Informatique et Calcul Scientifique

Cours 10 : Algorithmes de tri

La fois passée, on a vu..

- Un algorithme *linéaire* de recherche dans une liste
- La fonction *logarithme* et son utilisation en informatique
- Un algorithme *logarithmique* de recherche binaire dans une liste *triée*

Aujourd'hui, on verra..

Plusieurs algorithmes de tri et leurs temps de parcours

- Le tri par sélection
- Le tri par insertion
- Le tri par fusion.

Algorithmes de tri

- Problème : étant donnée une liste (de nombres, de strings...) de taille n, trier les éléments de cette liste dans l'ordre croissant.
  - `L = [0, -3, 2, 4, 2]` $\rightarrow$ `L = [-3, 0, 2, 2, 4]`

Algorithmes de tri

- Problème : étant donnée une liste (de nombres, de strings...) de taille n, trier les éléments de cette liste dans l'ordre croissant.
  - `L = [0, -3, 2, 4, 2]` $\rightarrow$ `L = [-3, 0, 2, 2, 4]`
- Ce problème peut être résolu par une multitude d'algorithmes...

Algorithmes de tri

- Problème : étant donnée une liste (de nombres, de strings...) de taille n, trier les éléments de cette liste dans l'ordre croissant.
  - `L = [0, -3, 2, 4, 2]` $\rightarrow$ `L = [-3, 0, 2, 2, 4]`
- Ce problème peut être résolu par une multitude d'algorithmes...
- On présente plusieurs algorithmes qui, étant donnée une liste `L`, la modifient pour en trier les éléments (comme le fait la méthode `sort()` de la classe `list` en Python).

Algorithmes de tri

- Pourquoi trier ?
  - Par exemple, trier une liste sur laquelle on va souvent appeler l'opération recherche.
  - Dictionnaires (les vrais) ordonnés par ordre alphabétique des mots, annuaires ordonnés par ordre alphabétique des noms, listes d'étudiants au SAC ordonnés (par Sciper ?)...

Tri par sélection

- Idée : dans une liste triée, le premier élément est le plus petit, le deuxième est le deuxième plus petit, etc.
- Pour trier la liste, on va rechercher le plus petit élément et le mettre à la bonne place (en première position) ; puis rechercher le deuxième plus petit élément (le plus petit parmi ceux qui restent) et le mettre à la bonne place en deuxième position, etc.
- On fait donc grandir une sous-liste triée, en insérant à chaque fois le minimum des éléments restants à la fin de cette sous-liste.

. Exemple interactif : https://visualgo.net/bn/sorting

Tri par sélection

```
def tri_par_selection(L):
    '''
    Entree: liste L de nombres
    Trie L
    '''
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

Tri par sélection : correctitude

```python
def tri_par_selection(L):
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

Invariant de boucle (boucle extérieure) :
Au début de l'itération `i`, la sous-liste `L[0:i]` consiste des *i* plus petits éléments de la liste `L` donnée en entrée, **triés**.

Initialisation : au début de l'itération `0`, `L[0:0]` est vide et consiste donc bien des 0 plus petits éléments de `L`.

Tri par sélection : correctitude

```python
def tri_par_selection(L):
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

Invariant de boucle (boucle extérieure) :
Au début de l'itération `i`, la sous-liste `L[0:i]` consiste des *i* plus petits éléments de la liste `L` donnée en entrée, **triés**.

Maintenance : l'itération `i` consiste à mettre le *i*ème plus petit élément à la *i*ème position dans la liste.

- Pour le prouver, il faut prouver que la boucle for intérieure sélectionne bien le minimum de `L[i:n]`
- Et donc il faut formuler et prouver un invariant de boucle pour la boucle `for` *dans la ième itération de la boucle extérieure :*

  Au début de l'itération `j` de la boucle, `m` contient le minimum de `L[i:j]` et `m_index` l'index de ce minimum.

Tri par sélection : correctitude

```python
def tri_par_selection(L):
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

Invariant de boucle (boucle extérieure) : Au début de l'itération `i`, la sous-liste `L[0:i]` consiste des *i* plus petits éléments de la liste `L` donnée en entrée, **triés**.

Terminaison : à la sortie de la boucle, donc au début de l'itération `n` (qui n'aura pas lieu), `L[0:n]` consiste des *n* éléments de la liste `L`, **triés**. Donc la liste entière est triée.

Tri par sélection : temps de parcours

```
def tri_par_selection(L):
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

- Ces deux boucles `for` imbriquées impliquent un temps de parcours qui est $\Theta(n^2)$.
- Y a-t-il une distinction entre le pire des cas et d'autres cas ?

Tri par insertion

- Idée : trier une liste comme on trie une main à un **jeu de cartes**.
- On fait grandir une sous-liste triée, en insérant un élément à la fois **à la bonne place** dans cette sous-liste.

. Exemple interactif : https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

Tri par insertion

```
def tri_par_insertion(L):
    '''
    Entree: liste L de nombres
    Trie L
    '''
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
```

Tri par insertion : correctitude

```
def tri_par_insertion(L):
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
```

Invariant de boucle (boucle extérieure) :
Au début de l'itération `i`, la sous-liste `L[0:i]` consiste des mêmes *i* éléments initialement dans `L[0:i]`, mais **triés**.

Tri par insertion : temps de parcours

```
1  def tri_par_insertion(L):
2      n = len(L)
3      for i in range(n):
4          j = i
5          while j > 0 and L[j] < L[j-1]:
6                  L[j], L[j-1] = L[j-1], L[j]
7                  j -= 1
```

- Le temps de parcours au pire des cas est lorsque à l'itération `i` de la boucle `for`, la boucle `while` itère i fois,
- c'est-à-dire lorsque la liste est triée dans l'ordre décroissant.
- Dans ce cas, les instructions aux lignes 6 et 7 s'exécutent au total

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \text{ fois.}$$

- Le temps de parcours au pire des cas est $\Theta(n^2)$.

Un algorithme de tri récursif

- Peut-on faire mieux que $\Theta(n^2)$?

  $\Longrightarrow$ Oui, en utilisant un algorithme **récursif** : le tri par fusion.

- Paradigme **"diviser pour régner"** : Pour une instance de taille n du problème du tri, càd une liste de taille n, `tri_par_fusion` :
  - **Divise** la liste en deux sous-listes de taille (approximativement) $n/2$
  - **Trie récursivement** chaque sous-liste
  - **Fusionne** les sous-listes triées pour obtenir une version triée de la liste initiale.
- Le tri des sous-listes se fait récursivement et les appels récursifs s'arrêtent pour les listes de taille 1 (cas de base).

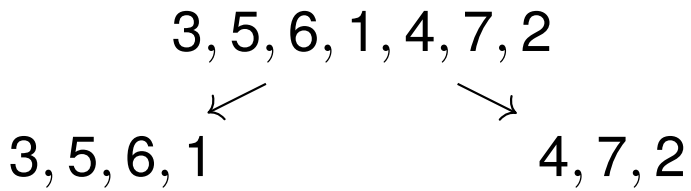
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.

$$3, 5, 6, 1, 4, 7, 2$$

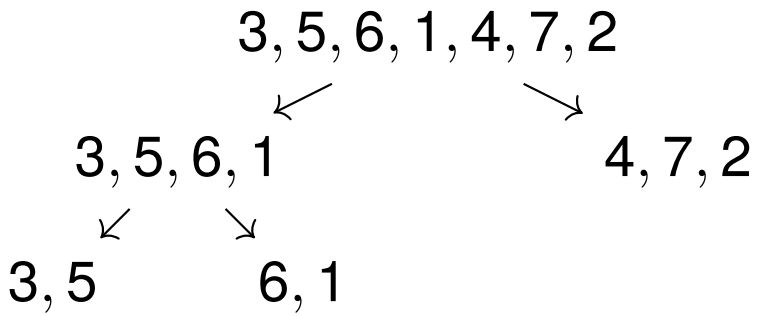
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



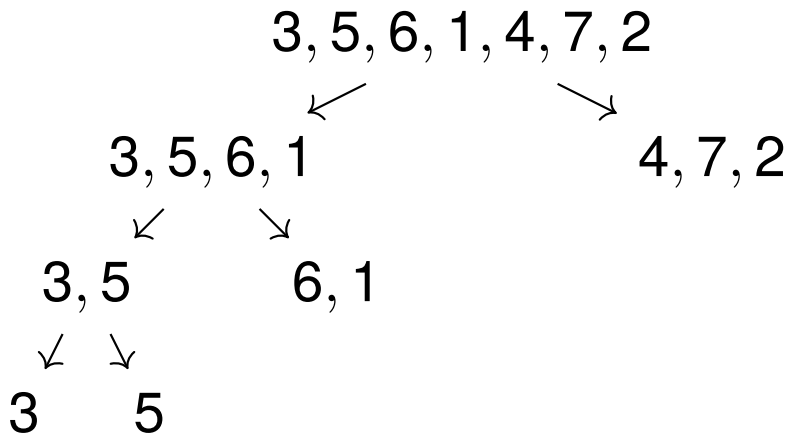
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



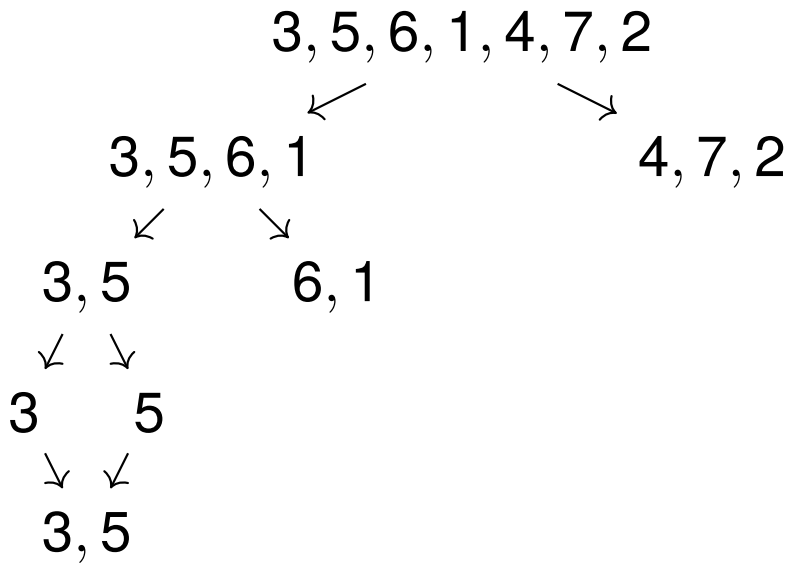
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



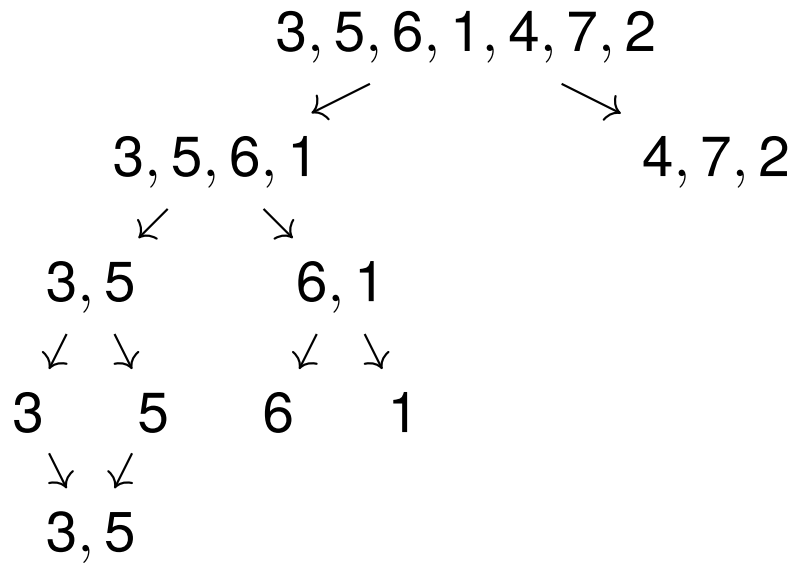
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



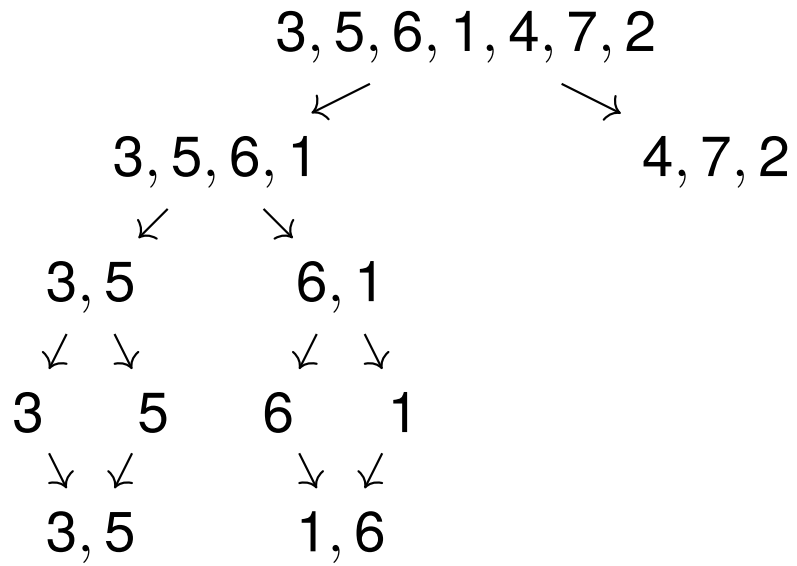
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



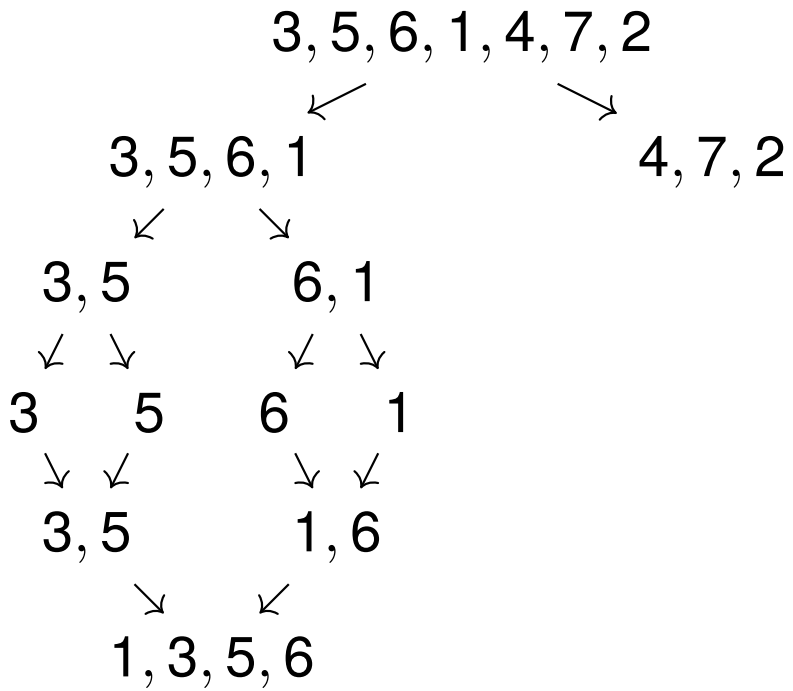
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



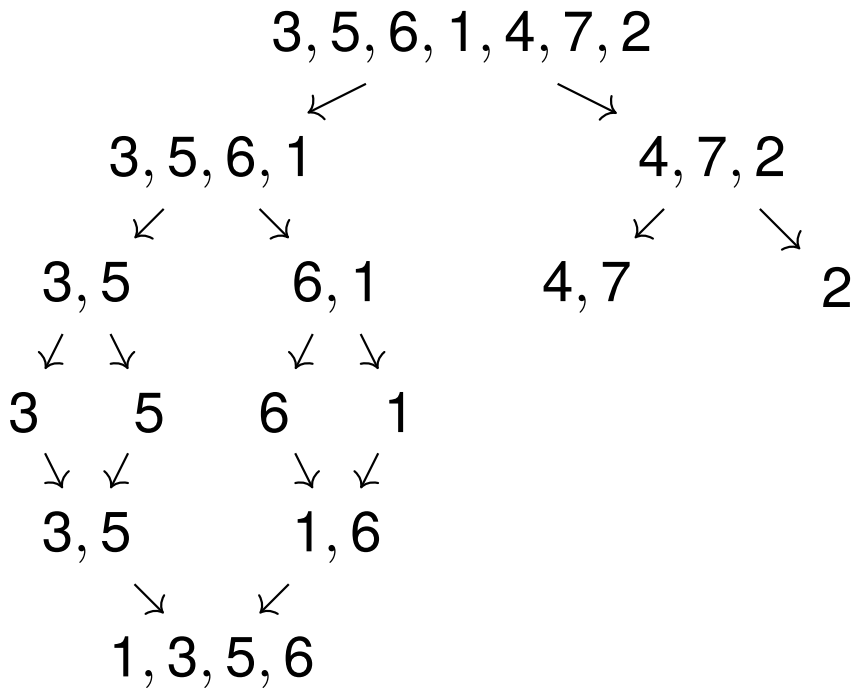
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



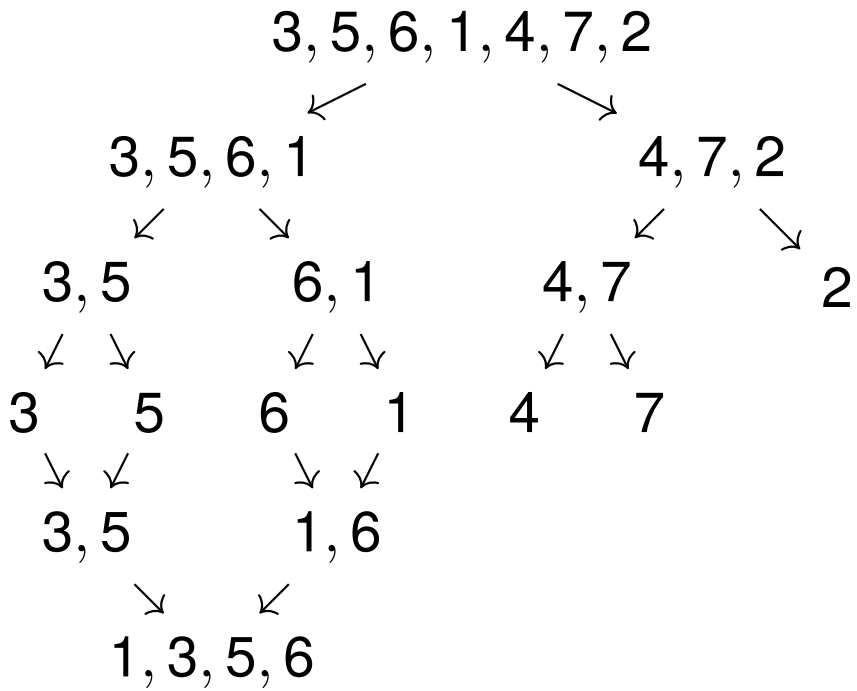
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



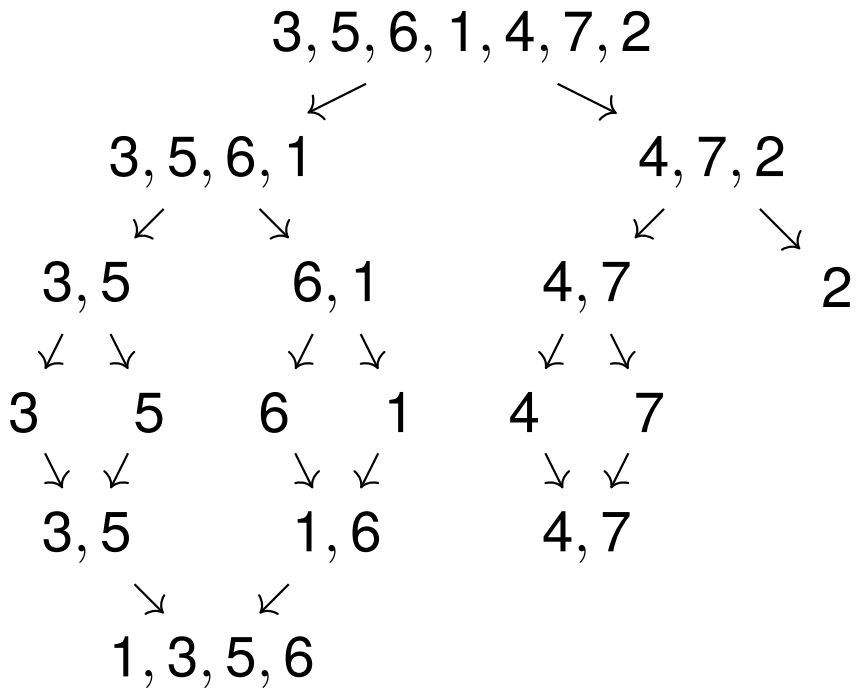
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



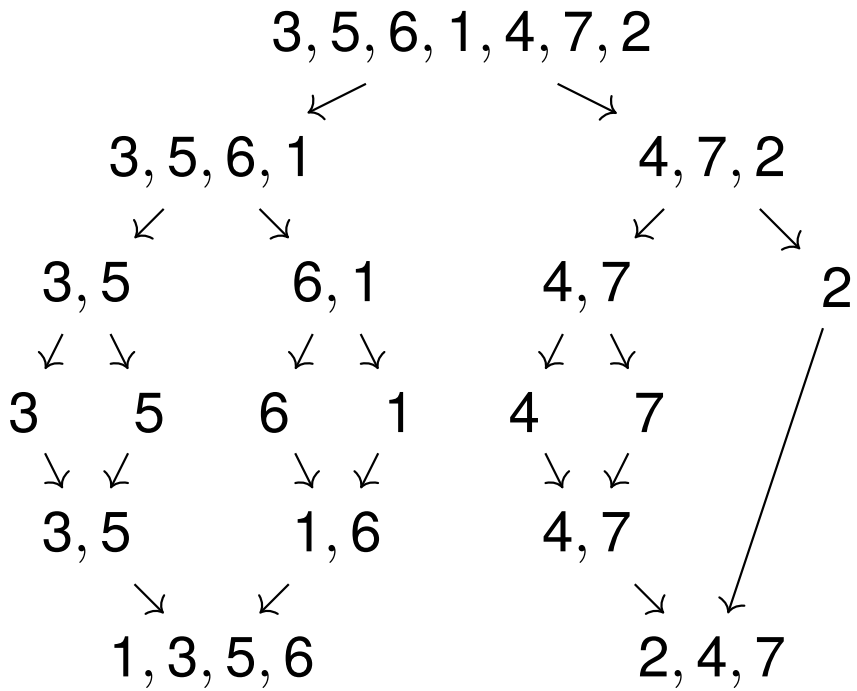
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



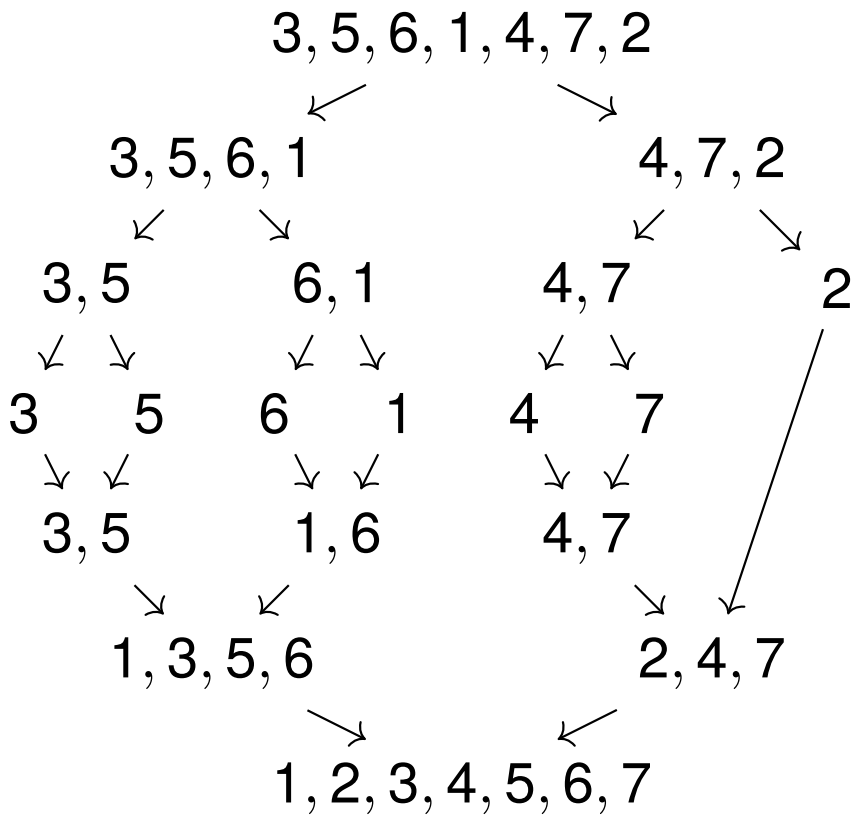
Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



Tri par fusion - idée

On suppose qu'on a accès à un algorithme de **fusion**, qui, étant donnée une liste `L` dont les deux moitiés sont triées, trie tous les éléments de `L`.



Tri par fusion

L'algorithme de tri par fusion est donc implémenté de manière récursive pour une instance L :

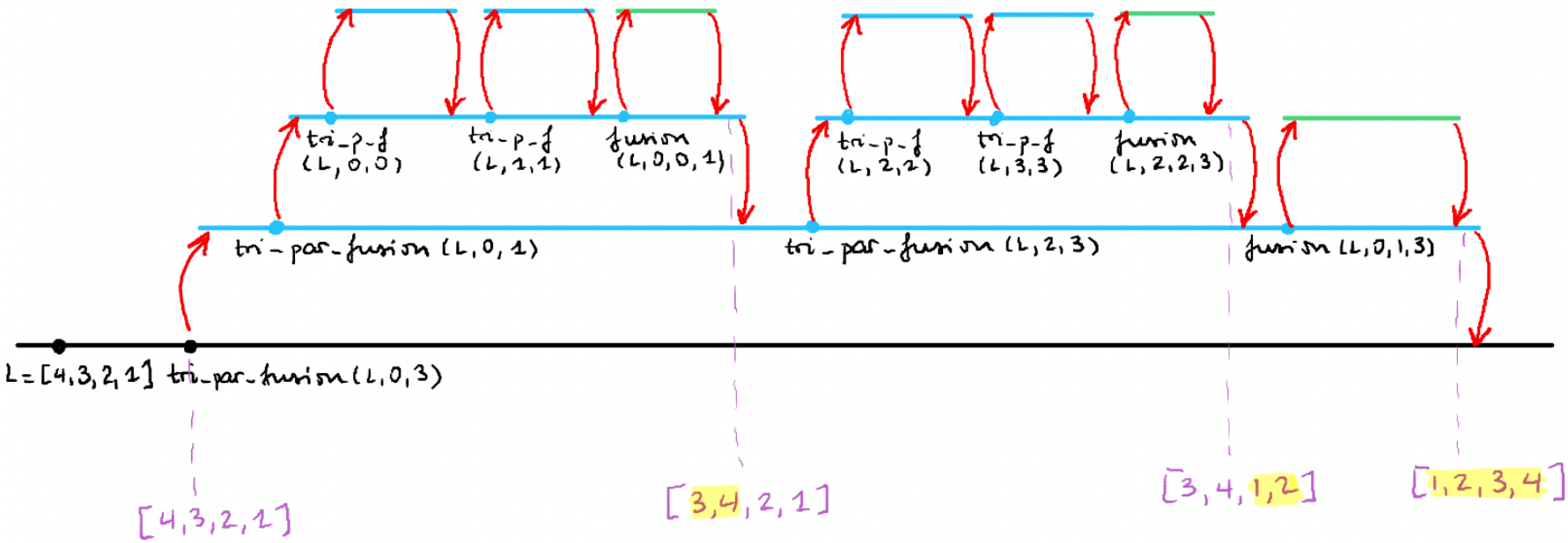
```
def tri_par_fusion(L, bas, haut):
    '''
    Entree: liste L, indices bas et haut
    Trie la tranche L[bas:haut+1] entre bas et haut
    '''
    if haut - bas > 0:
        milieu = (bas + haut)//2
        tri_par_fusion(L, bas, milieu)
        tri_par_fusion(L, milieu+1, haut)
        fusion(L, bas, milieu, haut)
```

Remarque. Ici, le cas de base c'est de ne rien faire !

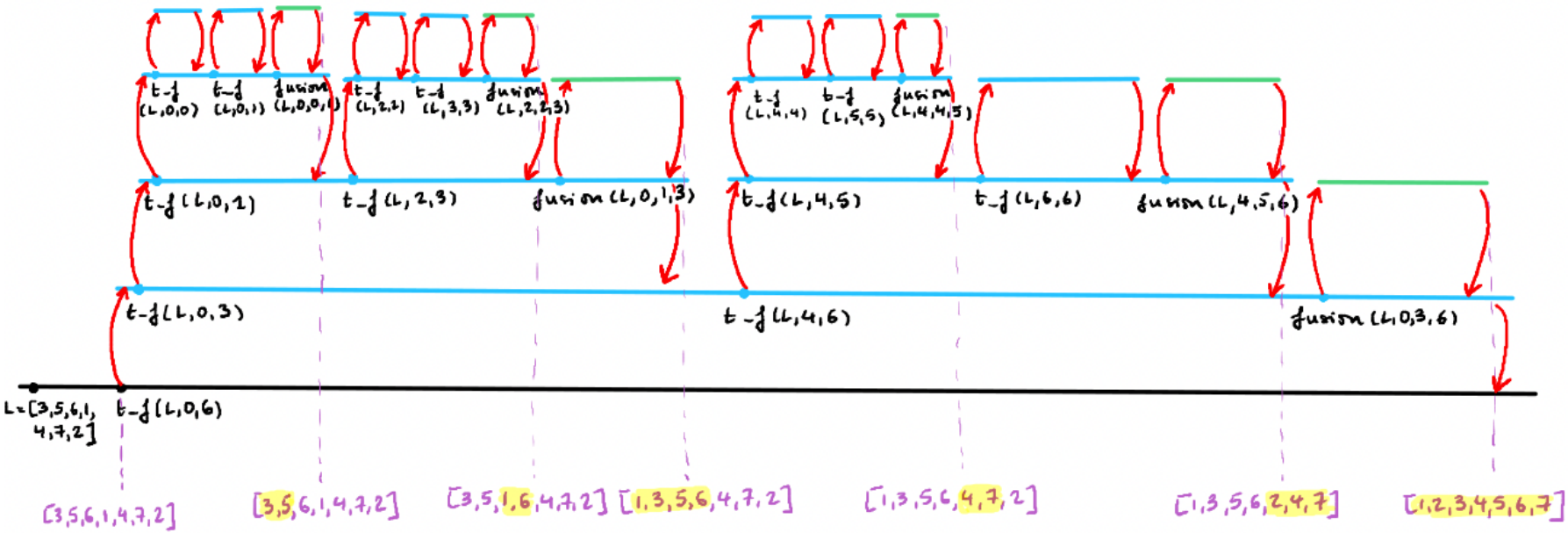
Fusion de deux sous-listes

```
def fusion(L, bas, milieu, haut):
    '''Entree: liste L t.q. L[bas:milieu+1] et
              L[milieu+1:haut+1] sont triees
    Trie L[bas:haut+1]'''
    L1 = L[bas:milieu+1]
    L2 = L[milieu+1:haut+1]
    L1.append(float('inf'))
    L2.append(float('inf'))
    L1_index = 0
    L2_index = 0
    for i in range(bas, haut+1):
        if L1[L1_index] <= L2[L2_index]:
            L[i] = L1[L1_index]
            L1_index += 1
        else:
            L[i] = L2[L2_index]
            L2_index += 1
```

Tri par fusion : exemple



Tri par fusion : exemple



Correctitude (idée)

- La correctitude de `tri_par_fusion(L, bas, haut)` découle, par un argument inductif, de la correctitude de `fusion` et de la correctitude de `tri_par_fusion` sur des instances plus petites (cas de base : une liste de taille 1 est déjà triée).
- La correctitude de `fusion` est impliquée par l'invariant de boucle suivant pour la boucle `for` :

  Au début de l'itération `i` de la boucle, `L[bas:i]` contient les `(i - bas)` plus petits éléments parmi ceux de `L1` et `L2`, triés ; et `L1[L1_index]` et `L2[L2_index]` sont les plus petits éléments de leurs listes respectives qui n'ont pas encore été copiés dans `L`.

Fusion : temps de parcours

Soit $F(n)$ le temps de de parcours de fusion (au pire de cas), où n est la taille de `L[bas:haut+1]`, càd $n = \text{haut} - \text{bas} + 1$.

```
def fusion(L, bas, milieu, haut):
    L1 = L[bas:milieu+1]
    L2 = L[milieu+1:haut+1]
    L1.append(float('inf'))
    L2.append(float('inf'))
    L1_index = 0
    L2_index = 0
    for i in range(bas, haut+1):
        if L1[L1_index] <= L2[L2_index]:
            L[i] = L1[L1_index]
            L1_index += 1
        else:
            L[i] = L2[L2_index]
            L2_index += 1
```

slicing : $\Theta(\text{milieu} - \text{bas})$

$\Theta(\text{haut} - \text{milieu})$

$\Theta(\text{haut} - \text{bas})$

On voit ici que $F = \Theta(n)$. **Pour simplifier notre analyse ultérieure** on suppose qu'il y a un $c \in \mathbb{R}_+$ tel que

$$\boxed{F(n) = cn.}$$

Tri par fusion : temps de parcours

- Soit $T(n)$ le temps de parcours, au pire des cas, de `tri_par_fusion` pour une entrée de taille $n = \text{haut - bas} + 1$.
- On suppose que $n = 2^k$ où $k > 0$.

```
def tri_par_fusion(L, bas, haut):
    if haut - bas > 0:
        milieu = (bas + haut)//2
        tri_par_fusion(L, bas, milieu)
        tri_par_fusion(L, milieu+1, haut)
        fusion(L, bas, milieu, haut)
```

Tri par fusion : temps de parcours

- Soit $T(n)$ le temps de parcours, au pire des cas, de `tri_par_fusion` pour une entrée de taille $n =$ haut - bas$+1$.
- On suppose que $n = 2^k$ où $k > 0$.

```
def tri_par_fusion(L, bas, haut):
    if haut - bas > 0:
        milieu = (bas + haut)//2
        tri_par_fusion(L, bas, milieu)
        tri_par_fusion(L, milieu+1, haut)
        fusion(L, bas, milieu, haut)
```

- On a alors

$$\begin{aligned} T(n) &= 2T(n/2) + F(n), \\ &= \boxed{2T(n/2) + cn}. \end{aligned}$$

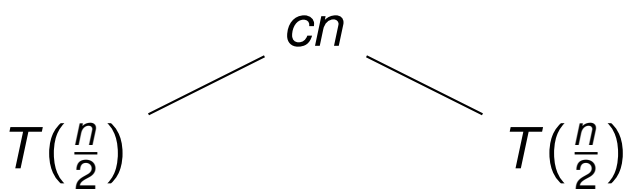
Arbre de récurrence

$$\boxed{T(n) = 2T(n/2) + cn}$$

$$T(n)$$

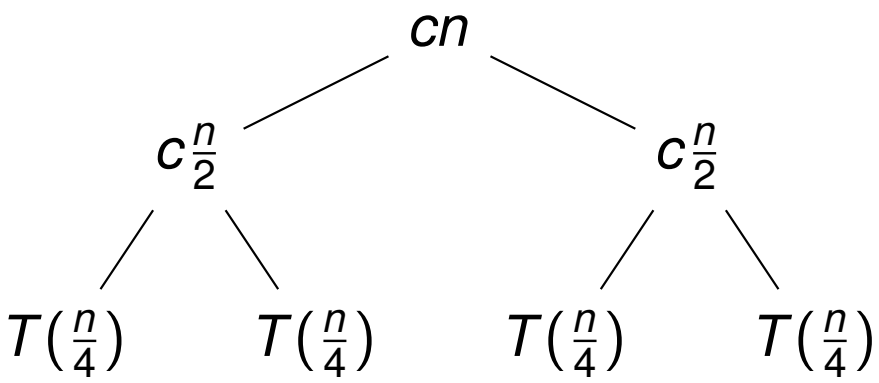
Arbre de récurrence

$$\boxed{T(n) = 2T(n/2) + cn}$$



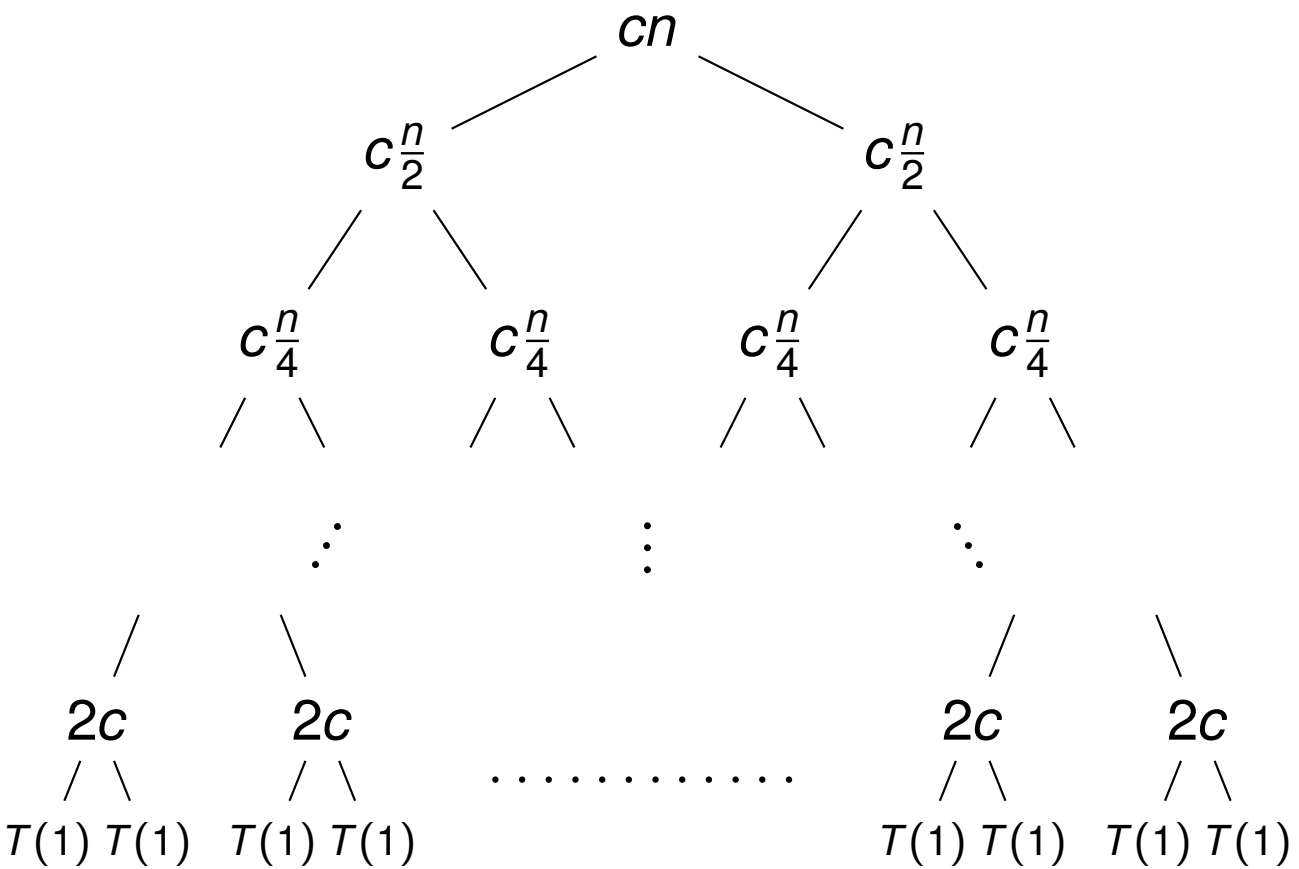
Arbre de récurrence

$$\boxed{T(n) = 2T(n/2) + cn}$$



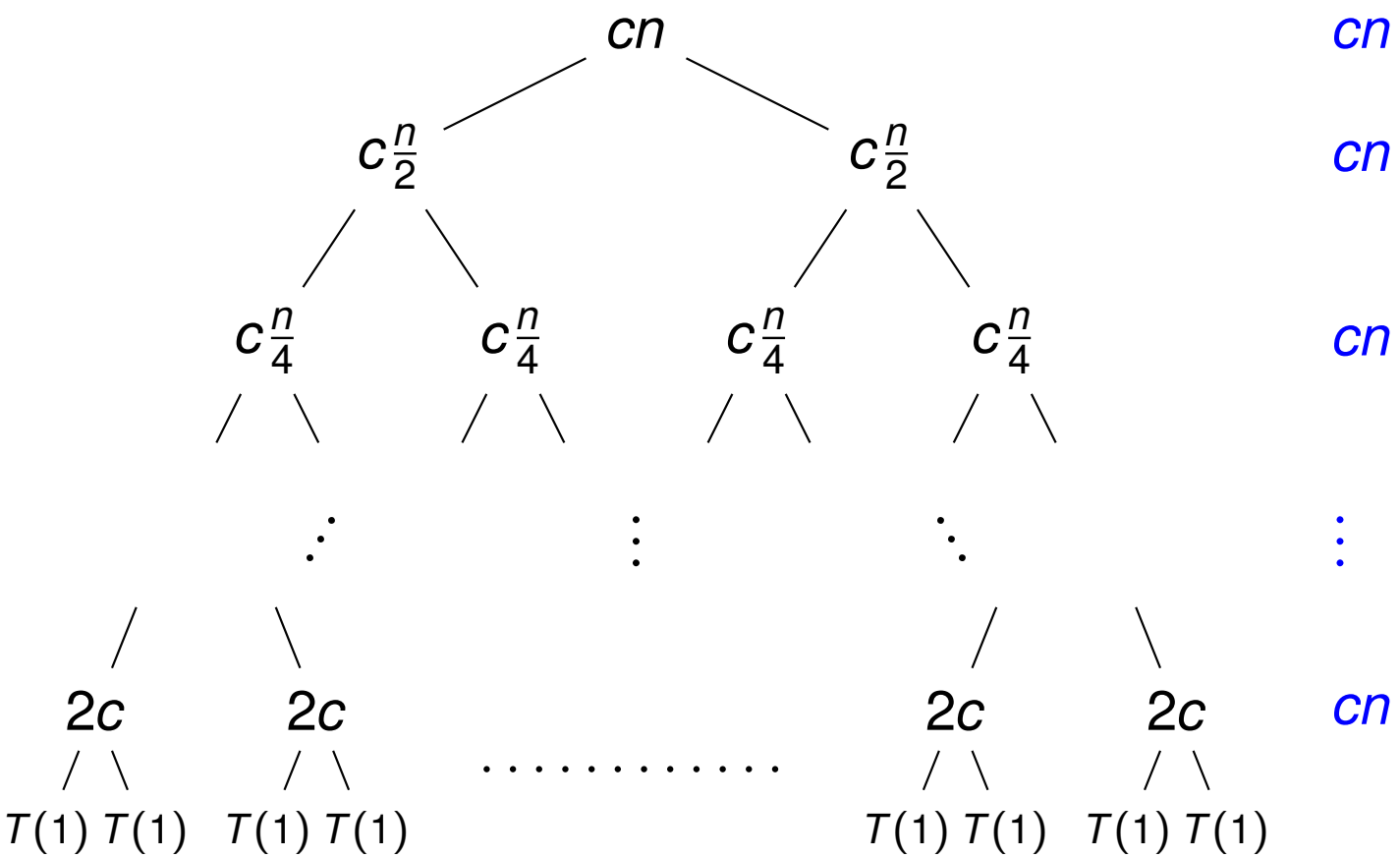
Arbre de récurrence

$$\boxed{T(n) = 2T(n/2) + cn}$$



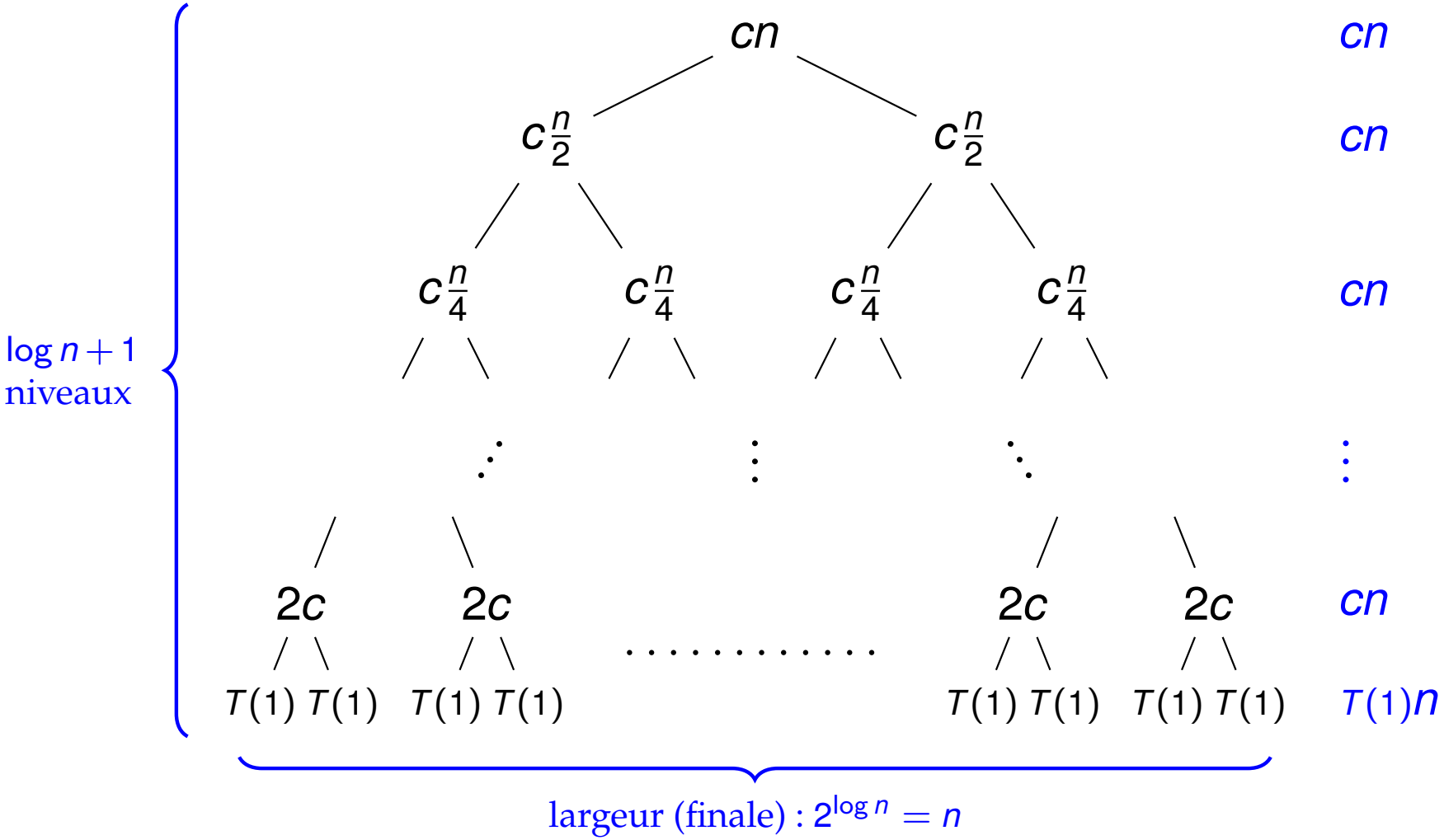
Arbre de récurrence

$$T(n) = 2T(n/2) + cn$$



Arbre de récurrence

$$T(n) = 2T(n/2) + cn$$



Tri par fusion - temps de parcours

- La somme des temps de parcours à tous les sommets de l'arbre est

$$cn \log n + T(1)n$$

- On en déduit le temps de parcours de `tri_par_fusion` :

$$T(n) = \Theta(n \log n).$$

Tri par fusion - temps de parcours

- La somme des temps de parcours à tous les sommets de l'arbre est

$$cn \log n + T(1)n$$

- On en déduit le temps de parcours de `tri_par_fusion` :

$$T(n) = \Theta(n \log n).$$

Tri par fusion - temps de parcours

- La somme des temps de parcours à tous les sommets de l'arbre est

$$cn \log n + T(1)n$$

- On en déduit le temps de parcours de `tri_par_fusion` :

$$T(n) = \Theta(n \log n).$$

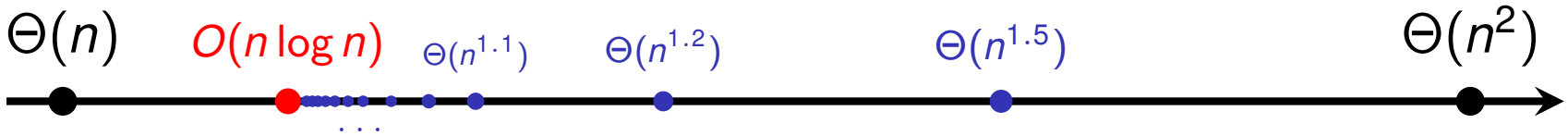
Tri par fusion - temps de parcours

- La somme des temps de parcours à tous les sommets de l'arbre est

$$cn \log n + T(1)n$$

- On en déduit le temps de parcours de `tri_par_fusion` :

$$T(n) = \Theta(n \log n).$$



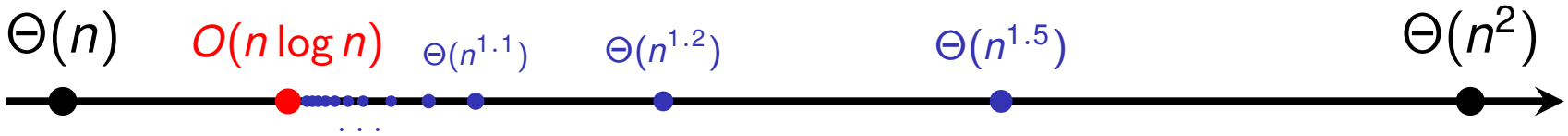
Tri par fusion - temps de parcours

- La somme des temps de parcours à tous les sommets de l'arbre est

$$cn \log n + T(1)n$$

- On en déduit le temps de parcours de `tri_par_fusion` :

$$T(n) = \Theta(n \log n).$$



- Notre approche ici n'était pas rigoureuse à 100 %. Formellement, nous avons effectivement formulé une hypothèse, qui pourrait ensuite être prouvée par induction.

Master Theorem

- Il existe aussi un théorème (**the master theorem**) qu'on peut appliquer pour trouver l'ordre de croissance d'une fonction $T(n)$ qui satisfait une récurrence de la forme

$$T(n) = aT(n/b) + f(n),$$

  pour des constantes $a \geq 1$ et $b > 1$ (et avec un ou des cas de base appropriés).
- Cette récurrence modélise le temps de parcours pour un algorithme où on divise un problème de taille n en a sous-problèmes de taille (approximativement) n/b chacun. $f(n)$ représente le coût de créer/recombiner les sous-instances.
- Il existe aussi des théorèmes pour résoudre des récurrences plus compliquées, par exemples lorsque les tailles des sous-problèmes sont inégales.

Comparaison des algorithmes de tri

- Pour une entrée de taille n, `tri_par_selection` et `tri_par_insertion` ont temps de parcours $\Theta(n^2)$ alors que `tri_par_fusion` a temps de parcours $\Theta(n \log_2(n))$.
- Par contre, `tri_par_selection` et `tri_par_insertion` trient **sur place**, alors que `tri_par_fusion` a besoin de $\Theta(n)$ espace de travail en mémoire.
- La constante qui se cache dans la notation $\Theta(\cdot)$ pour `tri_par_fusion` est assez grande. En pratique, pour de petites valeurs de n, `tri_par_insertion` est l'algorithme à préférer.
- L'algorithme implémenté par la méthode `sort` et la fonction native `sorted` en Python est `timsort` : c'est un algorithme hybride basé sur le tri par fusion et le tri par insertion.

Comparaison des algorithmes de tri

- Un autre algorithme de tri répandu est le **tri rapide** ou **tri pivot** (**quicksort**).
- Le tri rapide choisit un élément de la liste comme pivot, et en comparant chaque autre élément de la liste au pivot, crée deux sous-listes : la liste des éléments plus petits que le pivot et la liste des éléments plus grand que le pivot. Puis il trie récursivement ces deux sous-listes.
- Le tri rapide trie sur place.
- Si le pivot est choisi aléatoirement à chaque étape, le temps de parcours de tri rapide est $\Theta(n^2)$ au pire des cas mais $\Theta(n \log n)$ en moyenne (avec une plus petite constante que celle du tri par fusion).

Comparaison des algorithmes de tri

- Tous les algorithmes que nous avons vus sont basés sur la comparaison de paires d'éléments.
- On peut prouver que tout algorithme basé sur la comparaison de paires d'éléments a un temps de parcours qui est $\Omega(n \log n)$.
- Le tri par fusion est donc asymptotiquement optimal parmi les algorithmes qui comparent des paires d'éléments.
- Si on a plus d'information sur les données de la liste, par exemple si on sait que tous les éléments sont entre 0 et une constante K, on peut trier en $\Theta(n)$ (sans effectuer de comparaisons d'éléments).