaire). Par contre, il est bien plus rapide qu'un algorithme de complexité quadratique (le tri par sélection). La figure ci-dessous permet de comparer la vitesse de croissance des complexités étudiées jusqu'ici. Le tri rapide est donc vraiment l'algorithme de tri le plus rapide vu jusqu'ici et la complexité nous permet de comprendre pourquoi c'est le cas.

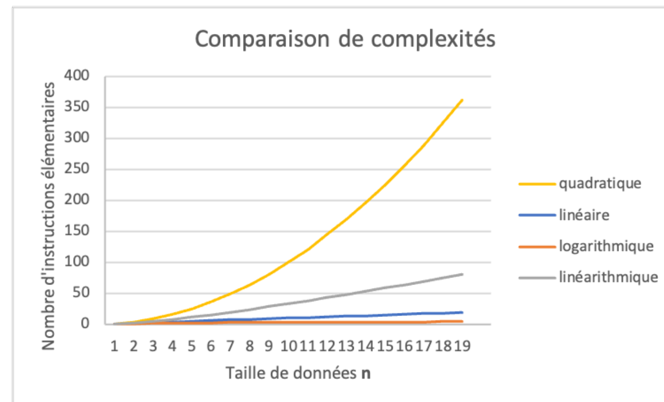


Fig. 1.7 – Comparaison de complexités. La vitesse de croissance en fonction de la taille du tableau n est montrée pour toutes les complexités vues jusqu'ici. Plus le nombre d'instructions élémentaires est grand en fonction de la taille des données, plus l'algorithme est lent.

La première question que l'on se pose lorsqu'on analyse un algorithme est son ordre de complexité temporelle. Si l'algorithme est trop lent, il ne sera pas utilisable dans la vie réelle. Pour le problème du Tri, la stratégie « **diviser pour régner** » vient à nouveau à la rescousse.

Exercice 3 – Le pire du Tri rapide

Que se passe-t-il si on essaie de trier un tableau déjà trié avec l'algorithme du **Tri rapide**, en prenant toujours comme pivot le dernier élément ? Essayer par exemple avec le tableau `[1,2,3,4,5,6,7]`.

Combien d'étapes sont nécessaires pour que l'algorithme se termine ? Quelle est la complexité dans ce cas ? Est-ce qu'un autre choix de pivot aurait été plus judicieux ?

Solution 3 – Le pire du Tri rapide

Si on simule l'algorithme de Tri rapide pour le tableau `[1,2,3,4,5,6,7]` avec comme pivot le dernier élément on se retrouve avec les sous-tableaux suivants (le pivot est affiché en gras, les éléments déjà triés sont affichés en italique) :

```
[1,2,3,4,5,6,**7**]
   [1,2,3,4,5,**6**][*7*][]
   [1,2,3,4,**5**][*6*][][*7*][]
   [1,2,3,**4**][*5*][][*6*][][*7*][]
   [1,2,**3**][*4*][][*5*][][*6*][][*7*][]
   [1,**2**][*3*][][*4*][][*5*][][*6*][][*7*][]
   [**1**][*2*][][*3*][][*4*][][*5*][][*6*][][*7*][]
   [][*1*][][*2*],[*3*][*4*][][*5*][][*6*][][*7*][]
[1][2][3][4][5][6][7]
```

Lorsque les éléments du tableau sont déjà triés, l'espace de recherche n'est plus divisé par deux. On se retrouve avec des sous-tableaux déséquilibrés, vides d'un côté et pleins de l'autre. Le nombre d'étapes n'est donc plus $log(n)$, mais vaut n (7 étapes de traitement). Lorsqu'on multiple le nombre d'étapes (lignes) au nombre d'éléments à comparer par ligne, on est plutôt dans une complexité $n * n$ (ou n^2), donc quadratique. Dans ce scénario, le tri rapide n'est donc plus si rapide. Le choix du pivot est alors crucial et dépend du contenu du tableau.

Si on prend comme pivot l'élément du milieu du tableau, on se retrouve avec des sous-tableaux équilibrés, qui contiennent un nombre similaire d'éléments. Dans ce cas l'algorithme a besoin de moins d'étapes pour trouver la solution, de l'ordre de $log(n)$, ici 3 lignes et équivalent à $log_2(7)$, de traitement au lieu de 7 auparavant :

```
[1,2,3,**4**,5,6,7]

   [1,**2**,3][*4*],[5,**6**,7]

   [**1**][*2*][**3**][*4*],[**5**][*6*][**7**]

   [][*1*][][*2*][][*3*][][*4*][][*5*][][*6*][][*7*]

[1][2][3][4][5][6][7]
```

Pour aller plus loin

Même si deux algorithmes de tri ont la même complexité temporelle, c'est-à-dire qu'ils prennent un temps comparable pour trier des données, ils ne prennent pas la même place en mémoire. Pour un algorithme qui prend peu de place en memoire (par exemple le tri par insertion), on dit qu'il a une plus petite « **complexité spatiale** ».

Si on trie un tableau qui est en fait déjà trié avec le tri par insertion, la complexité dans ce cas est linéaire. Au contraire, si on trie ce même tableau avec le tri rapide, la complexité dans ce cas est quadratique. On voit donc que selon le tableau que l'on trie, le tri rapide peut être bien plus lent que le tri par insertion.

Une analyse complète d'un algorithme consiste à calculer la complexité non seulement dans le **cas moyen**, mais aussi dans le **meilleur cas** et dans le **pire cas**.

Pour aller plus loin

Une analyse complète va également calculer les constantes qui influencent l'ordre de complexité. Ces constantes ne sont pas importantes lors d'une première analyse d'un algorithme. En effet, les constantes n'ont que peu d'effet pour une grande taille des données n, c'est uniquement le terme qui grandit le plus rapidement en fonction de n qui compte, et qui figure dans un premier temps dans l'ordre de complexité. Par contre, lorsque l'on souhaite comparer deux algorithmes de même complexité, il faut estimer les constantes et choisir l'algorithme avec une constante plus petite.

La notation « Grand O », que l'on utilise pour écrire mathématiquement la complexité, désigne en fait la complexité dans le pire cas. Les différentes complexités vues jusqu'ici seraient notées : $O(n)$, $O(log(n))$, $O(n^2)$ et $O(nlog(n))$. Arrivez-vous à trouver les adjectifs correspondants ?

Exercice 4 – Le meilleur et le pire du Tri par insertion

Que se passe-t-il si on essaie de trier un tableau déjà trié avec l'algorithme du **Tri par insertion** ? Essayer par exemple avec le tableau `[1,2,3,4,5,6,7]`.

Combien d'étapes sont nécessaires pour que l'algorithme se termine ? Quelle est la complexité dans ce cas ?

Que se passe-t-il si on essaie de trier un tableau déjà trié, mais dans l'ordre inverse de celui qui est souhaité, avec l'algorithme du Tri par insertion ? Essayer par exemple avec le tableau `[4,3,2,1]`.

Solution 4 – Le meilleur et le pire du Tri par insertion

Si on simule l'algorithme de Tri par insertion pour le tableau `[1,2,3,4,5,6,7]` on se retrouve avec la configuration suivante (l'élément inséré est affiché en gras, l'élément auquel on le compare en italique) :

```
[**1**,2,3,4,5,6,7]

[*1*][**2**,3,4,5,6,7]

[1,*2*][**3**,4,5,6,7]

[1,2,*3*][**4**,5,6,7]

[1,2,3,*4*][**5**,6,7]

[1,2,3,4,*5*][**6**,7]

[1,2,3,4,5,*6*][**7**]
```

`[1,2,3,4,5,6,7]`

On voit qu'il y a besoin de 7 étapes, ou n étapes, car autant que d'éléments dans le tableau. Dans chaque étape on n'a besoin de comparer qu'une fois, avec l'élément précédent. La complexité dans ce cas est $n * 1 = n$ ou linéaire. Pour des données presque triées, le Tri par insertion est encore plus rapide que le Tri rapide.

A premier abord, trier le tableau `[5, 4,3,2,1]` avec le Tri par insertion ne présente pas de difficultés. Regardons ce qui se passe :

```
[**5**,4, 3,2,1]

[*5*][**4**,3,2,1]

[4,*5*][**3**,2,1]

[*4*,**3**,5][2,1]

[3,4,*5*][**2**,1]

[3,*4*,**2**,5][1]

[*3*,**2**,4,5][1]
```

```
    [2,3,4,*5*][**1**]

    [2,3,*4*,**1**,5]

    [2,*3*,**1**,4,5]

    [*2*,**1**,3,4,5]

[1,2,3,4,5]
```

Cette fois-ci on se retrouve dans la pire configuration pour le Tri par insertion, où chaque élément doit être comparé à chaque autre élément. Ici nous avons besoin de 11 étapes de traitement pour trier 5 éléments, alors qu'avant 7 étapes suffisaient pour trier 7 éléments. Lorsqu'on doit trier un grand nombre d'éléments, ces différence est significative et peut rendre un algorithme nonutilisable.

1.4.3 Exercices

Exercice 5 – Une question à un million

Si une instruction prend 10^{-6} secondes, combien de temps faut-il pour trier un tableau d'un million d'éléments avec le tri à sélection comparé au tri rapide (sans tenir compte de la constante) ?

Exercice 6 – Une question de pivot

Trier le tableau suivant avec l'algorithme de tri rapide : `[3, 6, 8, 7, 1, 9, 4, 2, 5]` à la main, en prenant le dernier élément comme pivot. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Est-ce que le choix du pivot est important ?

Exercice 7 – Une question de sélection

Trier le tableau suivant avec l'algorithme de tri par sélection : `[3, 6, 8, 7, 1, 9, 4, 2, 5]` à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 8 – Une question d'insertion

Trier le tableau suivant avec l'algorithme de tri par insertion : `[3, 6, 8, 7, 1, 9, 4, 2, 5]` à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 9 – Une question de bulles

Trier le tableau suivant avec l'algorithme de tri à bulles : `[3, 6, 8, 7, 1, 9, 4, 2, 5]` à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 10 – Une question de chronomètre

Créer une liste qui contient les valeurs de 1 à n dans un ordre aléatoire, où n prend la valeur 100. Implémenter au moins deux des trois algorithmes de tri vu au cours.

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 peut-on constater ? Sur la base de ces mesures, peut-on estimer le temps que prendrait le tri de 100000 éléments ? Lancer votre programme avec 100000 éléments et comparer le temps obtenu avec votre estimation.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais que grâce à la stratégie algorithmique « diviser pour régner », je ne passe pas mon temps à attendre que l'ordinateur me donne une réponse.
2. Je sais comment trier de manière rapide.
3. Je sais que la complexité temporelle peut changer selon la configuration des données, en plus du cas moyen, il est également utile d'estimer le pire et le meilleur cas.

1.5 Récursivité [niveau avancé]

Ce chapitre est prévu en tant que chapitre optionnel. Il présente un autre algorithme de tri célèbre, le **Tri par fusion**. Cet algorithme utilise la **récursivité**, une stratégie qui consiste en ce qu'un algorithme s'invoque lui-même. La récursivité, c'est un peu comme si on essayait de définir le terme « définition » en disant c'est une phrase qui nous donne la définition de quelque chose. C'est certes circonvolu que de vouloir utiliser dans une définition *la chose-même* que l'on est en train de définir, mais si on respecte quelques conditions, « ça fonctionne » !

1.5.1 Tri par fusion

Un autre *algorithme* de tri célèbre, inventé par John von Neumann en 1945, est le **Tri par fusion**. L'algorithme se base sur l'idée qu'il est difficile de trier un tableau avec beaucoup d'éléments, mais qu'il est très facile de trier un tableau avec juste deux éléments. Il suffit ensuite de fusionner les plus petits tableaux déjà triés.

L'algorithme commence par une phase de ***division*** : on divise le tableau en deux, puis on divise *à nouveau* les tableaux ainsi obtenus en deux, et ceci jusqu'à arriver à des tableaux avec un seul élément (voir la figure ci-dessous). Comme pour la recherche binaire, le nombre d'étapes nécessaires pour arriver à des tableaux de 1 élément, en divisant toujours par deux, est $log(n)$.

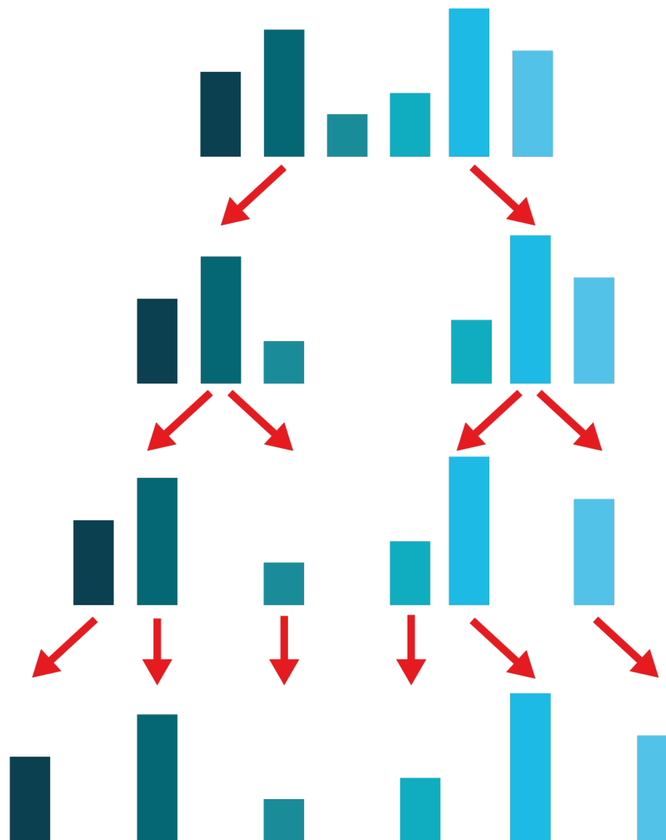


Fig. 1.8 – Phase de division. Illustration de la première phase du Tri par fusion : on commence par diviser le tableau en deux, puis à chaque étape on divise à nouveau les tableaux ainsi obtenus par deux, jusqu'à ce qu'il n'y ait plus que des tableaux à 1 élément.

La deuxième phase de ***fusion*** commence par fusionner des paires de tableaux à un élément, dans un ***ordre trié***. Il suffit d'assembler les deux éléments du plus petit au plus grand, comme on peut le voir sur la 2ème ligne de la figure ci-dessous. Dans les prochaines étapes, on continue à fusionner les tableaux par paires de deux, tout en respectant l'ordre de tri (lignes 3 et 4 de la figure). On continue de la sorte jusqu'à ce qu'il n'y ait plus de tableaux à fusionner.

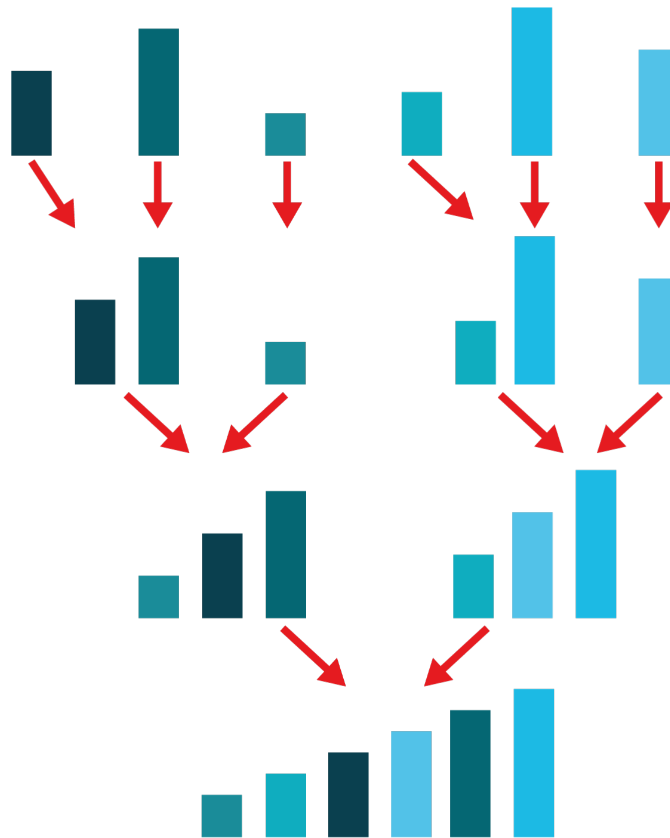


FIG. 1.9 – Phase de fusion. Illustration de la deuxième phase du Tri par fusion : on commence par fusionner les tableaux à un élément, en faisant attention à respecter l'ordre de tri (ligne 2); puis par fusionner à nouveau les tableaux obtenus à l'étape précédente, toujours en respectant l'ordre de tri (lignes 3 et 4). On continue de la sorte jusqu'à ce qu'il n'y ait plus qu'un tableau unique (ligne 4).

La fusion de tableaux **déjà triés**, par rapport à des tableaux non-triés, est très facile. Il suffit de comparer les premiers éléments des deux tableaux à fusionner et de prendre le plus petit des deux. Concrètement, on enlève le plus petit élément des deux tableaux pour le mettre dans le nouveau tableau fusionné. On compare ensuite les premiers éléments de ceux qui restent dans les tableaux à fusionner et on prend à nouveau le plus petit des deux pour le mettre à la suite dans le tableau fusionné.

Chaque étape de la phase de fusion consiste à comparer deux éléments n fois, autant de fois qu'il y a d'éléments à fusionner. Le temps de calcul grandit donc linéairement en fonction de la taille du tableau n (plus il y a d'éléments dans le tableau, plus la fusion prend du temps). En tout il y a besoin de $log(n)$ étapes (fusion deux par deux), dont chacune prend un temps qui dépend de n, ce qui nous donne un ordre de complexité linéarithmique.

1.5.2 Focus sur la récursivité

Nous allons maintenant programmer l'*algorithme* du Tri par fusion. Pour rappel, la première phase de l'*algorithme* divise *continuellement* le tableau par deux, comme illustré dans la première figure ci-dessus. Voici le code qui permet de diviser un tableau en deux une seule fois :

```
# Tri par fusion
def tri_par_fusion(elements):

        ### Phase DIVISION

        # détermine l'indice au milieu du tableau (division entière)
        milieu = len(elements)//2

        # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
        elements_gauche = elements[:milieu]

        # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
        elements_droite = elements[milieu:]
```

La division utilisée pour déterminer le milieu du tableau est une division entière `//` au lieu de `/`. En effet, on souhaite obtenir un résultat entier et non un nombre à virgule, car les indices pour accéder aux éléments du tableau doivent être des entiers. Par exemple, si le tableau contient `5` éléments, cela n'aurait pas de sens de prendre les premiers `2.5` éléments, et `5//2` nous retournerait `2`.

Ce qui suit est très intéressant. Dans l'étape d'après, on souhaite faire exactement la même chose pour les nouveaux tableaux `elements_gauche` (équivalent à `elements[:milieu]`) et `elements_droite` (équivalent à `elements[milieu:]`), c'est-à-dire que l'on souhaite à nouveau les diviser en deux, comme sur la deuxième ligne dans la première figure ci-dessus. On va donc appeler la fonction `tri_par_fusion` sur les deux moitiés de tableaux :

```
        # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
        elements_gauche = tri_par_fusion(elements[:milieu])

        # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
        elements_droite = tri_par_fusion(elements[milieu:])
```

Regardez bien ce qui se passe. Nous avons fait appel à la même *fonction* `tri_par_fusion` que l'on est en train de définir ! Pour l'instant cette fonction ne fait que diviser le tableau `elements` en deux, elle va donc diviser le tableau reçu en entrée en deux. Au début le tableau en entrée sera le tableau entier, mais ensuite il s'agira des deux moitiés du tableau, puis des moitiés de la moitié et ainsi de suite. La fonction `tri_par_fusion` appelle la fonction `tri_par_fusion` (elle s'appelle donc elle-même), qui va à nouveau s'appeler et ainsi de suite…

Si on laisse le programme tel quel, on est face à un problème. La fonction `tri_par_fusion` continue de s'appeler elle-même et ce processus ne s'arrête jamais. En réalité, il faut arrêter de diviser lorsque les tableaux obtenus ont au moins un élément ou lorsqu'ils sont vides, car dans ces cas on ne peut plus les diviser en deux. On rajoute donc cette **condition d'arrêt** de la récursion :

```
        # condition d'arrêt la récursion
        if len(elements) <= 1:
                return(elements)

        # détermine l'indice au milieu du tableau (division entière)
        milieu = len(elements)//2

        # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
        elements_gauche = tri_par_fusion(elements[:milieu])

        # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
        elements_droite = tri_par_fusion(elements[milieu:])
```

Voici le programme appliqué sur l'exemple de la figure. Essayez de comprendre dans quel ordre sont appelées les fonctions `tri_par_fusion` et avec quel paramètre en entrée. Pour une meilleure visibilité, nous affichons l'état des *variables* avec `print`.

```
# Tri par fusion : phase de division
def division(elements, ligne, side=0):

        # nous dit où on en est
        print("Appel de la fonction division avec ", str(elements), "ligne", ligne,
             "depuis", side)

        # correspond à la ligne sur la figure division du tri par fusion
        ligne = ligne + 1

        # condition d'arrêt la récursion
        if len(elements) <= 1:
                return(elements)

        # détermine l'indice au milieu du tableau (division entière)
        milieu = len(elements)//2

        # prend tous les éléments depuis le début, jusqu'à  (et sans) milieu
        elements_gauche = division(elements[:milieu], ligne, 'gauche')
        if elements_gauche :
                print('Eléments à gauche : ', elements_gauche, 'de :', elements, "
                    ligne", ligne)

        # prend tous les éléments depuis le milieu (y compris), jusqu'à  la fin
        elements_droite = division(elements[milieu:], ligne, 'droite')
        if elements_droite :
                print('Eléments à droite : ', elements_droite, 'de :', elements, "
                    ligne", ligne)

division([3,5,1,2,6,4], 0)
```

Une *fonction* qui s'appelle elle-même est appelée ***fonction récursive***. Il s'agit d'une *miseenabime*, d'une *définitioncirculaire*. Lorsqu'on entre dans la fonction, des opérations sont exécutées et on fait ànouveau ***appel à la même fonction***, mais cette fois-ci avec ***d'autres éléments en entrée***, afin de refaire les mêmes opérations, comme le montre cette figure :

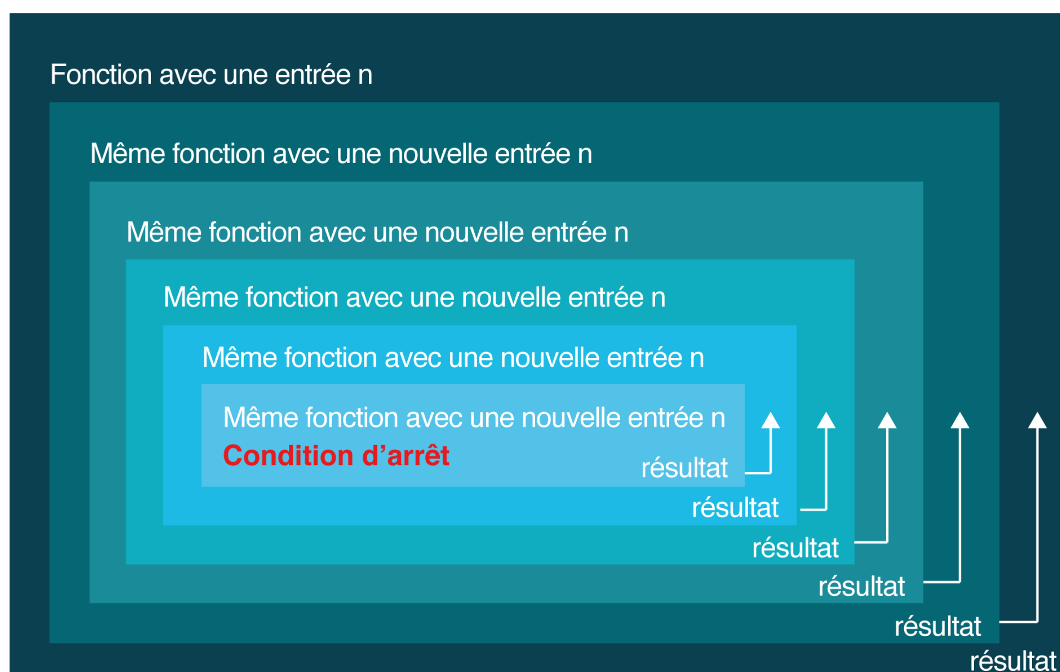


Fig. 1.10 – Schéma d'une fonction récursive. La fonction s'appele elle-même. toujours avec un autre paramètre en entrée, jusqu'à ce que la condition d'arrêt soit remplie. A ce moment-là, un résultat est calculé et retourné à la fonction du dessus (celle qui à appelé la fonction). Ainsi tous les résultats sont retournés au fur et à mesure et permettent de calculer la fonction souhaitée.

Les deux ingrédients indispensables à toute *fonction récursive* sont donc :

1. un **appel à la fonction elle-même** à l'intérieur de la définition de la fonction.
2. une **condition d'arrêt**, qui permet de terminer les appels imbriqués.

Exercice 1 – Position de la condition d'arrêt

Sans la condition d'arrêt, un programme récursif ne se termine pas, et s'appelle soi-même indéfiniment. Il est important que cette condition d'arrêt précède l'appel récursif à la fonction. Pourquoi est-ce le cas ?

Solution 1 – Position de la condition d'arrêt

Cliquer ici pour voir la réponse

Si la condition d'arrêt est après l'appel à la fonction, la fonction est appelée avant d'avoir pu vérifié si la condition d'arrêt est remplie. Dans ce cas, la condition d'arrêt n'est jamais testée.

Maintenant que nous avons programmé la première phase de division du Tri par fusion il nous faut programmer la deuxième phase de fusion (voir la deuxième figure du Tri par fusion). Nous allons définir cette phase de fusion de manière récursive :

```
# Phase de fusion du Tri par fusion
def fusion(elements_gauche, elements_droite):

        # trouve le plus petit premier élément des deux listes
        if elements_gauche[0] < elements_droite[0]:

                # appelle fusion récursivement avec le reste des listes
                elements_reste = fusion(elements_gauche[1:], elements_droite)

                # crée une liste fusionnée avec le résultat
                elements_fusion = [elements_gauche[0]] + elements_reste

        else:

                # appelle fusion récursivement avec le reste des listes
                elements_reste = fusion(elements_gauche, elements_droite[1:])

                # crée une liste fusionnée avec le résultat
                elements_fusion = [elements_droite[0]] + elements_reste

        return(elements_fusion)
```

Quelle est la différence entre le code dans la partie `if` de la condition et dans la partie `else` de la condition ? Lorsqu'on fusionne deux tableaux qui sont **déjà triés**, le plus petit élément se trouve parmi les premiers éléments des deux tableaux à fusionner. On commence alors par prendre le plus petit des premiers éléments des deux tableaux à fusionner, que l'on met au début de notre tableau fusionné. On refait ensuite la même opération avec le reste des éléments : on sélectionne le plus petit élément des tableaux de départ et on le met à la suite de notre tableau fusionné. On recommence de la sorte tant qu'il n'y ait plus d'éléments dans les tableaux.

Dans la partie `if` de la fonction `fusion`, c'est le tableau de gauche qui contient le plus petit élément. On prend cet élément pour le mettre au début d'un nouveau tableau fusionné et on appelle la fonction `fusion` sur les éléments restants. Dans la partie `else` on fait la même chose, sauf que l'on commence notre tableau fusionné par le premier élément du tableau de droite.

Mais n'y a-t-il pas quelque chose qui manque à cette fonction ? En effet, il manque la condition d'arrêt. Il faut arrêter la fusion lorsqu'un des deux tableaux à fusionner est vide. Dans ce cas la solution de fusionner un tableau vide avec un autre tableau est triviale : c'est l'autre tableau non vide. Mettons ceci sous forme de code :

```
# Phase de fusion du Tri par fusion
def fusion(elements_gauche, elements_droite):

        # conditions d'arrêt de la récursivité
        if elements_gauche == []:
```

(suite sur la page suivante)

(suite de la page précédente)

```
                return(elements_droite)
        if elements_droite == []:
                return(elements_gauche)

        # trouve le plus petit premier élément des deux listes
        if elements_gauche[0] < elements_droite[0]:

                # appelle fusion récursivement avec le reste des listes
                elements_reste = fusion(elements_gauche[1:], elements_droite)

                # crée une liste fusionnée avec le résultat
                elements_fusion = [elements_gauche[0]] + elements_reste

        else:

                # appelle fusion récursivement avec le reste des listes
                elements_reste = fusion(elements_gauche, elements_droite[1:])

                # crée une liste fusionnée avec le résultat
                elements_fusion = [elements_droite[0]] + elements_reste

        return(elements_fusion)
```

Pour que le programme soit complet, il faut faire appel cette fonction `fusion` dans la fonction `tri_fusion` ci-dessus :

```
# Phase de division du Tri par fusion
def tri_par_fusion(elements):

        ### Phase DIVISION

        # condition d'arrêt la récursion
        if len(elements) <= 1:
                return(elements)

        # détermine l'indice au milieu du tableau (division entière)
        milieu = len(elements)//2

        # prend tous les éléments depuis le début, jusqu'à (et sans) milieu
        elements_gauche = tri_par_fusion(elements[:milieu])

        # prend tous les éléments depuis le milieu (y compris), jusqu'à la fin
        elements_droite = tri_par_fusion(elements[milieu:])

        # fusionne les éléments un par un, puis deux par deux, etc..
        resultat = fusion(elements_gauche, elements_droite)

        # retourner le résultat
        return(resultat)
```

Ces deux *fonctions* fusion et division ensemble implémentent l'*algorithme* du Tri par fusion de manière *récursive*. La *récursivité* est un concept difficile à appréhender. Le mieux est d'essayer de coder différents *algorithmes récursifs* et d'afficher ce qui se passe au fur et à mesure. Voici le programme du tri par fusion :

```
# TRI PAR FUSION

# Phase de fusion
def fusion(elements_gauche, elements_droite):

      # conditions d'arrêt de la récursivité
      if elements_gauche == []:
            print("\n4. Tableau gauche vide :", elements_droite)
            return(elements_droite)
      if elements_droite == []:
            print("\n5. Tableau droite vide :", elements_droite)
            return(elements_gauche)

      # trouve le plus petit premier élément des deux listes
      if elements_gauche[0] < elements_droite[0]:

            # appelle fusion récursivement avec le reste des listes
            elements_reste = fusion(elements_gauche[1:], elements_droite)

            # crée une liste fusionnée avec le résultat
            elements_fusion = [elements_gauche[0]] + elements_reste

            # affiche ce qui se passe
            print("\n6. Retour fusion :", [elements_gauche[0]], '+',
                elements_reste)

      else:

            # appelle fusion récursivement avec le reste des listes
            elements_reste = fusion(elements_gauche, elements_droite[1:])

            # crée une liste fusionnée avec le résultat
            elements_fusion = [elements_droite[0]] + elements_reste

            # affiche ce qui se passe
            print("\n7. Retour fusion :", [elements_droite[0]], '+',
                elements_reste)

      # retourner le résultat
      return(elements_fusion)
```

La fonction division s'appelle aussi elle-même, mais appelle également la fonction fusion précédemment définie :

```
# TRI PAR FUSION

# Phase de division
def division(elements, ligne, side=0):

```

```
        # nous dit où on en est                                                6
        print("\n1. Appel de la fonction division avec ", str(elements), "ligne",   7
            ligne, "depuis", side)
                                                                               8
        # correspond à la ligne sur la figure division du tri par fusion  9
        ligne = ligne + 1                                                      10
                                                                               11
        # condition d'arrêt la récursion                                       12
        if len(elements) <= 1:                                                 13
                return(elements)                                               14
                                                                               15
        # détermine l'indice au milieu du tableau (division entière)           16
        milieu = len(elements)//2                                              17
                                                                               18
        # prend tous les éléments depuis le début, jusqu'à  (et sans) milieu   19
        elements_gauche = division(elements[:milieu], ligne, 'gauche')         20
        if elements_gauche :                                                   21
                print('\n2. Eléments à gauche : ', elements_gauche, 'de :', elements   22
                    , "ligne", ligne)
                                                                               23
        # prend tous les éléments depuis le milieu (y compris), jusqu'à  la fin    24
        elements_droite = division(elements[milieu:], ligne, 'droite')         25
        if elements_droite :                                                   26
                print('\n3. Eléments à droite : ', elements_droite, 'de :', elements   27
                    , "ligne", ligne)
                                                                               28
        #Â fusionne les éléments un par un, puis deux par deux, etc..          29
        resultat = fusion(elements_gauche, elements_droite)                    30
                                                                               31
        # retourner le résultat                                                32
        return(resultat)                                                       33
                                                                               34
resultat = division([3,5,1,2,6,4], 0)                                          35
                                                                               36
print("\nVoici le tableau trié : ", resultat)                                  37
```

1.5.3 Exercices

Exercice 2 – Fractale

Une fractale est un objet géométrique, dont la définition récursive est naturelle. Essayez le code suivant pour différentes valeurs de n (augmenter à chaque fois de 1).

Essayez de comprendre comment le flocon se construit de manière **récursive**. Vous pouvez aussi varier la longueur du segment dessiné et la vitesse d'affichage en décommentant la ligne correspondante.

```
import turtle                                  1
                                               2
def courbeKoch(n, segment) :                   3
        if n == 0 :                            4
```

```
            turtle.forward(segment)
        else :
            courbeKoch(n-1, segment/3)
            turtle.left(60)
            courbeKoch(n-1, segment/3)
            turtle.left(-120)
            courbeKoch(n-1, segment/3)
            turtle.left(60)
            courbeKoch(n-1, segment/3)

def flocon(n, segment) :
    for i in range(3) :
        courbeKoch(n, segment)
        turtle.left(-120)

turtle.hideturtle()   # cache la tortue
# turtle.speed(0)   # ACCELERE LA TORTUE
turtle.forward(-10)   # positionne la tortue en haut à gauche
turtle.left(150)
turtle.forward(150)
window = turtle.Screen()
window.bgcolor("black")  # tableau noir
turtle.color("white")  # dessine avec une trace blanche
turtle.setheading(0)  # orientation initiale de la tête : droite

# AUGMENTER ICI
n = 1
# DIMINUER ICI
segment = 300
flocon(n, segment)  # dessine le flocon
turtle.exitonclick() # garde la fenêtre ouverte
```

Exercice 3 – Une question de fusion

Trier le tableau suivant avec l'algorithme de tri par fusion : [3, 6, 8, 7, 1, 9, 4, 2, 5] à la main. Représenter l'état du tableau lors de toutes les étapes intermédiaires.

Exercice 4 – Dans l'autre sens

En Python, proposer une fonction qui inverse l'ordre des lettres dans un mot. Vous pouvez parcourir les lettres du mot directement ou à travers un indice.

Proposer une autre fonction qui inverse l'ordre des lettres dans un mot de manière récursive.

Exercice 5 – Factorielle

La fonction factorielle $n!$ en mathématiques est le produit de tous les nombres entiers jusqu'à n. C'est une des fonctions les plus simples à calculer de manière récursive. Elle peut être définie comme ceci :

$n! = (n - 1)! * n$

Programmer cette fonction de manière récursive en Python. Proposer également une implémentation itérative de la factorielle où les éléments de 1 à n sont traités l'un après l'autre.

1.6 Conclusion

Important

L'analyse de complexité des algorithmes nous permet de sélectionner les meilleurs algorithmes pour un problème donné et nous permet de comprendre pourquoi certains problèmes ne peuvent pas être (à ce stade) résolus dans un temps raisonnable.

L'algorithmique a permis de mettre en place des stratégies intelligentes de résolution de problèmes comme les principes de « diviser pour régner » ou encore la récursivité. Ces stratégies ont rendu possibles les avancées technologiques fulgurantes du dernier demi-siècle.

Pour des problèmes difficiles, s'il est impossible de trouver la solution exacte, on peut souvent trouver une solution approchée. L'étude formelle de l'algorithmique nous permet d'estimer la qualité de notre solution approchée.

À retenir

Dans la pratique, il est important de garantir qu'un algorithme va se **terminer**.

L'algorithme de tri rapide (et du tri par fusion) est plus efficace que les algorithmes de tri vus dans les chapitres précédents. Ceci est possible grâce à la stratégie algorithmique **« diviser pour régner »**, qui divise un grand problème difficile à résoudre en plein de petits sous-problèmes plus faciles à résoudre. La solution au grand problème s'obtient en combinant les solutions des petits problèmes.

L'**ordre de complexité** d'un algorithme nous dit si l'algorithme est lent ou rapide. Un algorithme avec un ordre de complexité logarithmique est plus rapide qu'un algorithme avec complexité linéaire, qui à son tour est plus rapide qu'un algorithme de complexité quadratique, ou pire, exponentielle.

Un algorithme **récursif** est un algorithme qui fait appel à lui-même. Une condition d'arrêt est nécessaire pour que l'algorithme se termine.

Un algorithme avec une **complexité exponentielle** implique que le temps nécessaire pour résoudre problème est trop long en pratique. Dans ce cas, on ne va pas pouvoir trouver une solution exacte, mais seulement une solution approchée en utilisant des méthodes heuristiques.

Ai-je compris ?

Vérifiez votre compréhension :

1. Je sais appliquer l'algorithme de recherche binaire.
2. Je comprends comment fonctionne la stratégie algorithmique « diviser pour régner ».
3. Je sais appliquer l'algorithme de tri rapide.
4. Je sais calculer la complexité temporelle d'un algorithme.
5. [En option] Je comprends comment fonctionne la récursivité.
6. Je sais pourquoi un algorithme de complexité exponentielle est lent.

7. Je comprends ce qu'est une solution heuristique.

Pour aller plus loin – Quelques liens web

Visualisation de problèmes

https ://imgur.com/gallery/voutF

https ://interstices.info/le-probleme-du-sac-a-dos/

https ://visualgo.net/en

https ://graphonline.ru/fr

https ://clementmihailescu.github.io/Pathfinding-Visualizer/

Problèmes difficiles

https ://www.franceculture.fr/emissions/le-journal-des-sciences/le-journal-des-sciences-du-mardi-01-decembre-2020

https ://www.bfmtv.com/sciences/ou-est-charlie-l-algorithme-pour-le-detecter-du-premier-coup_AN-201502100004.html

https ://www.lebigdata.fr/algorithme-definition-tout-savoir

P = NP ?

https ://www.youtube.com/watch ?v=AgtOCNCejQ8