COURS 4 : TRIS DE TABLEAUX

Compétences attendues :

- Connaitre les méthodes de tris les plus connues
- S'interroger sur l'efficacité algorithmique temporelle de ces algorithmes
- Distinguer par leurs complexités deux algorithmes résolvant un même problème

1. INTRODUCTION

Un algorithme de tri est un algorithme qui permet d'**organiser une collection d'objets** selon un **ordre déterminé**. Le tri permet notamment de **faciliter les recherches** ultérieures d'un élément dans une liste (recherche dichotomique).

On s'intéresse ici à des méthodes de tri d'une liste de valeurs numériques. Celle-ci est implémentée sous la forme d'un tableau à une dimension.

La plupart des algorithmes de tri sont fondés sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. Nous appellerons tri comparatif un tel tri.

La complexité de l'algorithme *a alors le même ordre de grandeur que le* ***nombre de comparaisons*** *entre les données faites par l'algorithme.*

Remarque : Pour trier des chaînes de caractères (mots), il suffit d'associer une valeur numérique à chaque caractère (code ASCII par exemple).

On se limite dans le cadre du programme aux cas détaillés du **tri par insertion**, du **tri rapide** et du **tri fusion**.

2. LES METHODES « SIMPLES »

2.1. Tri par sélection

Le tri par sélection est un des algorithmes de tri les plus triviaux. On recherche le plus grand élément que l'on va replacer à sa position finale, c'est-à-dire en dernière position.

Puis on recherche le second plus grand élément que l'on va placer en avant-dernière position, etc., jusqu'à ce que le tableau soit entièrement trié.

Données : tableau *T* entre les indices 1 et *n*.

PYTHON

```
def echange(l,i,j):
    # echange 2 valeurs d'une liste
    l[i],l[j] = l[j], l[i]
def tri_selection(l):
```

Complexité

- Meilleur des cas : $O(n^2)$ quand le tableau est déjà trié.
- Pire cas : $O(n^2)$ quand le tableau est trié en ordre inverse.
- En moyenne : $O(n^2)$

Ce n'est donc pas un tri très efficace, mais en revanche c'est un tri très simple.

2.2. Tri à bulles (Bubble sort)

Le tri à bulles est un algorithme de tri qui consiste à faire remonter progressivement les plus petits éléments d'une liste, comme les bulles d'air remontent à la surface d'un liquide.

- L'algorithme parcourt la liste, et compare les couples d'éléments successifs en **commençant par la fin de la liste**
- Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés.
- Après chaque parcours complet de la liste, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que la liste est triée : l'algorithme peut s'arrêter.

PYTHON

```
def echange(l,i,j):
    # echange 2 valeurs d'une liste
    l[i],l[j] = l[j], l[i]
def tri_abulles(l):
```

Cet algorithme est souvent enseigné en tant qu'exemple algorithmique.

Cependant, il présente une complexité en $O(n^2)$ dans le pire des cas (où n est la longueur de la liste), ce qui le classe parmi les mauvais algorithmes de tri. Il n'est donc quasiment pas utilisé en pratique.

Complexité

- Meilleur cas : $O(n)$ quand le tableau est trié.
- Pire des cas : $O(n^2)$ quand le tableau est trié en ordre inverse $(n-1)\cdot(n-1) = O(n^2)$

2.3. Tri par insertion

Exemple 1 : Tri d'un jeu de cartes

Problème : Comment classer les cartes d'un jeu ?

Soit un paquet de « n » cartes.

On prend la première dans une main.

On saisie la seconde carte et on l'insère avant ou après la première selon le cas.

A l'étape « i », la $i^{ème}$ carte est insérée à sa place dans le paquet déjà trié.

Pour cela, on peut :

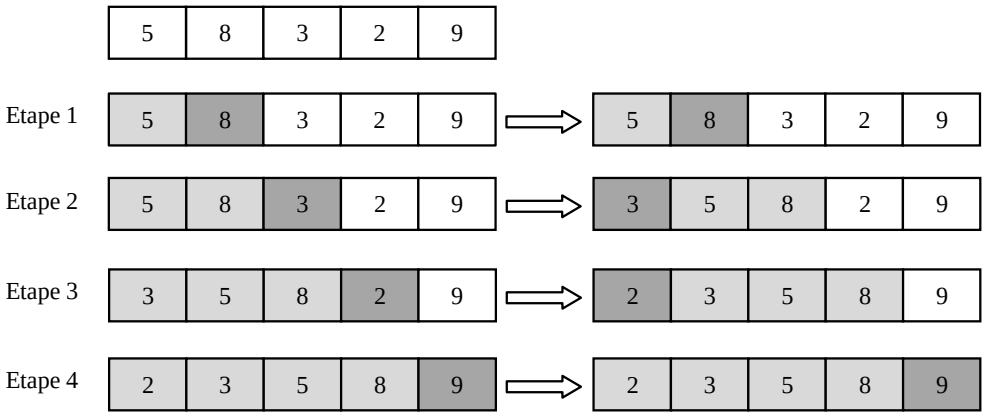
- **méthode 1 :** partir du début du tas déjà trié et s'arrêter lorsque l'on rencontre une carte plus grande que la $i^{ème}$,
- **méthode 2 :** partir de la fin du tas déjà trié, et s'arrêter si l'on rencontre une carte plus petite que la $i^{ème}$.

Le paquet contient alors « i » cartes triées.

On procède ainsi de suite jusqu'à la dernière carte.

Exemple 2 : Tri de valeurs numériques

Problème : Comment trier une liste de nombres ?



Algorithme : Tri par insertion (fonction **Tri_insertion)**

Données : T : un tableau de valeurs num [1..n] ;
Résultat : le tableau T trié par ordre croissant

PYTHO N

```
def Tri_insertion(          ) :
```

L'analyse de la complexité de l'algorithme peut se faire par l'étude du nombre de comparaisons à effectuer.

Complexité :

Meilleur des cas : le tableau est déjà trié. Il y a donc n – 1 comparaisons à effectuer. La complexité est donc de classe **linéaire** : *C(n)=O(n).*

Dans le **pire des cas**, le tableau est trié à l'envers. Il y a alors une comparaison à effectuer à la première étape, puis deux, … puis n-1. On en déduit donc un nombre total de $\frac{n\times(n-1)}{2}$ comparaisons. ***La complexité est donc de classe quadratique : C(n)=O(n^2).***

Algorithme : Tri par insertion méthode2 (**Tri_inser2)**

Données : T : un tableau de valeurs num [1..n] ;
Résultat : le tableau T trié par ordre croissant

PYTHO N

```
def Tri_inser2(          ) :
```

3. LE TRI RAPIDE « QUICK SORT »

L'algorithme fait parti de la catégorie des algorithmes « diviser pour régner ».

A chaque appel de la fonction de tri, le nombre de données à traiter est diminué de un. C'est-à-dire que l'on ne traite plus l'élément appelé « pivot » dans les appels de fonction ultérieurs, il est placé à sa place définitive dans le tableau.

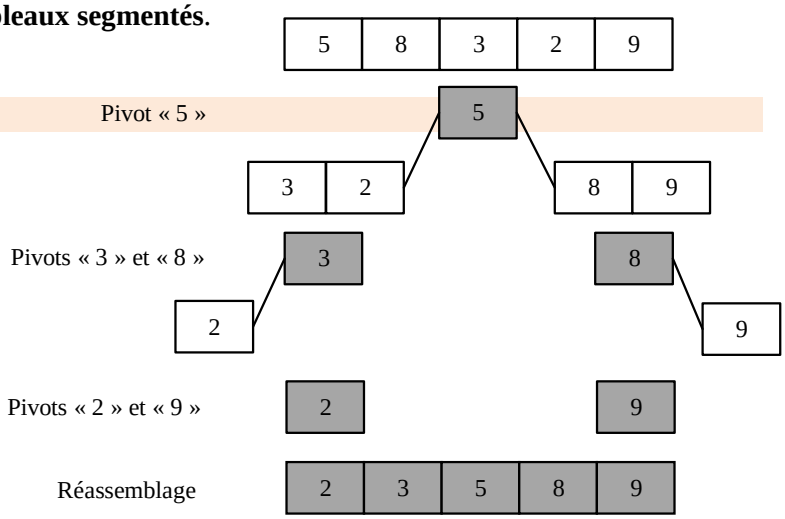
Le tableau de valeurs est ensuite segmenté en deux parties :

- dans un premier tableau, toutes les valeurs numériques sont inférieures au « pivot »,
- dans un second tableau, toutes valeurs numériques sont supérieures au « pivot ».

L'appel de la fonction de tri est **récursif** sur les **tableaux segmentés**.

Exemple 3 : Tri de valeurs numériques

On peut par exemple choisir le premier élément du tableau comme « pivot » :



Le « coût » temporel de l'algorithme de tri est principalement donné par des opérations de comparaison sur les éléments à trier. On raisonne donc sur le nombre de données à traiter pour l'analyse de la complexité de l'algorithme.

➔ Dans le **pire des cas**, un des deux segments est vide à chaque appel de la fonction de tri. Cela arrive lorsque le tableau est déjà trié. Le nombre de données à traiter pour le $i^{ème}$ appel, est n – i + 1.

Le nombre total pour n appels de fonction est donc $\frac{n\times(n+1)}{2}$.

On peut aussi écrire une relation de récurrence du type $C(n) = C(n-1) + n - 1$

La complexité est donc de classe quadratique $C(n) = O(n^2)$.

➔ Dans le **meilleur des cas**, les deux segments sont de taille égale. Pour un nombre de données à traiter n, chacun des segments suivant a donc au plus (n-1)/2 éléments (on retire le pivot). On répète ainsi la segmentation des tableaux jusqu'à arriver au plus à un seul élément.

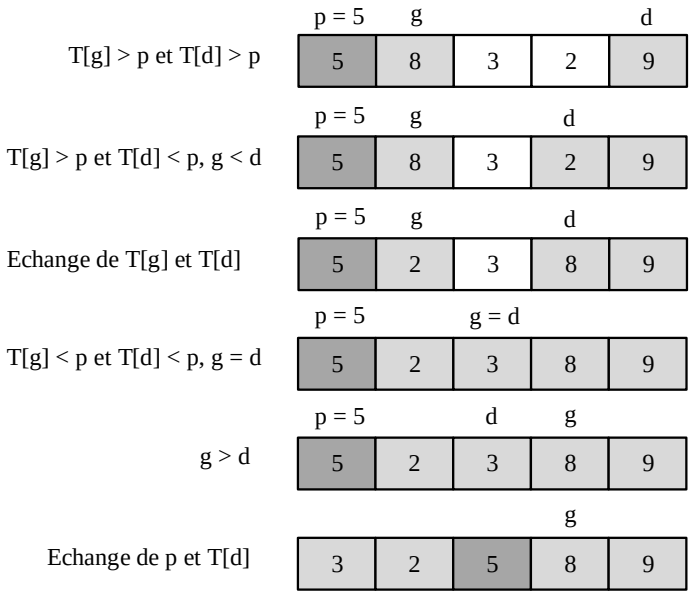
On peut écrire une relation de récurrence du type

$C(n) = 2\times C((n - 1) / 2) + n - 1.$

La complexité est donc de classe quasi linéaire

$C(n)=O(n\times ln(n)).$

Remarques : Certains algorithmes de tri rapide prennent pour « pivot » le dernier élément, la valeur moyenne du premier et du dernier, ou un positionnement aléatoire dans le tableau. Pour se placer dans le meilleur des cas pour chaque segment de tableau, il faut prendre pour pivot la ***valeur médiane*** *du tableau de valeurs. Le problème est que cette recherche de pivot idéal a aussi un « coût ».*



Le pivot a sa place définitive
Les éléments à sa gauche sont plus petits ou égaux
Les éléments à sa droite sont plus grands

On écrit tout d'abord l'algorithme effectuant la **segmentation du tableau**

- Le pivot (1er élément du tableau) est mis à sa place définitive,
- Pour des indices inférieurs, toutes les valeurs sont plus petites ou égales,
- Pour des indices supérieurs, toutes les valeurs sont plus grandes.

Algorithme : Segmentation (fonction **Segmente)**

Données : T : un tableau de valeurs num [1..n] ;

i, j : les indices de début et de fin de la segmentation à effectuer

Résultat : le tableau T « segmenté » avec le pivot à sa place définitive, l'indice de la place du pivot

PYTHO N

```
def Segmente( T ,i ,j  ) :
```

Complexité :

Le nombre de comparaisons du type `T[d]>p` et `T[g]<=p` est égal dans meilleur des cas à n – 1.

La complexité de cet algorithme est donc au mieux de classe linéaire : $C(n)=O(n)$.
Dans le pire des cas, tous les éléments sont identiques. La complexité est alors quadratique : $C(n)=O(n^2)$.

Il existe des variantes de cet algorithme qui permettent d'avoir une complexité en $O(n \times ln(n))$ même lorsque tous les éléments sont identiques.

Ecrire l'algorithme **récursif** de tri rapide (on utilisera la méthode de segmentation)

Algorithme : Tri rapide (fonction **Tri_rapide)**

Données : T : un tableau de valeurs num [1..n] ;

i, j : les indices de début et de fin de la segmentation à effectuer

Résultat : le tableau T trié entre les indices i et j compris

PYTHO N

```
def Tri_rapide(          ) :
```

*Cette méthode de tri est **très efficace lorsque les données sont distinctes et non ordonnées**. La complexité est alors globalement en **$O(n \times ln(n))$.** Lorsque le nombre de données devient petit (<15) lors des appels récursifs de la fonction de tri, on peut avantageusement le **remplacer par un tri par insertion** dont la complexité est linéaire lorsque les données sont triées ou presque.*

Algorithme : Tri rapide optimisé (fonction **Tri_rapide2)**

Données : T : un tableau de valeurs num [1..n] ;

i, j : les indices de début et de fin de la segmentation à effectuer

Résultat : le tableau T trié entre les indices i et j compris

Pseudo Code

```
Tri_rapide2(T,i,j)
    Si i < j alors
        k = Segmente(T,i,j)
        Si k - i > 15 alors
            Tri_rapide(T,i,k-1)
        Sinon
            Tri_insertion(T,i,k-1)
        Fin Si
        Si j - k > 15 alors
            Tri_rapide(T,k+1,j)
        Sinon
            Tri_insertion(T,k+1,j)
        Fin Si
    Fin Si
```

PYTHON

```
def Tri_rapide2(          ) :
```

4. LE TRI FUSION

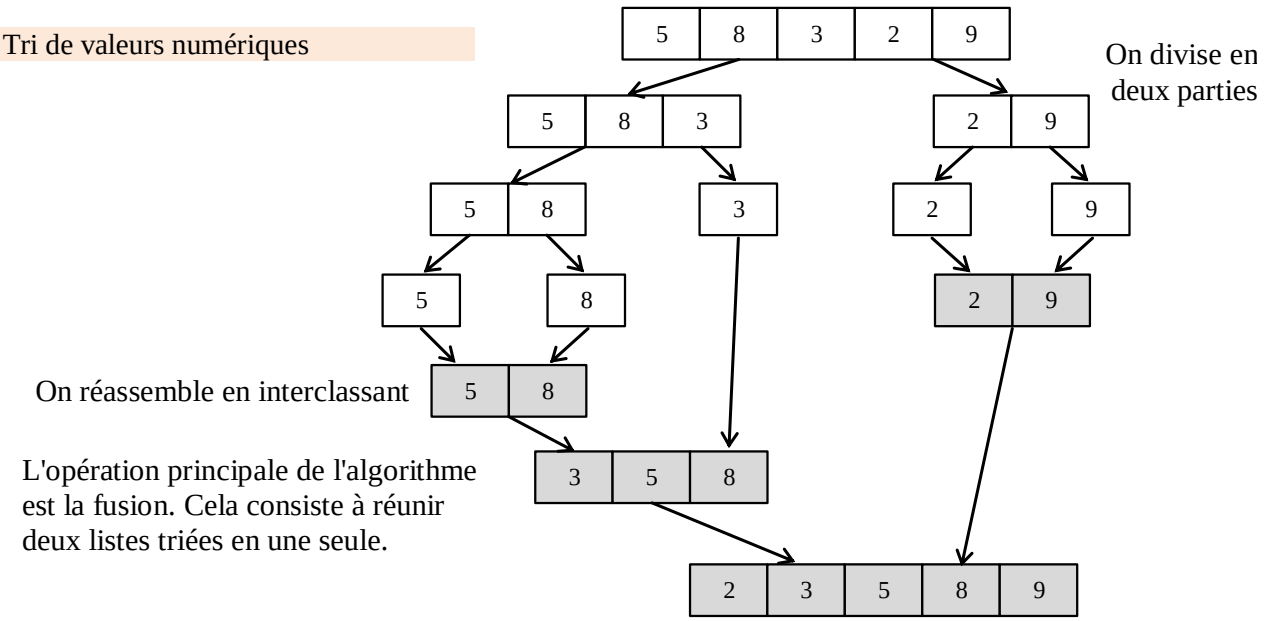
La méthode de tri fusion pour un tableau de données est la suivante :

- On coupe en deux parties à peu près égales les données à trier.
- On trie les données de chaque partie par la méthode de tri fusion.
- On fusionne les deux parties en interclassant les données.

L'algorithme est donc **récursif**. Il fait partie des algorithmes « diviser pour régner ».

La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.

Exemple 4 : Tri de valeurs numériques



Algorithme : fusion de 2 listes triées (fonction **Fusion)**

Données : T : un tableau de valeurs numériques [g..d]

m : un indice tel que $g \leq m < d$ et que les sous-tableaux T[g..m] et T[m+1..d] soient ordonnés

Résultat : le tableau T de valeurs numériques [g..d] ordonnées

Pseudo Code

```
Fusion(T,g,d,m)
    n1 ← m − g + 1          # taille du tableau intermédiaire G
    n2 ← d − m              # taille du tableau intermédiaire D
    Pour i de 1 à n1 faire
        G[i] ← T[g + i − 1]
    Fin Pour
    Pour j de 1 à n2 faire
        D[j] ← T[m + j]
    Fin Pour
    i ← 1
    j ← 1
    G[n1 + 1] ← ∞           # On marque la fin du tableau G
    D[n2 + 1] ← ∞           # On marque la fin du tableau D
    Pour k de g à d faire
        Si i <= n1 et G[i] ≤ D[j] alors
            T[k] ← G[i]
            i ← i + 1
        Sinon
            Si j <= n2 et G[i] > D[j] alors
                T[k] ← D[j]
                j ← j + 1
            Fin Si
        Fin Si
    Fin Pour
```

PYTHO N

```
def Fusion(          ) :
```

Complexité :

Cet algorithme a une complexité en temps de classe linéaire : *C(n)=O(n).*

Par contre, il oblige à utiliser un espace supplémentaire égal à la taille du tableau original T.

Ecrire alors l'algorithme **récursif** de tri fusion :

Algorithme : Tri fusion (fonction **Tri_fusion)**

Données : T : un tableau de valeurs numériques non triées [g..d]
g, d : les indices de début et de fin de tableau

Résultat : le tableau T avec les mêmes valeurs mais triées

PYTHO N

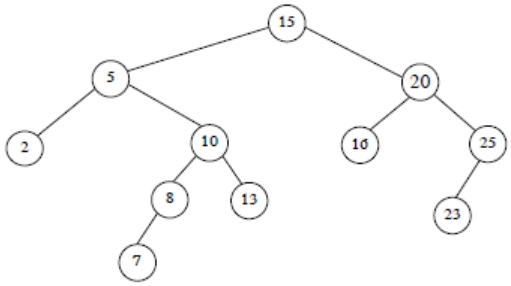
```
def Tri_fusion(          ) :
```

Si l'on s'intéresse au nombre de données à traiter à chaque appel de fonction, la relation de récurrence est du type :

$$C(n) = 2 \times C(n/2) + n.$$

La méthode de tri fusion a donc une ***efficacité temporelle comparable au tri rapide en $O(n \times ln(n))$.*** *Par contre, 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.*

5. ARBRES ET TRI TAS



Un *arbre binaire de recherche* est un arbre dont les noeuds appartiennent à un ensemble totalement ordonné et tel que, pour **tout** noeud interne *a*, les données contenues dans le sous-arbre gauche de *a* sont inférieures ou égales à la donnée contenue dans le noeud *a*, qui est elle-même inférieure ou égale aux données contenues dans le sous-arbre droit de *a*.

Par définition d'un arbre binaire de recherche, un parcours en ordre symétrique de l'arbre donne la liste triée des données que contient l'arbre.

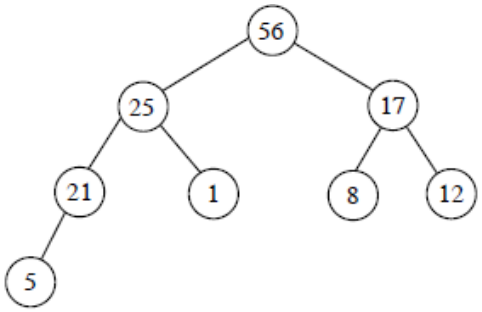
Un arbre binaire fonctionne comme un algorithme de *recherche* qui retourne vrai ou faux selon qu'une clé reçue en paramètre appartient ou non à l'arbre binaire de recherche.

La complexité d'une insertion ou d'une recherche dans un arbre binaire de recherche est :

- dans le pire des cas égale à la profondeur de l'arbre binaire dans lequel se fait l'insertion ;
- en moyenne égale à la profondeur moyenne des feuilles de l'arbre binaire.

Le tri par arbre binaire de recherche est donc au pire en $O(n^2)$ et en moyenne en $O(n\ ln(n))$.

➔ ***Cette méthode est utilisée dans la recherche du chemin le plus court (Djikstra), étudiée au TP4***



On appelle *tas* un arbre binaire parfait, c'est-à-dire où tous les niveaux sont remplis sauf éventuellement le dernier, celui-ci étant rempli de la gauche vers la droite, et dans les sommets duquel figurent des éléments d'un ensemble totalement ordonné, l'élément stocké en un noeud quelconque étant plus grand que les éléments stockés dans ses deux noeuds fils s'il a deux fils, dans son fils s'il a un seul fils. Il n'y a pas d'autre relation d'ordre : un « neveu » peut être plus grand que son « oncle ».

Le module heapq de Python permet de traiter les *tas.*

➔ ***Cette méthode est utilisée dans l'algorithme des arbres de Huffman (codage), étudié au cours 1 et 4.***

6. SYNTHESE

Exemple 5 : Tri de 10000 valeurs numériques générées aléatoirement

Problème : Quelle est l'efficacité des différentes méthodes de tri ?

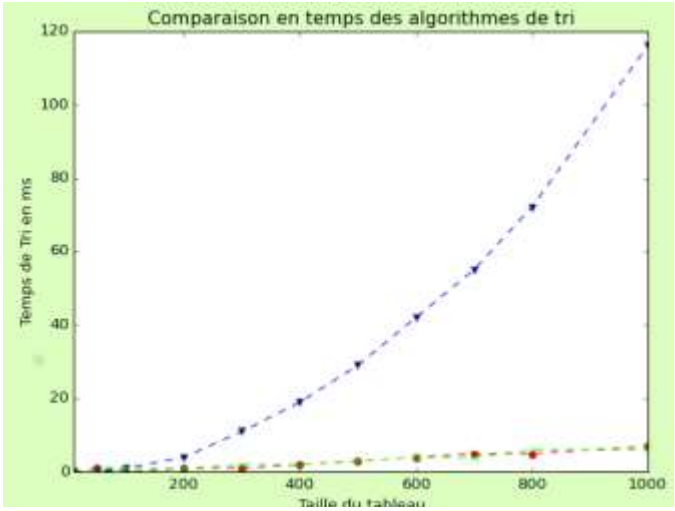
La simulation a été effectuée pour un tableau de 10000 valeurs numériques générées aléatoirement.

Les temps sont donnés en secondes.

	Temps en secondes
Tri Bulle	37.03597636012455
Tri par Insertion	16.26099099187195
Tri par Sélection	15.775444085769777
Tri Fusion	0.157514432699827
Tri Rapide	0.09675168212743301
Tri Natif Python (méthode sort) :	0.0075889533436566126

➔ *On s'aperçoit que le tri Python (méthode sort) est le plus efficace !!*

Pour comparer les temps globaux sur une même machine (Ici processeur Intel Core i7-4510U Haswell (2 GHz, TDP 15W),Mémoire vive 4 Go) on génère des tableaux de dimensions différentes avec des nombres aléatoires. On utilise ***le module time*** *pour déterminer les temps. Ici c'est un temps global dépendant fortement de la machine utilisée. On fait varier la taille du tableau et on compare les temps mis par chacun des algorithmes (point= tri rapide, triangles= tri insertion, croix= tri fusion).*



DISTINGUER PAR LEURS COMPLEXITÉS DEUX ALGORITHMES RÉSOLVANT UN MÊME PROBLÈME :

-- La première étape consiste à vérifier que les algorithmes résolvent exactement le même problème.
– On compare la ***complexité en temps dans le pire des cas (préférable à l'utilisation du module Time*** *car indépendant de la machine).*
– Il faut vérifier que le pire des cas peut arriver en situation réelle
– Penser à comparer la complexité en espace qui peut permettre de départager des algorithmes équivalents