Diviser pour régner

Algorithmique

1 Le paradigme diviser pour régner

Le paradigme **diviser pour régner** (*divide and conquer*) est une méthode de conception d'algorithmes qui repose sur trois étapes :

1. **Diviser** : découper le problème initial en sous-problèmes plus petits de même nature ;
2. **Régner** : résoudre chaque sous-problème récursivement, jusqu'à obtenir des cas suffisamment simples ;
3. **Combiner** : assembler les solutions des sous-problèmes pour construire la solution du problème initial.

Ce paradigme est particulièrement efficace lorsque le découpage réduit significativement la taille des données à chaque étape. Beaucoup d'algorithmes DPR coupent le problème en deux moitiés ; selon le nombre de moitiés à traiter et le coût de la combinaison, on obtient par exemple une complexité en $O(\log n)$ (recherche dichotomique) ou en $O(n \log n)$ (tri fusion, au lieu de $O(n^2)$ pour les tris par sélection et par insertion).

2 Recherche dichotomique

2.1 Principe

La recherche dichotomique (*binary search*) recherche un élément dans un tableau **trié** en divisant l'espace de recherche par deux à chaque étape :

— comparer l'élément cherché avec l'élément du milieu ;

— s'il est égal, on a trouvé ;

— s'il est inférieur, on cherche dans la moitié gauche ;

— s'il est supérieur, on cherche dans la moitié droite.

2.2 Version itérative

```
def recherche_dichotomique(x, tab):
    """Renvoie l'indice de x dans tab (trié), ou -1 si absent."""
    gauche = 0
    droite = len(tab) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tab[milieu] == x:
            return milieu
        elif tab[milieu] < x:
            gauche = milieu + 1
        else:
            droite = milieu - 1
    return -1
```

2.3 Version récursive

```python
def recherche_dicho_rec(x, tab, gauche, droite):
    """Version récursive de la recherche dichotomique."""
    if gauche > droite:
        return -1
    milieu = (gauche + droite) // 2
    if tab[milieu] == x:
        return milieu
    elif tab[milieu] < x:
        return recherche_dicho_rec(x, tab, milieu + 1, droite)
    else:
        return recherche_dicho_rec(x, tab, gauche, milieu - 1)
```

2.4 Complexité

À chaque étape, l'intervalle de recherche est divisé par deux. Si le tableau contient n éléments, le nombre maximal d'étapes est $\lfloor \log_2 n \rfloor + 1$.

La complexité de la recherche dichotomique est en $O(\log n)$.

Taille du tableau	Recherche séquentielle	Recherche dichotomique
1 000	1 000 comparaisons	10 comparaisons
1 000 000	1 000 000 comparaisons	20 comparaisons

③ Tri fusion

3.1 Principe

Le **tri fusion** (*merge sort*) applique le paradigme diviser pour régner :

1. **Diviser** : couper le tableau en deux moitiés ;
2. **Régner** : trier récursivement chaque moitié ;
3. **Combiner** : fusionner les deux moitiés triées.

Le cas de base est un tableau de zéro ou un élément, déjà trié par définition.

3.2 La fonction de fusion

```python
def fusion(gauche, droite):
    """Fusionne deux listes triées en une seule liste triée."""
    resultat = []
    i, j = 0, 0
    while i < len(gauche) and j < len(droite):
        if gauche[i] <= droite[j]:
            resultat.append(gauche[i])
            i += 1
        else:
            resultat.append(droite[j])
            j += 1
    resultat.extend(gauche[i:])
    resultat.extend(droite[j:])
```

```
    return resultat
```

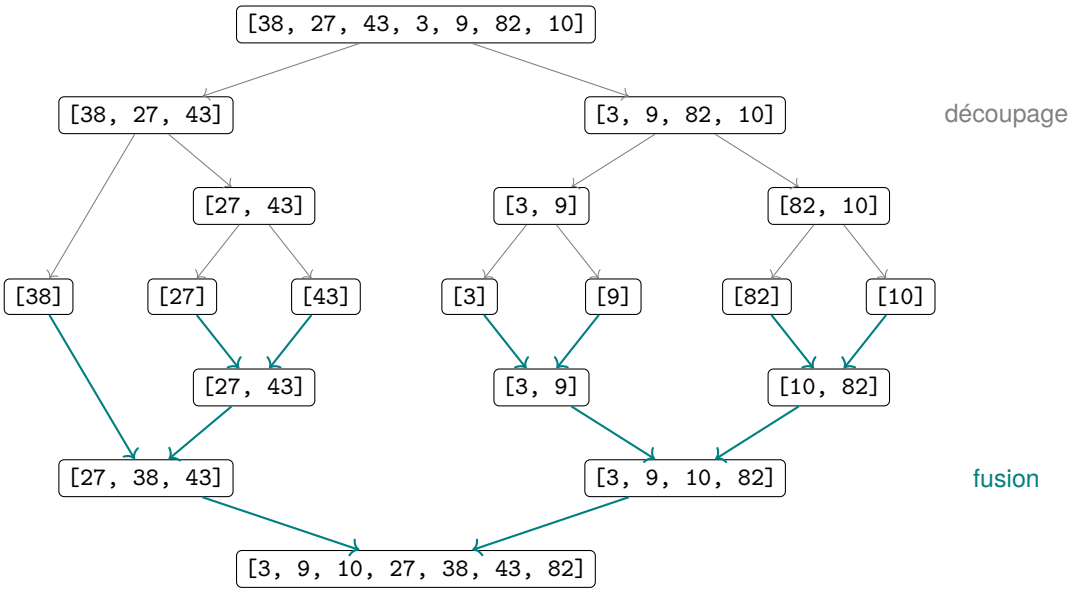
Quand la boucle s'arrête, l'une des deux listes est épuisée et il reste à recopier les éléments de l'autre, déjà triés. La méthode `extend` s'en charge : `resultat.extend(L)` ajoute à la fin de `resultat` tous les éléments de la liste `L`, dans l'ordre.

3.3 Le tri fusion complet

```
def tri_fusion(tab):
    """Trie une liste par l'algorithme du tri fusion."""
    if len(tab) <= 1:
        return tab
    milieu = len(tab) // 2
    gauche = tri_fusion(tab[:milieu])
    droite = tri_fusion(tab[milieu:])
    return fusion(gauche, droite)
```

3.4 Exemple pas à pas

Tri fusion de `[38, 27, 43, 3, 9, 82, 10]` :



3.5 Complexité

À chaque niveau de récursion, on divise le tableau en deux moitiés. Il y a donc $\log_2 n$ niveaux. À chaque niveau, la fusion parcourt l'ensemble des n éléments.

La complexité du tri fusion est en $O(n \log n)$ dans **tous** les cas (meilleur, pire, moyen).

Inconvénient : le tri fusion nécessite de la mémoire supplémentaire pour stocker les sous-tableaux lors de la fusion (il n'est pas *en place*).

④ Exponentiation rapide

L'exponentiation rapide calcule a^n en exploitant la parité de l'exposant :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a \times (a^{(n-1)/2})^2 & \text{si } n \text{ est impair} \end{cases}$$

```python
def puissance(a, n):
    """Calcule a^n par exponentiation rapide."""
    if n == 0:
        return 1
    if n % 2 == 0:
        demi = puissance(a, n // 2)
        return demi * demi
    else:
        demi = puissance(a, (n - 1) // 2)
        return a * demi * demi
```

Au lieu de $n - 1$ multiplications, l'exponentiation rapide n'en effectue que $O(\log n)$.

⑤ Synthèse

Algorithme	**Diviser**	**Combiner**	**Complexité**
Recherche dichotomique	Couper l'intervalle en deux	Renvoyer l'indice	$O(\log n)$
Tri fusion	Couper le tableau en deux	Fusionner	$O(n \log n)$
Exponentiation rapide	Diviser l'exposant par deux	Multiplier	$O(\log n)$