Quelques algorithmes classiques

Table des matières

I - Contexte 3

II - Rechercher dans des listes 4

III - Exercice : Appliquer la notion 6

IV - Rechercher dans des listes triées 7

V - Exercice : Appliquer la notion 11

VI - Créer des paires (produit cartésien) 12

VII - Exercice : Appliquer la notion 14

VIII - Suite de Fibonacci 15

IX - Exercice : Appliquer la notion 17

X - Trier des listes (tri à bulles) 18

XI - Exercice : Appliquer la notion 21

XII - Essentiel 22

XIII - Quiz 23

Contenus annexes 25

Crédits des ressources 28

Contexte

Durée : 2h

Environnement de travail : Repl.it

Pré-requis : Connaître les bases du JavaScript, *Connaître le concept de complexité algorithmique* *(cf. p.25)*

[cf. YbtN3GkD]

Pour être un bon développeur, maîtriser des langages de programmation est un bon début mais il est également très important d'avoir une culture des algorithmes classiques. Ce sont des algorithmes qui ont été écrits il y a plusieurs décennies ou même siècles, et qui sont reconnus comme étant **la** meilleure solution pour un problème donné. Les connaître fera gagner du temps à un développeur car il possède une solution déjà prête, tout en lui assurant avoir l'algorithme le plus optimisé possible. La plupart du temps, le développeur devra adapter très légèrement ces algorithmes pour qu'ils répondent aux problèmes spécifiques du quotidien.

Rechercher dans des listes

[cf. kGzqAKdn]

Objectif

- Savoir rechercher efficacement une valeur dans une liste non triée.

Mise en situation

La recherche de valeurs dans une liste est un des problèmes les plus simples et se présentant très souvent dans le quotidien d'un développeur.

Liste non triée

La recherche dans une liste non triée ne fait aucune hypothèse sur la position des éléments. Par exemple, la valeur maximale peut se trouver n'importe où dans la liste. Ainsi, pour trouver une valeur il faudra toujours parcourir les éléments un à un.

Recherche de la position d'une valeur

 *Méthode*

Ce problème consiste à renvoyer l'indice (la position) d'une valeur donnée si elle est présente dans une liste. Dans ce cas aussi, nous travaillons sur une liste non triée. Il sera nécessaire de parcourir toutes les cases une à une. Le programme ci-dessous renvoie toutes les position de la valeur est celle recherchée. Si elle est présente plusieurs fois, le programme affichera chacun des indices correspondant.

```javascript
/** JavaScript : cherche une valeur dans une liste de températures. */
const temperatures = [12, 5, 29, -3]
const val = 5 // La valeur à trouver

for (let i = 0; i < temperatures.length; i++) {
  if (val === temperatures[i]) {
    console.log(i)
  }
}

```

```python
"""Python : cherche une valeur dans une liste de températures. """
temperatures = [12, 5, 29, -3]
val = 5  # Valeur à trouver

for i in range(len(temperatures)):
  if val == temperatures[i]:
    print(i)
```

Recherche du maximum

 *Méthode*

Voici un algorithme qui retourne la valeur maximale d'une liste non triée. L'algorithme teste les éléments un à un et possède une variable qui stocke le maximum.

Cette variable est initialisée avec le premier élément de la liste et est mise à jour dès qu'une valeur plus grande que le maximum actuel est trouvée.

À noter que la recherche du minimum est identique. Il faudra simplement inverser la condition du `if` à l'intérieur de la boucle..

```javascript
/** JavaScript : cherche le maximum d'une liste de températures. */
const temperatures = [12, 5, 29, -3]
let max = temperatures[0]
for (let i = 1; i < temperatures.length; i = i + 1) {
  if (max < temperatures[i]) {
    max = temperatures[i]
  }
}

console.log(max)

```

```python
"""Python : cherche le maximum d'une liste de températures. """
temperatures = [12, 5, 29, -3]
max = temperatures[0]
for i in range(1, len(temperatures)):
  if max < temperatures[i]:
    max = temperatures[i]

print(max)
```

Déroulement pas à pas

Exemple

Pour mieux comprendre cet algorithme, il est intéressant de le faire tourner à la main avec l'exemple `temperatures = [12, 5, 29, -3]` :

- Avant la boucle : max stocker la valeur 12.
- Première itération, i = 1 : `max` est plus grand que 5 donc l'algorithme passe à l'itération suivante.
- Deuxième itération, i = 2 : `max` est plus petit que 29 donc `max` prend la valeur 29.
- Troisième itération, i = 3 : `max` est plus grand que -3 donc l'algorithme passe à l'itération suivante.
- i = 4 donc la boucle se termine et le programme affiche le résultat, c'est-à-dire 29.

Complexité de ces recherches

Complément

La complexité en temps de ces deux algorithmes est **O(*n*)** car il est nécessaire de parcourir tous les éléments de la liste pour trouver ce que l'on cherche.

À retenir

- Lorsque l'on cherche un élément dans une liste non-triée, il est nécessaire de parcourir un à un les éléments de celle-ci.

[cf. dhj3iVvE]

Exercice : Appliquer la notion

Question 1

Compléter le code suivant pour rechercher le prix maximum dans la liste de prix. Quel est ce maximum ?

```
1 const prices = [1, 39, 25, 112, 111, 30, 211, 300, 5, 67]
```

Question 2

Voici les dix derniers gagnants de la coupe départementale de Quidditch. Compléter le code suivant pour afficher les indices des valeurs de la liste égales à *Broom broom*. Combien de fois ont-ils gagné ?

```
1 const teams = ['Bois mort', 'Broom broom', 'Broom broom', 'Snek', 'Snek', 'Merlin
  FTW', 'Gandalf FTW', 'Merlin FTW', 'Broom broom', 'Bois mort']
2 const val = 'Broom broom' // La valeur à trouver
```

Rechercher dans des listes triées

[cf. y5lMxuf4]

Objectif

- Savoir rechercher efficacement une valeur dans une liste triée.

Mise en situation

La complexité d'une recherche dans une liste non-triée, à savoir **O(*n*)**, n'est pas très optimale. En effet, sur de grands volumes de données ou en cas de recherches fréquentes, le temps d'exécution ou les performances pourraient ne pas être au rendez-vous. Pour améliorer cela, le tri préalable de notre liste puis l'utilisation d'un algorithme de recherche dans une liste triée, peuvent grandement améliorer la complexité du problème.

Liste triée

Faire l'hypothèse qu'une liste est triée apporte beaucoup de faciliter à la recherche du développeur. Nous ne traitons ici que des listes triées dans l'ordre croissant seront considérées mais les algorithmes sont pratiquement identiques pour des listes triées dans l'ordre décroissant.

Recherche du maximum

 *Méthode*

L'algorithme dans ce cas est bien plus simple car il suffit d'une seule instruction pour trouver le maximum. La liste étant triée, il s'agit forcément du dernier élément. L'algorithme est donc en **O(1)** (par rapport au **O(*n*)** dans le cas d'une liste non triée).

```javascript
/** JavaScript : Recherche du maximum dans une liste triée. */
const sortedTemperatures = [-3, 5, 12, 29]

const max = sortedTemperatures[sortedTemperatures.length - 1]

console.log(max)
```

```python
"""Python : Recherche du maximum dans une liste triée. """
sorted_temperatures = [-3, 5, 12, 29]

max = sorted_temperatures[len(sorted_temperatures)-1]

print(max)
```

Recherche naïve des indices d'une valeur

 *Méthode*

Voici un algorithme naïf qui permet de retourner les indices d'une valeur donnée dans une liste triée. Cet algorithme est dit naïf car il est très intuitif mais n'est pas le plus optimisé dans ce cas.

```javascript
/** JavaScript : Recherche naïve dans une liste triée. */
const sortedTemperatures = [-3, 5, 12, 50, 50, 78, 94, 113, 129]
const val = 50
let i = 0
while (i < sortedTemperatures.length && sortedTemperatures[i] <= val) {
  if (sortedTemperatures[i] === val) {
    console.log(i)
  }
  i = i + 1
}
console.log('Fin')

```

```python
"""Python : Recherche naïve dans une liste triée. """
sorted_temperatures = [-3, 5, 12, 50, 50, 78, 94, 113, 129]
val = 50
i = 0
while (i < len(sorted_temperatures) and sorted_temperatures[i] <= val):
  if sorted_temperatures[i] == val:
    print(i)
  i = i + 1
print("Fin")
```

Déroulement pas à pas de la recherche naïve

Exemple

Pour mieux comprendre cet algorithme, il est intéressant de le faire tourner à la main avec l'exemple `sorted_temperatures = [-3, 5, 12, 50, 50, 78, 94, 113, 129]` ; `val = 50` :

- Initialement `i` = 0 et `sorted_temperatures[0]` = -3 < 50 donc on entre dans la boucle `while`.
- Première itération: `i` = 0, `sorted_temperatures[0]` = -3 est différent de 50 donc la condition `if` est fausse. On incrémente `i` pour finir l'itération.
- Deuxième itération : `i` = 1, `sorted_temperatures[1]` = 5 est différent de 50 donc la condition `if` est fausse. On incrémente `i` pour finir l'itération.
- Troisième itération : `i` = 2, `sorted_temperatures[2]` = 12 est différent de 50 donc la condition `if` est fausse. On incrémente `i` pour finir l'itération.
- Quatrième itération : `i` = 3, `sorted_temperatures[3]` = 50 est égal à 50 donc la condition `if` est vraie. Le programme affiche `i`. On incrémente `i` pour finir l'itération.
- Cinquième itération : `i` = 4, `sorted_temperatures[4]` = 50 est égal à 50 donc la condition `if` est vraie. Le programme affiche `i`. On incrémente `i` pour finir l'itération.
- `i` = 5 et `sorted_temperatures[5]` = 78 > 50, la condition `while` est fausse donc on sort de la boucle et le programme se termine.

Pour résumer, l'affichage du programme sera donc :

```
3
4
Fin
```

Recherche de l'indice d'une valeur : dichotomie

Complément

Ce problème consiste à renvoie l'indice (le position) d'une valeur donnée, si elle est présente dans une liste. Sachant que la liste est triée, l'algorithme réalise une dichotomie qui permet d'optimiser la recherche. Pour simplifier notre programme, on considère des listes sans doublons (aucune valeur n'est présente deux fois).

L'idée est la suivante : la recherche se fait dans une fenêtre (la liste dans notre cas). La fenêtre initiale est divisée en deux, si l'élément central est plus petit que la valeur, la fenêtre supérieure devient la nouvelle fenêtre de recherche. Sinon c'est la fenêtre inférieure qui le devient. Une fois la nouvelle fenêtre définie, le processus recommence. Ainsi, à chaque itération, la fenêtre de recherche est divisée par deux et l'algorithme converge très vite vers la bonne solution. Dans notre programme, la fenêtre de recherche est définie par les bornes ***a*** et ***b*** qui sont respectivement les bornes inférieure et supérieure.

```javascript
/** JavaScript : Dichotomie. */
const sortedTemperatures = [-3, 5, 12, 50, 78, 94, 113, 129]
const val = 50

let a = 0
let b = sortedTemperatures.length - 1
while (a <= b) {
  const m = Math.floor((a + b) / 2)
  if (sortedTemperatures[m] === val) {
    // on a trouvé v
    console.log(m)
    break
  } else if (sortedTemperatures[m] < val) {
    a = m + 1
  } else {
    b = m - 1
  }
}

```

```python
"""Python : Dichotomie. """
sorted_temperatures = [-3, 5, 12, 50, 78, 94, 113, 129]
val = 50

a = 0
b = len(sorted_temperatures) - 1
while a <= b:
  m = (a + b) // 2
  if sorted_temperatures[m] == val:
    # on a trouvé v
    print(m)
    break
  elif sorted_temperatures[m] < val:
    a = m + 1
  else:
    b = m - 1
```

Déroulement pas à pas de la dichotomie ? ***Exemple***

Pour mieux comprendre cet algorithme, il est intéressant de le faire tourner à la main avec l'exemple `sorted_temperatures = [-3, 5, 12, 50, 78, 94, 113, 129] ; val = 50`:

- Notre fenêtre est définie avec `a` = 0 et `b` = 7
- Première itération: `m` prend la valeur 3 et `sorted_temperatures[3]` = 50. 50 est supérieur à 12 donc la fenêtre inférieure est conservée, c'est-à-dire `b` = 2.
- Deuxième itération : `m` prend la valeur 1 et `sorted_temperatures[1]` = 5. 5 est inférieur à 12 donc la fenêtre supérieure est conservée, c'est-à-dire `a` = 2.
- Troisième itération : `m` prend la valeur 2 et `sorted_temperatures[2]` = 12. La valeur a été trouvée, on affiche la position et on sort de la boucle.

Complexité de ces recherches *Complément*

La complexité en temps de la dichotomie est **O(log(*n*))**. Cette complexité est bien meilleure que **O(*n*)** notamment pour des très grandes listes.

À retenir

- Une liste triée permet de réduire drastiquement les temps de recherche d'éléments.

[cf. 5yEvyOs0]

Exercice : Appliquer la notion

Question 1

Le programme suivant contient les performances des candidats dans un concours de lancer de javelots.

Compléter le programme pour trouver les positions de la valeur 46 en utilisant l'algorithme naïf de recherche. Donner l'affichage du programme.

```
1 const sortedPerf = [0, 6, 7, 16, 25, 30, 32, 38, 46, 46, 59, 70, 87, 93, 111]
2 const val = 46
```

Question 2

Modifier ce programme pour rechercher la valeur 42 au lieu de la valeur 46. Qu'affiche-t-il ?

Créer des paires (produit cartésien)

[cf. NKhxSOXM]

Objectif

- Connaître le fonctionnement d'un produit cartésien.

Mise en situation

Les bases de données sont aujourd'hui très utilisées, et cet usage est largement connu du grand public. Cependant une des opérations de base qui est effectuée sur des bases de données reste très peu connue : le produit cartésien. Cette opération mathématique permet en effet de regrouper deux ensembles de données et de donner la liste des combinaisons possibles entre les éléments de cet ensemble. Nous allons voir que c'est une opération simple à réaliser, mais relativement coûteuse en terme de complexité algorithmique.

Produit cartésien

Le produit cartésien est une opération mathématique entre deux ensembles. Faire le produit cartésien de l'ensemble **X** avec l'ensemble **Y** correspond à lister tous les couples possibles entre les éléments de **X** et ceux de **Y**. Cette opération est très utilisée dans les bases de données pour créer des relations entre les tables de celle-ci. Voici un exemple pour mieux comprendre cette opération mathématique :

```
persons = {"Jean", "Anne", "Maria"}
martialArts = {"Karaté", "Judo"}
ProduitCartésien(persons, martialArts) = [
 ("Jean", "Karaté"), ("Jean", "Judo"),
 ("Anne", "Karaté"), ("Anne", "Judo"),
 ("Maria", "Karaté"), ("Maria", "Judo")
]
```

Attention : il ne faut pas lister les couples possibles entre les éléments de X ou de Y **confondus**.

Par exemple, *("Jean", "Anne")* ne fait pas partie du produit cartésien car Jean et Anne font tous les deux partie de X.

Implémentation *Méthode*

Pour implémenter, le produit cartésien il faut deux boucles imbriquées. En français, on formulerait l'idée de l'algorithme de la manière suivante : pour chaque élément de **X** nous créons un couple avec chaque élément de **Y**.

```
/** JavaScript : Produit cartésien pour obtenir tous les menus possibles. */
const vegetables = ['Frites', 'Riz', 'Coquillettes']
const sauces = ['Pesto', 'Ketchup']

for (let i = 0; i < vegetables.length; i = i + 1) {
  for (let j = 0; j < sauces.length; j = j + 1) {
    console.log(vegetables[i], sauces[j])
  }
}

```

```python
"""Python : Produit cartésien pour obtenir tous les menus possibles. """
vegetables = ["Frites", "Riz", "Coquillettes"]
sauces = ["Pesto", "Ketchup"]

for i in range(len(vegetables)):
  for j in range(len(sauces)):
    print(vegetables[i],sauces[j])
```

Déroulement pas à pas

Exemple

Essayons un déroulement pas à pas avec X = `["Frites", "Riz", "Coquillettes"]` et Y = `["Pesto", "Ketchup"]`.

- Première itération principale, `i` = 0 :
  - Première itération secondaire, `j` = 0 : On affiche « *Frites Pesto* ».
  - Deuxième itération secondaire, `j` =1 : On affiche « *Frites Ketchup* ».
- Deuxième itération principale, `i` = 1 :
  - Première itération secondaire, `j` = 0 : On affiche « *Riz Pesto* ».
  - Deuxième itération secondaire, `j` =1 : On affiche « *Riz Ketchup* ».
- Troisième itération principale, `i` = 2 :
  - Première itération secondaire, `j` = 0 : On affiche « *Coquillettes Pesto* ».
  - Deuxième itération secondaire, `j` = 1 : On affiche « *Coquillettes Ketchup* ».
- Fin itération principale.

Tous les éléments du produit cartésien ont été affichés.

Complexité de produit cartésien

Complément

La complexité du produit cartésien est **O(*n*m*)**. ***n*** est le nombre d'éléments de X et ***m*** celui de Y. Cette complexité est visible grâce à l'imbrication des boucles.

À retenir

- Le produit cartésien est une opération mathématique utilisée notamment par les bases de données.
- Son implémentation est très simple grâce à deux boucles imbriquées.

[cf. 9J8CAZER]

Exercice : Appliquer la notion

Question

Compléter le code suivant pour afficher le produit cartésien de la liste `vehicles` avec la liste `colors`. Quel est le résultat ?

```
1 const vehicles = ['Vélo', 'Trottinette', 'Roller']
2 const colors = ['Rouge', 'Bleu']
```

Suite de Fibonacci

[cf. FHNaM3um]

Objectifs

- Connaître la suite de Fibonacci ;
- Savoir construire la suite de Fibonacci.

Mise en situation

Dans leur observation de la nature, certains mathématiciens ont découvert la récurrence d'une suite de nombres bien précise : la suite de Fibonacci.

Suite de Fibonacci

La suite de Fibonacci est un suite infinie de nombres, définie de la manière suivante.

```
Fibonacci[0] = 0
Fibonacci[1] = 1
```

Puis, pour tout entier `i` plus grand que 1 :

```
Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2]
```

- Pour `i` = 2, Fibonacci[2] = Fibonacci[1] + Fibonacci[0] = 1 + 0 = 1.
- Pour `i` = 3, Fibonacci[3] = Fibonacci[2] + Fibonacci[1] = 1 + 1 = 2.
- Pour `i` = 4, Fibonacci[4] = Fibonacci[3] + Fibonacci[2] = 2 + 1 = 3.
- ... Pour `i` = 10, Fibonacci[10] = Fibonacci[9] + Fibonacci[8] = 34 + 21 = 55.
- Et ainsi de suite.

F(0)	*F(1)*	*F(2)*	*F(3)*	*F(4)*	*F(5)*	*F(6)*	*F(7)*	*F(8)*	*F(9)*	*F(10)*	*F(11)*	...
0	1	1	2	3	5	8	13	21	34	55	89	...

Douze premiers termes de la suite de Fibonacci

 ***Remarque***

On retrouve, par exemple, cette suite dans la floraison de l'artichaut ou dans la disposition d'une pomme de pain. Cette présence énigmatique dans de nombreuses dispositions naturelles continuent d'intriguer les scientifiques et alimente encore aujourd'hui des recherches ou des créations artistiques.

Implémentation  ***Méthode***

Voici l'implémentation du calcul des `n` premiers termes la suite de Fibonacci en Python puis en JavaScript. L'implémentation est assez simple car il suffit d'itérer en additionnant systématiquement les deux valeurs précédentes.

```javascript
/** JavaScript : Suite de Fibonacci. */
const n = Number(prompt('Entrer un entier supérieur à 1 :')) // Par exemple, 4

const fibo = new Array(n)
fibo[0] = 0
fibo[1] = 1

for (let i = 2; i < n; i = i + 1) {
  fibo[i] = fibo[i - 1] + fibo[i - 2]
}

console.log(fibo)

```

```python
"""Python : Suite de Fibonacci. """
n = int(input("Entrer un entier supérieur à 1:"))  #  Par exemple, 4

fibo = [0]*(n)
fibo[0] = 0
fibo[1] = 1

for i in range(2,n):
  fibo[i] = fibo[i-1] + fibo[i-2]

print(fibo)
```

Déroulement pas à pas

Exemple

Déroulons l'algorithme pas à pas pour `n` = 5. Tout d'abord, une liste vide de taille 5 est créée et les éléments 0 et 1 sont positionnés sur les deux premières positions, fibo = **[0, 1, _, _, _]**. Ensuite, on entre dans la boucle :

- Première itération, `i` = 2 : fibo[2] = fibo[1] + fibo[0] = 1. On a donc fibo = **[0, 1, 1, _, _]**.
- Deuxième itération, `i` = 3 : fibo[3] = fibo[2] + fibo[1] = 2. On a donc fibo = **[0, 1, 1, 2, _]**.
- Troisième itération, `i` = 4 : fibo[4] = fibo[3] + fibo[2] = 3. On a donc fibo = **[0, 1, 1, 2, 3]**.
- `i` = `n` donc la boucle se termine.

La liste des cinq premiers termes de la suite est affichée dans la console.

Complexité

Complément

La complexité en temps de cet algorithme est **O(*n*)**. On peut voir très clairement ce qui explique cette complexité car on voit deux boucles simples. On pourrait être plus précis en disant **O(2 * *n*)** mais cette complexité est considérée comme équivalente à **O(*n*)** car on souhaite simplement un ordre de grandeur.

À retenir

- La suite de Fibonacci est une suite de nombres qu'on retrouve souvent dans la nature.
- L'implémentation du calcul de ses termes est très simple.

[cf. 8jtwn7mO]

Exercice : Appliquer la notion

Question

Modifier et compléter le code suivant pour afficher les quinze premiers termes de la suite de Fibonacci. Quels sont-ils ?

```
const n = -1

const fibo = new Array(n)
fibo[0] = 0
fibo[1] = 1
```

Indice :

Les deux premiers termes sont déjà *« calculés »*. La boucle de calcul ne doit calculer que les treize termes restants.

Trier des listes (tri à bulles)

[cf. xF3g6VyO]

Objectif

- Connaître le fonctionnement d'un algorithme de tri.

Mise en situation

Les tris forment une famille d'algorithmes très classiques, le tri étant très utile pour optimiser les algorithmes de recherche. Mais l'optimisation d'un algorithme de tri lui même est aussi très intéressant, d'autant plus qu'il en existe plusieurs différents. C'est un cas d'école souvent étudié pour comparer la complexité des différents algorithmes. Cependant aujourd'hui, la quasi-totalité des langages de programmation implémentent directement des fonctionnalités de tri en interne, ce qui fait que les algorithmes de tri sont désormais plus de l'ordre de la culture générale que du besoin pratique.

Algorithmes de tri

Il existe de nombreux algorithmes de tri. Chacun ont leurs avantages et leurs inconvénients. Certains sont plus efficaces sur des listes peu mélangées (juste quelques éléments dans le mauvais ordre) d'autres seront plus lents mais ont une complexité en temps qui est moindre. Voici une liste de quelques uns des tris les plus connus :

- Tri rapide,
- Tri fusion,
- Tri par tas,
- Tri à bulles,
- Tri par insertion,
- etc.

Tri à bulles *Méthode*

L'idée de ce tri est de faire remonter itérations après itérations les valeurs hautes vers la fin de la liste. On peut voir un analogie avec les gouttes d'huiles remontant petit à petit vers la surface de l'eau. Après la première itération, la valeur maximale sera à la fin de la liste. En parallèle, il est très probable que d'autres valeurs hautes aient avancé légèrement en direction de la fin de la liste. Voici les implémentations JavaScript et Python pour mieux comprendre cela :

```
/** JavaScript : tri à bulles d'une liste d'âges. */
const ages = [17, 1, 28, 5]

for (let i = ages.length - 1; i > 0; i = i - 1) {
  for (let j = 0; j < i; j = j + 1) {
    if (ages[j + 1] < ages[j]) {
      // Echanger les deux valeurs
      const temp = ages[j + 1]
      ages[j + 1] = ages[j]
      ages[j] = temp
```

```
    }
  }
}

console.log(ages)

```

```
"""Python : Tri à bulles d'une liste d'âges. """
ages = [17, 1, 28, 5]

for i in range(len(ages)-1, 0, -1):
  for j in range(0, i):
    if ages[j+1] < ages[j]:
      # Echanger les deux valeurs
      temp = ages[j+1]
      ages[j+1] = ages[j]
      ages[j] = temp

print(ages)
```

Déroulement pas à pas ? *Exemple*

Sachant qu'il y a deux boucles imbriquées, on prend un exemple très simple qui permettra de comprendre rapidement le fonctionnement de ce tri : ages = `[17, 1, 28, 5]`

- Première itération principale, `i` = 3 :
  - Première itération secondaire, `j` = 0 : ages[0] (= 17) > ages[1] (= 1) donc on échange les deux valeurs et la liste devient `[1, 17, 28, 5]`.
  - Deuxième itération secondaire, `j` = 1 : ages[1] (= 17) < ages[2] (= 28) donc la liste reste `[1, 17, 28, 5]`.
  - Troisième itération secondaire, `j` = 2 : ages[2] (= 28) > ages[3] (= 5) donc on échange les deux valeurs et la liste devient `[1, 17, 5, 28]`.
  - `j` = 3 = `i` donc la boucle secondaire se termine.
- Deuxième itération principale, `i` = 2 :
  - Première itération secondaire, `j` = 0 : ages[0] (= 1) < ages[1] (= 17) donc la liste reste `[1, 17, 5, 28]`.
  - Deuxième itération secondaire, `j` = 1 : ages[1] (= 17) > ages[2] (= 5) donc on échange les deux valeurs et la liste devient `[1, 5, 17, 28]`.
  - `j` = 2 = `i` donc la boucle secondaire se termine.
- Troisième itération principale, `i` = 1 :
  - Première itération secondaire, `j` = 0 : ages[0] (= 1) < ages[1] (= 5) donc la liste reste `[1, 5, 17, 28]`.
  - `j` = 1 = `i` donc la boucle secondaire se termine.
- `i` = 0 donc la boucle principale se termine

Dans ce déroulement, on constate nettement le concept de bulles dans la première itération principale : `28` remonte « à la surface » et finit en dernière position tandis que `17` remonte légèrement et atteindra sa position finale à l'itération principale suivante.

Complexité du tri à bulles *Complément*

La complexité en temps de ce tri est **O(n^2)**. On peut voir très clairement ce qui explique cette complexité grâce à la boucle imbriquée. Il ne fait pas partir des tris les plus rapides tel que le tri fusion qui est en **O(n*log(n))**.

Ne jamais implémenter un tri soi-même *Attention*

Il est important pour un développeur de connaître des algorithmes de tri car ces algorithmes mettent en œuvre de manière très élégante des concepts algorithmiques majeurs. Ils font partie de la culture générale des développeurs.

En revanche, il faut jamais implémenter ce type d'algorithme soi-même.

La quasi-totalité des langages possède des fonctions de tri déjà implémentées. Celles-ci ne proposent même pas à l'utilisateur de choisir le tri à utiliser car elles sont optimisées pour utiliser le tri optimal selon la liste fournie. Voici comment faire en JavaScript et en Python :

```javascript
const ages = [18, 6, 82]
ages.sort()
```

```python
ages = [18, 6, 82]
ages.sort()
```

À retenir

- Il existe de nombreux algorithmes de tri avec chacun leurs avantages.
- Il ne faut pas implémenter d'algorithme de tri soi-même car les langages ont déjà des fonctions de tri parfaitement optimisées.

[cf. FQ422dF7]

Exercice : Appliquer la notion

Question

Compléter le code suivant pour afficher les scores qu'ont fait une bande d'amis lors d'une partie de bowling, triés grâce au tri à bulle. Quel est le résultat ?

```
1 const scores = [58, 13, 29, 100, 203, 1, 5, 13, 56, 33, 123]
```

Essentiel

[cf. 4LtdOzxJ]

Il est important d'avoir connaissance des algorithmes principaux permettant de réaliser les opérations de traitement de données les plus courantes. La recherche, le tri ou le croisement de données est aujourd'hui monnaie courante, et il serait contre-productif de réinventer les algorithmes à chaque fois. Ces algorithmes sont tellement utiles qu'ils sont en général directement implémentés dans les langages de programmation. Cependant il est intéressant d'en avoir connaissance, pour la culture informatique, mais aussi car cela permet de les adapter légèrement lorsque les besoins sont proches.

Quiz

Exercice 1 : Quiz - Culture

Exercice

Laquelle de ces propositions correspond au produit cartésien de `['Robert', 'Anne', 'Bob']` et `['Alice', 'Charlie']` ?

- O `[('Robert', 'Alice'), ('Robert', 'Charlie'), ('Anne', 'Alice'), ('Anne', 'Charlie'), ('Bob', 'Alice'), ('Bob', 'Charlie')]`
- O `[('Robert', 'Anne'), ('Robert', 'Bob), ('Anne', 'Bob'), ('Alice', 'Charlie')]`
- O `[('Robert', 'Anne'), ('Robert', 'Bob), ('Anne', 'Bob'), ('Alice', 'Charlie'), ('Robert', 'Robert'), ('Anne', 'Anne), ('Bob', 'Bob'), ('Alice', 'Alice'), ('Charlie', 'Charlie')]`
- O `[('Robert', 'Robert'), ('Anne', 'Anne), ('Bob', 'Bob'), ('Alice', 'Alice'), ('Charlie', 'Charlie')]`

Exercice

Les algorithmes de tri prêts à l'endroit sont regroupés au sein de **bibliothèques**. Ces bibliothèques sont souvent open source (tout le monde peut notamment consulter le code des implémentations).

Quelles sont les raisons principales justifiant l'utilisation de bibliothèques open source de fonctions plutôt que d'implémentation personnelles ou propriétaires ?

- ☐ Optimisation
- ☐ Économie de temps
- ☐ Standardisation

Exercice

Il existe un unique algorithme de tri qui s'est distingué comme étant le meilleur quelle que soit la liste à trier.

- O Vrai
- O Faux

Exercice

Quel est le nom de l'algorithme **optimisé** à utiliser pour vérifier l'existence d'une valeur dans une liste triée ?

- ○ Recherche à bulles
- ○ Balayage logarithmique
- ○ Dichotomie
- ○ Trachéotomie

Exercice 6 : Quiz - Méthode

Exercice

Nous disposons d'une liste non triée de 5 entiers. Combien faut-il inspecter d'éléments pour trouver le maximum ?

Exercice

Nous disposons d'une liste **triée** de 5 entiers. Combien faut-il inspecter d'éléments pour trouver le maximum ?

Exercice

Une implémentation du calcul des 10 premiers termes de la suite de Fibonacci a produit le résultat suivant :

```
[0, 1, 1, 2, 3, 5, 9, 13, 21, 34]
```

Une erreur s'est glissée dans l'algorithme : l'élément 9 ne fait pas partie de la suite. Quelle devrait-être la bonne valeur ?

Exercice 10 : Quiz - Code

Exercice

Donner l'instruction JavaScript pour trier la liste `letter_list`.

Contenus annexes

1. Complexité algorithmique

[cf. vZWfpVOC]

Objectif

- Découvrir la notion de complexité algorithmique.

Mise en situation

Lorsqu'un développeur écrit un programme, il lui est nécessaire d'estimer si ce programme est coûteux. Coûteux cela signifie par exemple qu'il nécessite beaucoup de calculs de la part du processeur. Cela peut aussi signifier qu'il nécessite beaucoup d'espace mémoire.

Savoir calculer ce coût permet de comparer plusieurs solutions alternatives entre elles. Cela permet aussi de savoir si un programme peut « passer à l'échelle » c'est à dire continuer de fonctionner correctement si on le confronte à la réalité.

Par exemple si un développeur a écrit un programme permettant de vérifier une empreinte de carte de paiement, mais que cette vérification prend plusieurs minutes ou nécessite un ordinateur doté d'une mémoire très importante, il ne sera pas possible de l'utiliser pour du paiement en ligne.

Compter le nombre d'opérations pour résoudre un problème ? *Exemple*

Pour créer la table de 2 des **n** premiers entiers, il faut réaliser **n** calculs de type multiplication.

De manière générale, on cherche à estimer de manière théorique le nombre d'opérations à effectuer pour résoudre un problème : c'est ici que l'on parle de **complexité** d'un problème.

Complexité algorithmique *Définition*

La **complexité algorithmique** est une estimation du nombre d'opérations élémentaires pour résoudre un problème en fonction de la taille des données d'entrée, notée **n**.

O() § *Syntaxe*

On note O() **l'ordre de grandeur** de la complexité d'un algorithme.

- O(n) signifie que le nombre d'opérations est de l'ordre de n (si on a 1000 données en entrée, on s'attend à environ 1000 opérations).
- $O(n^2)$ signifie que le nombre d'opérations est de l'ordre de n au carré (si on a 1000 données en entrée, on s'attend à environ 1 000 000 d'opérations).

Exemple simple

Exemple

Ce code fait la somme d'une liste d'entiers.

```
function somme(number_list) {
  let res = 0;
  for (var i = 0; i < number_list.length; i++) {
      res += number_list[i];
  }
  return res;
}
```

Étant donné **n** la taille de la liste, il serait nécessaire d'effectuer **n** opérations d'addition pour obtenir la somme finale. Ici, on dira que la complexité de l'algorithme est de l'ordre de **n** et on la notera **O(n)**.

Pourquoi utiliser la complexité algorithmique ?

Remarque

La complexité algorithmique peut se voir comme l'estimation du "temps d'exécution" d'un algorithme en fonction de la taille des données d'entrée.

On suppose pour ce faire que les opérations de base (affectation, addition, etc.) ont le même temps d'exécution : estimer le temps d'exécution revient alors à estimer le nombre d'opérations de base.

L'exécution reste proportionnelle à la taille des données

Attention

La complexité ne change pas en fonction de la taille des données. En revanche, l'exécution sera évidemment plus longue si la taille des données augmente. Ainsi, pour 100 données, un algorithme en **O(n)** effectuera de l'ordre de 100 opérations. Pour 1000 données il en effectuera de l'ordre de 1000.

Estimer rapidement la complexité

Méthode

La complexité permet d'avoir une idée grossière de la durée d'un algorithme. Par exemple, si on détermine que pour une liste de **n** éléments, il faut effectuer **2×n** opérations, on simplifiera en disant que la complexité est **de l'ordre** de **O(n)**.

La raison principale est que les constantes ne changent pas fondamentalement l'ordre de grandeur de la complexité. Par exemple, $\mathbf{O(n^2)}$ et $\mathbf{O(2{\times}n^2)}$ sont très proches, et c'est surtout la valeur de **n** qui influencera la durée finale.

Complexités usuelles des algorithmes

Fondamental

On retrouve très régulièrement les complexités suivantes :

- **O(1)** : si le nombre d'opérations ne dépend pas de la taille des données. Par exemple, afficher le premier élément d'une liste.
- **O(n)** : cette complexité se retrouve souvent quand on a besoin de parcourir les éléments d'une liste, par exemple pour calculer la somme des éléments.
- $\mathbf{O(n^2)}$: par exemple, l'algorithme naïf pour trier une liste de nombre nécessite, pour chaque élément de la liste, de parcourir l'ensemble de la liste.
- **O(n×m)** : par exemple, pour trouver les nombres communs de deux listes de taille **n** et **m**, il faut pour chaque nombre de la première liste parcourir l'ensemble de la deuxième liste pour vérifier s'il existe.

Complément

La complexité algorithmique permet de dire, qu'à tailles de données égales, un algorithme en **O(n)** terminera bien avant un second algorithme dont la complexité est **O(n^2)**.

Ainsi, pour 100 données, le premier algorithme effectuera environ 100 opérations de bases et le second 10 000 opérations. On comprend que, si la taille des données se compte en millions (ou même en milliards), le second algorithme sera beaucoup plus coûteux que le premier.

Comparaison des complexités

n	O(n)	O(n^2)
10	10	100
100	100	10 000
1 000	1 000	1 000 000
10 000	10 000	100 000 000

Complexité algorithmique en espace

Complément

Le temps de calcul est important mais l'espace nécessaire au calcul l'est aussi. La mémoire d'un ordinateur n'est pas infinie : il est utile d'estimer l'espace mémoire nécessaire pour exécuter un algorithme.

La complexité décrite jusqu'ici se nomme complexité en **temps**. Ici, on parle complexité algorithmique en **espace**. La notation sera toujours la même : **O(1)**, **O(n)**, **O(n^2)**, etc.

Complexité et machine de Turing

Complément

Le concept de complexité algorithmique est étroitement lié aux machines de Turing, une des bases théorique des ordinateurs. Elles ont introduit une nouvelle définition de la notion de calcul qui permet également de théoriser sa complexité. Ainsi, aujourd'hui, on peut classer les problèmes de calcul selon la complexité des algorithmes pouvant les résoudre.

On peut distinguer, très schématiquement, trois grands types de problèmes :

- Problème dont l'algorithme finira en un temps polynomial : **O(n)**, **O(n^2)**, etc.
- Problème dont l'algorithme finira en un temps exponentiel : **O(2^n)**, **O(n!)**, etc. On évite que ce type d'algorithme car il suffit que **n** augmente un tout petit peu pour que le temps d'exécution se compte en mois, en années voire en siècles.
- Problème dont il n'existe pas de solution (dits indécidables).

À retenir

- Il est possible d'attribuer une complexité à un algorithme.
- Estimer la complexité d'un programme permet d'avoir un ordre de grandeur de son temps d'exécution.
- Comparer la complexité de deux algorithmes permet de déterminer lequel est le plus optimisé pour répondre à un problème donné.

[cf. vIbrsmA4]

Crédits des ressources

Douze premiers termes de la suite de Fibonacci p. 15
http://creativecommons.org/licenses/by-sa/3.0/fr/, Stéphane Crozat, Marc Damie