Diviser pour regner: résumé

Diviser pour régner:

Classe d'algorithme où l'on découpe un problème en sous problèmes qui s'énoncent de la même manière

et qu'on recompose à la fin pour former une solution

C'est une approche "du haut vers les bas".

Généralement, les algorithmes sont récursifs

Calculer la puissance d'un nombre

Comment calculer 3^7 ?

$$3^7 = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

Ce n'est pas *diviser pour régner*.

Algorithme naïf pour y^n

Puissance : $(y, n) \mapsto y^n$

1. On initialise $p = 0$ et $i = 0$
2. Tant que $i < n$ faire
   - $p = p \times y$
   - $i = i + 1$
3. Retourner p

Complexité

Clairement linéaire. Une seule boucle qui itère autant de fois que la puissance voulue.

Exponentiation rapide

Diviser pour régner

$ExpoRapide : (y, n) \mapsto y^n$

Si $n = 0$ alors

- retourner 1

Sinon si n est pair

- $a = ExpoRapide(y, n//2)$
- retourner $a \times a$

Sinon

- retourner $y * ExpoRapide(y, n - 1)$

```python
def expo_rapide(x, n: int):
    """Calcule x ^ n avec l'algorithme de l'exponentiation rapide"""
    if n == 0:
        return 1
    elif n % 2 == 0:
        a = expo_rapide(x, n // 2)
        return a * a
    else:
        return x * expo_rapide(x, n - 1)


assert expo_rapide(3, 7) == 3 ** 7
```

Complexité

La complexité est logarithmique : $O(\log_2 n)$

Tri fusion (*merge sort*)

Les algorithmes de tri étudiés en première (sélection, insertion) sont *simples* mais *lent*. Leur coût algorithmique est *quadratique* : le temps d'exécution évolue avec *le carré de la taille du tableau à trier*.

Si `tri_select(tab)` prend 1 seconde pour un tableau de taille 1000, il prendra environ $10 \times 10 = 100$ secondes pour un tableau 10 fois plus grand, de taille $10 \times 1000 = 10000$.

On dit que selection et insertion sont en $O(n^2)$.

Tri fusion

Il existe des tris par comparaison beaucoup plus rapides, en particulier le *tri fusion*, découvert par John Von Neumann en 1945.

Son coût calculatoire est en $O(n \log n)$

On démontre que ce coût est optimal, à un facteur près. Il n'est pas possible d'écrire un algorithme de tri par comparaison dont la complexité calculatoire soit meilleure. Certains peuvent aller plus vite sur des cas particuliers.

Algorithme

Le principe est :

- de découper la liste en tableaux en les divisant par 2, jusqu'à ce que chaque tableau ait une taille 1.
- de "fusionner" les deux tableaux.

Pour la fusion,

- si le tableau a une taille 1, il est trié, rien à faire,
- sinon, les deux tableaux étant triés, on les parcourt en prenant, à chaque fois, le plus petit des éléments de chaque pour le ranger dans un tableau final. Les éléments restant sont rangés à la fin.

Algorithme détaillé

> Tri Fusion (`tableau`):
>
> - Si `tableau` est de taille <= 1 on ne fait rien.
> - Sinon, On sépare `tableau` en 2 parties `gauche` et `droite`,
>   - On appelle Tri fusion sur `gauche` et sur `droite`
>   - On appelle Fusionner(`tableau`, `gauche`, `droite`)
>
> Fusionner(`tableau`, `gauche`, `droite`):
>
> - On parcourt les deux tableaux `gauche` et `droite` en même temps.
>   Pour chaque paire d'éléments, on place le plus petit dans `tableau`.
> - S'il reste des éléments dans `gauche` ou dans `droite` on les place à la fin de tableau

En une seule image

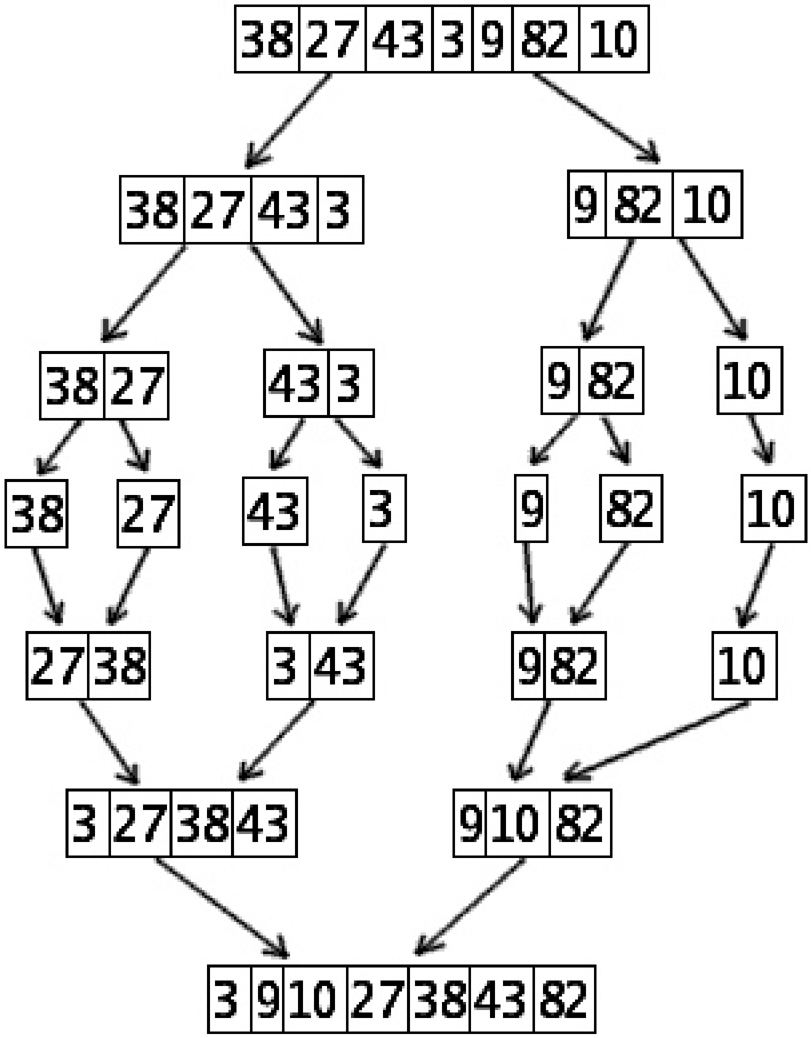

Figure 1: exemple

Complexité

- tri fusion est appelée autant de fois qu'il faut d'étapes pour arriver à une taille 1 : $\log_2(n)$, si n est la taille du tableau,
- fusion prend autant d'étapes qu'il y a d'éléments à fusionner : linéaire.

Le coût total est le produit des deux coûts : $O(n \log n)$.

Code

```python
def tri_fusion(tableau: list) -> None:
    """Tri fusion d'une liste. Modifie la liste reçue"""
    if len(tableau) > 1:
        milieu = len(tableau) // 2
        gauche = tableau[:milieu]
        droite = tableau[milieu:]
        tri_fusion(gauche)
        tri_fusion(droite)
        fusion(tableau, gauche, droite)


def fusion(tableau: list, gauche: list, droite: list) -> None:
    """fusionne deux tableaux triés en un seul, trié"""
    g = len(gauche)
    d = len(droite)
    i = j = k = 0
    while i < g and j < d:
        if gauche[i] < droite[j]:
            tableau[k] = gauche[i]
            i += 1
        else:
            tableau[k] = droite[j]
            j += 1
        k += 1
    # gauche est vide ou droite est vide
    while i < g:
        tableau[k] = gauche[i]
        i += 1
        k += 1
    while j < d:
        tableau[k] = droite[j]
        j += 1
        k += 1


def tester():
    """teste les différentes fonctions"""
    l = [1]
    tri_fusion(l)
    assert l == [1]

    l = [3, 2, 1]
    tri_fusion(l)
    assert l == [1, 2, 3], repr(l)

    from random import shuffle

    for _ in range(10):
        tableau = list(range(10))
        shuffle(tableau)
        tri_fusion(tableau)
        assert tableau == sorted(tableau), str(tableau)

    print("ok")

tester()
```

Dichotomie

Résumé de ce qu'on a fait en première

La recherche dichotomique est un algorithme diviser pour régner.

On sépare le tableau en **deux parties de même taille (environ)** à chaque étape selon la comparaison de `tableau[milieu]` et `cle`.

Il n'y a pas de "combinaison" des résultats, on renvoie simplement ce qu'on veut (l'indice ou un booléen).

En voici une version *récursive*.

```python
def dichotomie_rec(tableau: list, cle: int, debut: int=0, fin: int=None) -> int:
    """
    Renvoie l'indice de cle dans tableau ou -1 si cle ne figure pas dans le tableau
    >>> tableau = [5, 23, 28, 39, 45, 63, 71, 89]
    >>> dichotomie_rec(tableau, 23)
    1
    >>> dichotomie_rec(tableau, 3)
    -1
    """
    if fin is None:
        fin = len(tableau) - 1
    milieu = (debut + fin) // 2
    if debut > fin:
        return -1
    elif tableau[milieu] == cle:
        return milieu
    elif tableau[milieu] > cle:
        fin = milieu - 1
    else:
        debut = milieu + 1
    return dichotomie_rec(tableau, cle, debut=debut, fin=fin)


def exemple():
    tableau = [5, 23, 28, 39, 45, 63, 71, 89]
    print("tableau", tableau)
    print("indice de 89 dans tableau ?", dichotomie_rec(tableau, 89))
    print("indice de 30 dans tableau ?", dichotomie_rec(tableau, 30))


exemple()
```

Conclusion

La méthode ***diviser pour régner*** :

- découper le problème en sous-problèmes qui s'énoncent de la même manière
- résoudre les cas limites
- combiner les solutions