Diviser pour régner

Résumé

Pour trier un gros paquet de copies, on peut le couper en deux, trier chaque moitié séparément, puis fusionner les deux paquets triés. C'est l'idée de **diviser pour régner** : découper le problème en sous-problèmes indépendants de même nature et de taille réduite (diviser), les résoudre récursivement (régner), puis assembler les résultats (combiner). La recherche dichotomique et la puissance rapide relèvent de ce schéma. Le **tri fusion** en est l'exemple emblématique : il trie n éléments en $O(n \log n)$, là où les tris par sélection et par insertion exigent $O(n^2)$.

Prérequis : La récursivité, Tris naïfs.

① Mémo

Le schéma diviser pour régner

Diviser Découper le problème en sous-problèmes **indépendants** de même nature, en général deux moitiés de taille $n/2$.

Régner Résoudre chaque sous-problème par un appel récursif, jusqu'aux cas de base (taille 0 ou 1), résolus directement.

Combiner Assembler les résultats des sous-problèmes pour obtenir la solution du problème initial.

Profondeur Diviser la taille par deux à chaque appel donne une profondeur de récursion $\log_2 n$, contre n pour une récursion qui retire un seul élément (comme la factorielle).

Le tri fusion

Fusion Deux listes déjà triées se fusionnent en une liste triée par un parcours simultané avec deux indices : on prend à chaque étape le plus petit des deux éléments courants. Coût : au plus $n_1 + n_2 - 1$ comparaisons.

Tri Si la liste a 0 ou 1 élément, elle est triée. Sinon, couper au milieu, trier chaque moitié (appels récursifs), fusionner.

Complexité $\log_2 n$ niveaux de découpage, et à chaque niveau les fusions traitent n éléments au total : $O(n \log n)$ dans tous les cas.

Mémoire Le tri fusion construit de nouvelles listes : il n'est pas en place, contrairement aux tris par sélection et insertion.

D'autres exemples

Recherche dichotomique Un seul sous-problème de taille $n/2$ (la moitié utile) et rien à combiner : $O(\log n)$.

Puissance rapide $x^n = \left(x^{n//2}\right)^2$, multiplié par x si n est impair : $O(\log n)$ multiplications au lieu de n.

Maximum par moitiés Le maximum d'une liste est le plus grand des maximums des deux moitiés. Complexité $O(n)$, comme le parcours simple : diviser pour régner n'est pas toujours plus rapide.

Pièges fréquents

- Oublier le cas de base pour la liste vide : en découpant une liste à un élément, on obtient une moitié vide, et la récursion ne s'arrête plus.
- Découper en `L[:m]` et `L[m+1:]` : l'élément d'indice `m` disparaît.
- Oublier de recopier la fin de la liste non épuisée à la fin de la fusion.
- Croire que diviser accélère toujours : le gain vient de la combinaison peu coûteuse, pas du découpage lui-même.

Erreurs classiques

Code erroné	Code correct	Explication
`tri_fusion(L[:m])` et `tri_fusion(L[m+1:])`	`L[:m]` et `L[m:]`	Les tranches `L[:m]` et `L[m:]` recouvrent toute la liste ; avec `m+1`, l'élément `L[m]` n'est dans aucune moitié et disparaît du résultat.
`if len(L) == 1: return L` comme seul cas de base	`if len(L) <= 1: return L`	Découper `[x]` donne `[]` et `[x]` : sans cas de base pour la liste vide, `tri_fusion([])` se rappelle indéfiniment (`RecursionError`).
Fusion terminée dès qu'une liste est épuisée	`extend(L1[i:])` puis `extend(L2[j:])`	Quand une liste est vide, la fin de l'autre est déjà triée : il faut la recopier, sinon des éléments manquent.
`return L1 + L2` dans la fusion	Parcours avec deux indices `i` et `j`	La concaténation juxtapose les deux listes sans les mêler : `[1, 4] + [2, 3]` donne `[1, 4, 2, 3]`, qui n'est pas trié.

② Exemples

Programme 1 · Fusionner deux listes triées

```python
def fusion(L1, L2):
    """Fusionne deux listes triées en une seule liste triée."""
    resultat = []
    i, j = 0, 0
    while i < len(L1) and j < len(L2):
        if L1[i] <= L2[j]:
            resultat.append(L1[i])
            i += 1
        else:
            resultat.append(L2[j])
            j += 1
    resultat.extend(L1[i:])    # reste de L1 (éventuellement vide)
    resultat.extend(L2[j:])    # reste de L2
    return resultat

print(fusion([1, 4, 9], [2, 3, 10]))   # [1, 2, 3, 4, 9, 10]
```

Trace de `fusion([1, 4, 9], [2, 3, 10])` :

Étape	Comparaison	Élément pris	`resultat`
1	$1 \leqslant 2$	1 (de L1)	`[1]`
2	$4 > 2$	2 (de L2)	`[1, 2]`
3	$4 > 3$	3 (de L2)	`[1, 2, 3]`
4	$4 \leqslant 10$	4 (de L1)	`[1, 2, 3, 4]`
5	$9 \leqslant 10$	9 (de L1)	`[1, 2, 3, 4, 9]`
fin	L1 épuisée	reste de L2 recopié	`[1, 2, 3, 4, 9, 10]`

Cinq comparaisons pour six éléments : la fusion est linéaire.

Programme 2 · Le tri fusion

```
def tri_fusion(L):
    """Renvoie une nouvelle liste contenant les éléments de L triés."""
    if len(L) <= 1:                 # cas de base : 0 ou 1 élément
        return L
    m = len(L) // 2
    gauche = tri_fusion(L[:m])      # régner sur la moitié gauche
    droite = tri_fusion(L[m:])      # régner sur la moitié droite
    return fusion(gauche, droite)   # combiner

print(tri_fusion([8, 3, 5, 1, 9, 2]))   # [1, 2, 3, 5, 8, 9]
```

Déroulement sur `[8, 3, 5, 1, 9, 2]` :

Phase	**Listes**
découpage, niveau 1	`[8, 3, 5]` `[1, 9, 2]`
découpage, niveau 2	`[8]` `[3, 5]` `[1]` `[9, 2]`
découpage, niveau 3	`[3]` `[5]` `[9]` `[2]`
fusions, niveau 3	`[3, 5]` `[2, 9]`
fusions, niveau 2	`[3, 5, 8]` `[1, 2, 9]`
fusion, niveau 1	`[1, 2, 3, 5, 8, 9]`

Trois niveaux de découpage pour six éléments ($\lceil \log_2 6 \rceil = 3$) ; à chaque niveau, les fusions parcourent au total six éléments.

Ordres de grandeur : nombre d'opérations pour trier n éléments.

n	$\log_2 n$	**tri fusion,** $n \log_2 n$	**tri par sélection,** $n^2/2$
8	3	24	32
1 024	10	10 240	524 288
10^6	20	2×10^7	5×10^{11}

Programme 3 · Puissance rapide

```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    demi = puissance_rapide(x, n // 2)
    if n % 2 == 0:
        return demi * demi
    return x * demi * demi

print(puissance_rapide(2, 10))   # 1024, en 4 appels récursifs au lieu de 10
```

Programme 4 · Maximum par moitiés

```python
def maximum_dpr(L):
    """Maximum d'une liste non vide, par diviser pour régner."""
    if len(L) == 1:
        return L[0]
    m = len(L) // 2
    return max(maximum_dpr(L[:m]), maximum_dpr(L[m:]))

print(maximum_dpr([3, 9, 1, 7, 4]))   # 9
```

③ Exercices

Exercice 1 *Trace du tri fusion* On applique `tri_fusion` à la liste `[6, 2, 7, 4, 1, 5, 3]`.

1. Donner les listes obtenues à chaque niveau de découpage.
2. Donner, dans l'ordre où elles sont effectuées, les fusions réalisées.
3. Combien d'appels à `tri_fusion` sont effectués au total (appel initial compris) ?

Exercice 2 *Compter les comparaisons d'une fusion*

1. Écrire une fonction `fusion_compteur(L1, L2)` qui renvoie un couple formé de la liste fusionnée et du nombre de comparaisons effectuées.
2. Pour deux listes triées de quatre éléments chacune, donner un exemple qui demande le moins de comparaisons possible, puis un exemple qui en demande le plus. Combien dans chaque cas ?

Exercice 3 *Le maximum par moitiés*

1. Écrire `maximum_dpr(L)` qui renvoie le maximum d'une liste non vide en la coupant en deux moitiés.
2. Combien d'appels à `maximum_dpr` sont effectués pour une liste de 8 éléments ? Pour une liste de n éléments ?
3. Quelle est la complexité de cette fonction ? Est-elle meilleure que celle du parcours simple ?

Exercice 4 *Combien de temps pour trier ?* Une machine effectue 10^7 opérations élémentaires par seconde.

1. Compléter le tableau des ordres de grandeur pour $n = 16$, $n = 10^4$ et $n = 10^6$: $\log_2 n$, $n \log_2 n$ et $\frac{n^2}{2}$.
2. Estimer le temps nécessaire pour trier 10^6 éléments par tri fusion, puis par tri par sélection.
3. Une liste de 10^6 éléments déjà triée est-elle plus rapide à trier par tri fusion ? Et par tri par insertion ?

Solutions des exercices

Corrigé de l'exercice 1.

1. $m = 7//2 = 3$: niveau 1, `[6, 2, 7]` et `[4, 1, 5, 3]` ; niveau 2, `[6]`, `[2, 7]`, `[4, 1]`, `[5, 3]` ; niveau 3, `[2]`, `[7]`, `[4]`, `[1]`, `[5]`, `[3]`.
2. Les fusions se font en remontant, moitié gauche d'abord : `[2, 7]` ; `[2, 6, 7]` ; `[1, 4]` ; `[3, 5]` ; `[1, 3, 4, 5]` ; enfin `[1, 2, 3, 4, 5, 6, 7]`.
3. Un appel initial, deux appels au niveau 1, quatre au niveau 2 (`[6]`, `[2, 7]`, `[4, 1]`, `[5, 3]`) et six au niveau 3 : $1 + 2 + 4 + 6 = 13$ appels. Sept d'entre eux portent sur une liste à un élément (cas de base).

Corrigé de l'exercice 2.

```
def fusion_compteur(L1, L2):
    resultat = []
    i, j, nb = 0, 0, 0
    while i < len(L1) and j < len(L2):
        nb += 1
        if L1[i] <= L2[j]:
            resultat.append(L1[i])
            i += 1
        else:
            resultat.append(L2[j])
            j += 1
    resultat.extend(L1[i:])
    resultat.extend(L2[j:])
    return resultat, nb

print(fusion_compteur([1, 2, 3, 4], [5, 6, 7, 8]))   # ([1, ..., 8], 4)
print(fusion_compteur([1, 3, 5, 7], [2, 4, 6, 8]))   # ([1, ..., 8], 7)
```

Meilleur cas : tous les éléments de `L1` sont plus petits que ceux de `L2` : quatre comparaisons, puis `L2` est recopiée sans comparaison. **Pire cas :** les éléments alternent : $4 + 4 - 1 = 7$ comparaisons, la dernière liste restante n'ayant plus qu'un élément.

Corrigé de l'exercice 3.

```
def maximum_dpr(L):
    if len(L) == 1:
        return L[0]
    m = len(L) // 2
    return max(maximum_dpr(L[:m]), maximum_dpr(L[m:]))

print(maximum_dpr([3, 9, 1, 7, 4, 8, 2, 6]))   # 9
```

Pour 8 éléments : $1 + 2 + 4 + 8 = 15$ appels (un par nœud de l'arbre des appels). Pour n éléments, $2n - 1$ appels : n feuilles et $n - 1$ combinaisons.

Complexité : chaque appel fait au plus une comparaison (`max`), donc $O(n)$, comme le parcours simple. Ici, diviser pour régner n'apporte rien : la combinaison coûte autant que le problème lui-même, et la récursion consomme de la mémoire en plus. Le schéma n'est gagnant que lorsque la combinaison est peu coûteuse au regard du travail évité, comme dans le tri fusion.

Corrigé de l'exercice 4.

1.

n	$\log_2 n$	$n \log_2 n$	$n^2/2$
16	4	64	128
10^4	$\approx 13{,}3$	$\approx 1{,}3 \times 10^5$	5×10^7
10^6	≈ 20	$\approx 2 \times 10^7$	5×10^{11}

2. Tri fusion : $2 \times 10^7 / 10^7 = 2$ secondes. Tri par sélection : $5 \times 10^{11} / 10^7 = 5 \times 10^4$ secondes, soit environ 14 heures.
3. Le tri fusion effectue toujours les mêmes découpages et fusions : environ $n \log_2 n$ opérations, liste triée ou non. Le tri par insertion, lui, ne fait qu'une comparaison par élément sur une liste déjà triée : $O(n)$, soit 10^6 opérations, un dixième de seconde. Sur une liste presque triée, l'insertion bat le tri fusion ; sur une liste quelconque, le tri fusion l'emporte largement.