

Applications de la récursivité : Tris de tableaux

1. Rappels sur les algorithmes de tris vus en classe de première

Principe du tri par insertion

Le principe du tri par insertion est le suivant : au moment où on considère un élément du tableau à trier, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.

On peut voir sur l'animation suivante extraite de wikipedia :

6 5 3 1 8 7 2 4

La complexité du tri par insertion est quadratique (en $O(n^2)$) dans le pire cas et en moyenne, et linéaire (en $O(n)$) dans le meilleur cas (tableau presque trié). C'est donc un tri dont la vitesse d'exécution dépendra fortement de la situation initiale.

 Le code en Python



Principe du tri par sélection

Sur un tableau de n éléments, le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

On peut voir sur l'animation suivante extraite de [wikipedia](#) :

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Cet algorithme de tri, simple à comprendre, est considéré comme mauvais car sa complexité en temps est en $O(n^2)$, que ce soit dans le pire des cas ou bien le meilleur des cas (même pour un tableau déjà trié il faudra faire toutes les comparaisons).

 Le code en Python



2. Le tri fusion

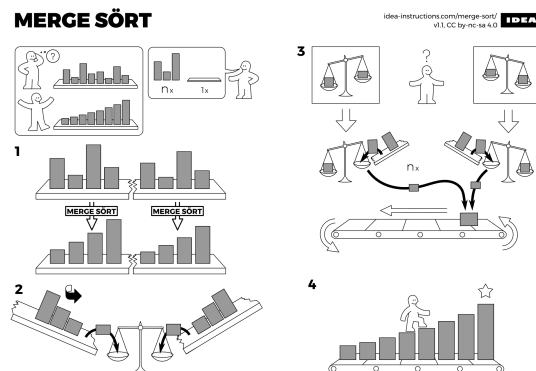


Principe du Tri Fusion (Merge Sort)

Le principe du **tri fusion**, est de séparer un tableau à trier en **deux sous-tableaux** qu'on triera récursivement (ou itérativement mais ce ne sera pas le cas dans ce cours) de nouveau par tri fusion.

Une fois les sous-tableaux triés, il faudra **fusionner** ces deux sous-tableaux en une seule identité.

Cet algorithme est du type **diviser pour régner** (*Divide and Conquer*) : on sépare la tâche à priori difficile en deux tâches plus simples (ici, il s'agit de diminuer la taille des tableaux). Il a été inventé par **John Von Neumann en 1945**.



? Fonction de fusion

Exercice

La principale difficulté dans cet algorithme est de créer une fonction fusionnant les deux tableaux triés.

1. Avant commencer, regardez la vidéo suivante :

2. De combien de variables compteurs avons nous besoins ?
3. Quel type de boucle allons nous utiliser ?
4. Que se passe-t-il dans la vidéo quand atteint la valeur ?
5. Rédiger en langage naturel l'algorithme fusionnant deux tableaux triés.

6. Implémenter une fonction `fusion(t1, t2)` qui prend en argument deux tableaux supposés triés (on ne vérifiera pas), et qui renvoie le tableau trié contenant tous les éléments des deux tableaux `t1` et `t2`. On pourra utiliser les tests suivants pour vérifier que la fonction est correcte :

” Les tests



Solution

1. C'est beau, c'est fait avec [manim](#).
2. En réalité il n'y a que deux compteurs : `i` et `j`. En effet `i` est en permanence égal à `j`.
3. On peut utiliser une boucle `Pour`, vu que le nombre d'itérations est connu au départ : il s'agit de la somme des longueurs des deux tableaux. Mais il est aussi tout à fait possible d'utiliser une boucle `Tant que`, puisque l'algorithme se poursuit tant qu'un des deux compteurs n'a pas atteint sa position finale.
4. Quand `i` atteint la position 3 (soit la longueur du deuxième tableau), il ne reste plus qu'à compléter avec les valeurs restantes du premier tableau.
5. Différentes versions :

Version `Pour` sans initialisation du tableau final

```
fonction fusion(t1, t2)
  tf <- tableau vide
  i <- 0
  j <- 0
  n1 <- longueur de t1
  n2 <- longueur de t2
  Pour k allant de 0 à n1+n2-1
    Si i<n1 et j<n2
```

? Tri Fusion Récursif

Exercice

Une fois la fonction `fusion` codée, l'algorithme de tri par fusion est simple :

1. Regardez la vidéo suivante :
2. Quel est le cas de base ?
3. Quel est le cas récursif ?
4. Implémenter une fonction `triFusion(t)` qui prend en argument un tableau et renvoie une copie triée de ce tableau.

Solution

A venir !

? Complexié en temps

Exercice

1. Copiez-collez le code ci-dessous permettant d'utiliser le décorateur `@timeit` :

```
import time

def timeit(method):

    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)
        te = time.time()
        print(f" {method.__name__} ( {args}, {kw} ) {te-ts}" )
        return result

    return timed
```

⚠ Décorateur `@timeit`

pour utiliser le décorateur, on le place dans la ligne précédant la définition de la fonction qu'on veut décorer. Par exemple :

```
@timeit
def tri_insertion(tab) :
    ...
```

Chaque fois que la fonction `tri_insertion` sera appelée, le décorateur sera appliqué et exécutera la fonction `timed`, qui calcule le temps d'exécution de la fonction décorée.

Il faudra être attentif à son utilisation **dans le cas des fonctions récursives** ! (Je vous laisse constater par vous même le problème rencontré.)

2. Créer une fonction `genereTab(n)` qui crée un tableau de taille `n` d'entiers aléatoires compris entre `0` et `1000`.
3. Créer à l'aide de toutes les fonctions précédentes une fonction `testeTemps(n)` qui compare les temps d'exécution des différents tris pour `n` valant `1000`, `2000`, `4000`, `8000`, `16000`. **Attention**, il faudra effectuer à chaque fois les tests sur le même tableau, et donc créer des copies du tableau original avant de le trier (on peut utiliser la fonction `deepcopy` du module `copy`)
4. Que peut-on en conclure quand à la complexité en temps du tri fusion ?

Solution

A venir !

Complexités du tri fusion

La version la plus simple du tri fusion sur les tableaux a une efficacité comparable au tri rapide, mais elle n'opère pas en place : une zone temporaire de données supplémentaire de taille égale à celle de l'entrée est nécessaire (des versions plus complexes peuvent être effectuées sur place mais sont moins rapides). La complexité en temps est en $O(n \log n)$ dans tous les cas, et la complexité en espace est en général en $O(n)$.

3. Plus vite ! QuickSort ! (Hors programme)

Principe du QuickSort

