Dans ce chapitre, on explore une méthode générale utilisée en algorithmique : la méthode diviser pour régner. Nous allons en voir le principe puis présenter deux exemples classiques d'algorithme mettant en oeuvre cette méthode.

1 Principe

La méthode diviser pour régner, consiste à traiter un problème de taille n selon les étapes suivantes :
— On découpe le problème en un certain nombre a de sous-problèmes de même tailles n/b ;
— On traite récursivement ces problèmes ;
— On recombine les différentes réponses obtenues.

Remarque : Cette définition générale est très proche de n'importe quel traitement récursif d'un problème. La spécificité ici est que les sous-problèmes sont toujours au moins 2 et de même taille.

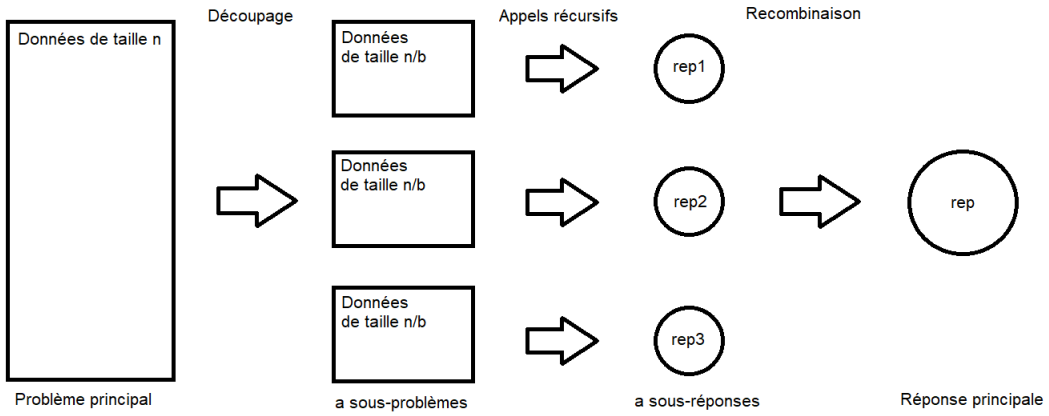

FIGURE 1 – Schéma générale de la méthode diviser pour régner

Cette méthode, on le verra dans la dernière partie, peu se révéler très efficace selon les valeurs de a, b et de la complexité de la recombinaison.

2 Retour sur la recherche dichotomique

En première, on étudie formule la recherche dichotomique dans un tableau trié de la manière suivante :

Algorithme 1 : Recherche dichotomique impérative

Données : un tableau trié t de taille n, une valeur x
Résultat : $B = VRAI$ si $x \in t$, $FAUX$ sinon
Idée : On définit un zone d'indices de recherche $[a, b]$ dont on divise la taille par deux à chaque étape, en gardant la partie droite ou gauche en fonction de la valeur de t à l'indice m moyenne de a et b

```
début
    a ← 0 ;
    b ← n − 1 ;
    B = FAUX ;
    tant que a < b et B̄ faire
        m ← ⌈(a+b)/2⌉ ;
        si t[m] = x alors
            B ← VRAI
        fin
        si t[m] < x alors
            a ← m + 1
        fin
        si t[m] > x alors
            b ← m − 1
        fin
    fin
    renvoyer B
fin
```

Donnons un exemple graphique :

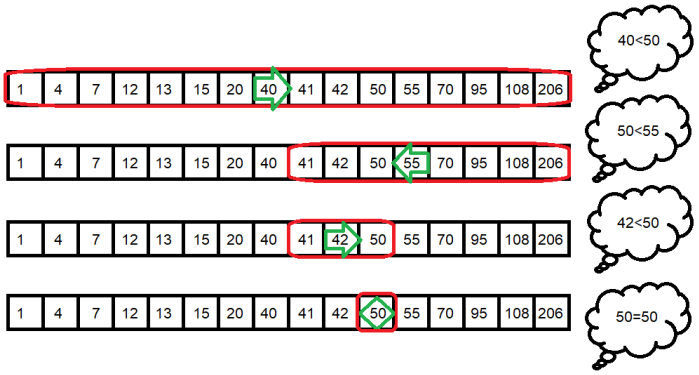

FIGURE 2 – recherche par dichotomie de 50

On a affaire à un algorithme de type diviser pour régner, adaptons le schéma général :

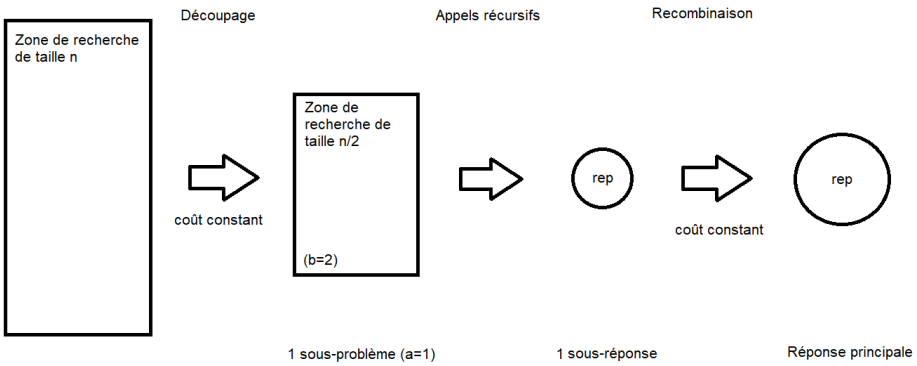

FIGURE 3 – Schéma récursif de la recherche dichotomique

Écrivons enfin une version récursive de l'algorithme :

Algorithme 2 : Recherche dichotomique récursive

Données : un tableau trié t de taille n, une valeur x
Résultat : $B = VRAI$ si $x \in t$, $FAUX$ sinon
Idée : On retrouve la zone de recherche mais étant maintenant des arguments de la fonction récursive, on relance récursivement la fonction sur $[m+1, b]$ ou $[a, m-1]$ en fonction de la valeur de x
Fonction : $recherche(x, t, a = 0, b = n-1)$

```
si a > b alors
    renvoyer FAUX
fin
m ← ⌈(a+b)/2⌉ ;
si t[m] = x alors
    renvoyer VRAI
fin
si t[m] < x alors
    renvoyer recherche(x, t, m+1, b)
fin
si t[m] > x alors
    renvoyer recherche(x, t, a, m-1)
fin
end
```

Cet algorithme, comme on l'a déjà vu l'an dernier et comme nous allons le revoir dans la dernière partie, a une complexité en $\log_2(n)$, ce qui est meilleur que la complexité linéaire de l'algorithme de recherche simple. Attention : on doit cette amélioration au caractère trié du tableau sans lequel l'algorithme dichotomique ne fonctionnerait pas !

3 Tri fusion

On s'intéresse au tri fusion (*merging sort*). Le principe du tri fusion est le suivant (on choisit de le décrire comme un algorithme du les listes chaînées car il y est assez adapté) :

— Une liste vide ou de taille 1 est triée ;
— fusionner deux liste triés en restant trié est faisable en temps linéaire, il suffit, tant que les deux listes sont non-vides, de comparer unes par unes les plus petites valeurs des deux listes et d'insérer la plus petite dans la liste fusionnée ;
— pour réaliser le tri fusion, on coupera donc notre liste en deux tant que sa taille sera supérieure à 1, puis on fusionnera toutes les listes obtenues.

Prenons un exemple graphique :

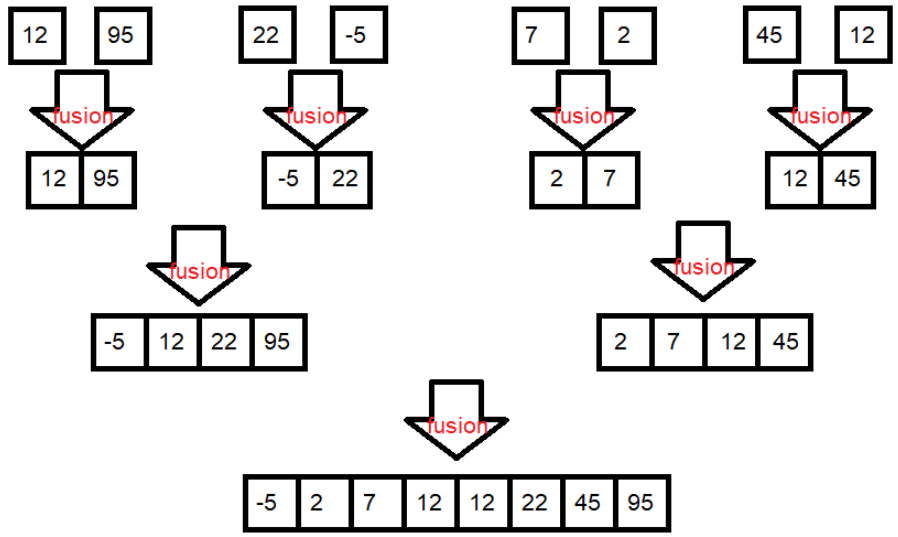

FIGURE 4 – Fusions menant à un tableau trié

Nous allons commencer par écrire une fonction $fusion(L1, L2)$:

Algorithme 3 : Fusion de deux listes triées

Données : Deux listes triées L_1 et L_2
Résultat : La liste triée L contenant les valeurs de $L1$ et de $L2$
Idée : Tant que les deux listes ne sont pas vides, on ajoute à L la plus petite valeur en tête de L_1 ou de L_2
Fonction : $fusion(L_1, L_2)$

```
L ←liste vide ;
tant que L1 non-vide et L2 non-vide faire
    v1 ← la tête de L1 ;
    v2 ← la tête de L2 ;
    si v1 < v2 alors
        ajouter v1 à L ;
        retirer v1 de L2
    sinon
        ajouter v2 à L ;
        retirer v2 de L1
    fin
fin
tant que L1 non-vide faire
    ajouter v1 à L ;
    retirer v1 de L1
fin
tant que L2 non-vide faire
    ajouter v2 à L ;
    retirer v2 de L2
fin
end
```

Nous avons maintenant besoin d'un algorithme permettant de couper une liste en deux sous-listes de tailles équivalentes. Une solution consiste à remplir deux listes à partir de la liste argument en alternant les destinations :

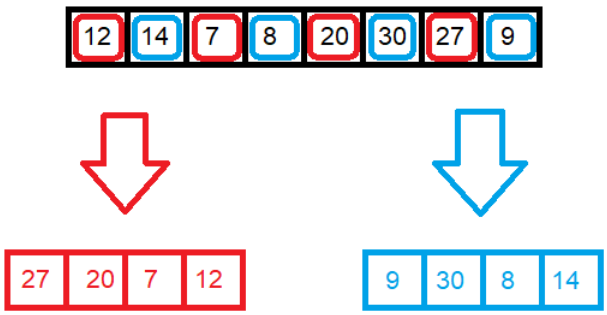

FIGURE 5 – Coupage d'une liste en deux

Algorithme 4 : Coupage d'une liste en deux

Données : Une liste L
Résultat : Deux liste L_1 et L_2
Idée : Tant que L n'est pas vide, on ajoute alternativement à L_1 ou L_2 la tête de L
Fonction : *coupe*(L)

```
L1 ←liste vide ;
L2 ←liste vide ;
cible ← 1 ;
tant que L n'est pas vide faire
    v ← tête de L ;
    si cible = 1 alors
        ajouter v à L1 ;
        cible ← 2
    sinon
        ajouter v à L2 ;
        cible ← 1
    fin
    retirer v de L ;
fin
end
```

Pour écrire l'algorithme du tri fusion , on suit donc le schéma suivant :

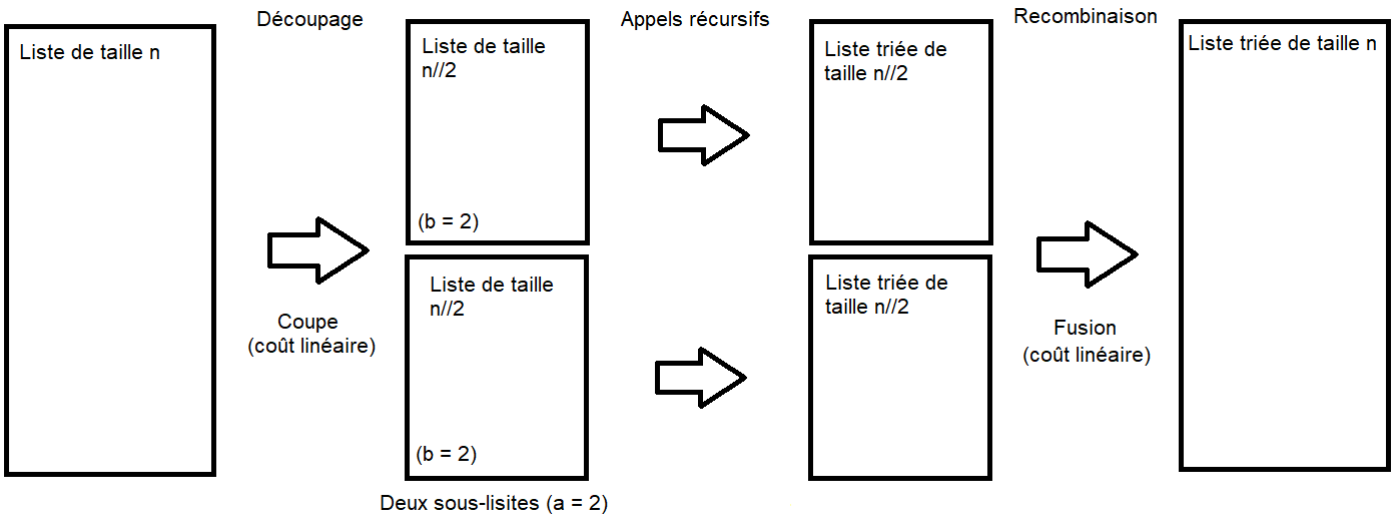

FIGURE 6 – Schéma récursif du tri fusion

Cela nous permet enfin d'écrire l'algorithme de tri fusion :

Algorithme 5 : Tri fusion d'une liste

Données : Une liste L
Résultat : Aucun, une fois l'algorithme terminé, L est triée
Idée : Couper, trier récursivement, fusionner
Fonction : $tri\_fusion(L)$
  si *L n'est ni vide ni L de taille 1* **alors**
    $L_1, L_2 \leftarrow couper(L)$;
    $tri\_fusion(L1)$;
    $tri\_fusion(L2)$;
    $L \leftarrow fusion(L1, L2)$
  fin
end

Comme on va le voir dans la prochaine partie, le tri fusion a une complexité en $\mathcal{O}(n\log_2(n))$, soit une meilleure complexité que les tris par insertion et sélection (quadratiques) étudiés l'an dernier. En fait, on peut même démontrer que tout algorithme de tri fonctionnant sans information sur les données stockées dans la liste / tableau est au mieux de cette complexité $n\log_2(n)$. En ce sens et même s'il existe bien d'autres algorithmes de tri, le tri fusion est l'un des meilleurs possibles.

4 BONUS : LE MASTER THEOREM

Le *master theorem* est un théorème permettant en général de connaître la complexité d'un algorithme utilisant la méthode diviser pour régner. Il est complètement hors programme et je ne le présente ici que dans une démarche d'approfondissement du cours.

Considérons un problème traité par la méthode diviser pour régner. On note en fonction de la taille n des données :
— $a > 1$ le nombre de sous-problèmes appelés récursivement ;
— $b > 1$ le facteur de division de la taille des sous-problèmes ;
— $T(n)$ le nombre d'opérations réalisées par l'algorithme pour des données de taille n ;
— $f(n)$ le nombre d'opérations nécessaires pour recombiner les réponses aux différents sous problèmes

Avec ces notations, on peut établir une formule de récurrence décrivant le nombre d'opération $T(n)$ en fonction de $T(\frac{n}{b})$:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

On suppose de plus que la recombinaison ait une complexité polynomiale $f(n) = \mathcal{O}(n^d)$ et on pose $c = \log_b(a)$.

Théorème (*master theorem*, version faible) :

1. si $d < c$ alors $T(n) = \mathcal{O}(n^c)$;
2. si $d = c$ alors $T(n) = \mathcal{O}(n^d\log_b(n))$;
3. si $d > c$ alors $T(n) = \mathcal{O}(n^d)$.

Exemples d'application sur les problèmes précédents :
— pour la recherche dichotomique, on coupe le tableau en deux mais on ne rappelle la fonction que sur l'un d'eux, soit $a = 1$, $b = 2$, $c = \log_2(1) = 0$ et la recombinaison est en temps constant soit $d = 0$. On a $d = c$ donc le *master theorem* nous assure une complexité en $\mathcal{O}(n^0\log_2(n)) = \mathcal{O}(\log_2(n))$.
— pour le tri fusion, on coupe le tableau en deux et on rappelle la fonction sur les deux sou-tableaux, soit $a = b = 2$, $c = \log_2(2) = 1$ et la recombinaison est linéaire soit $d = 1$. On a $d = c$ donc le *master theorem* nous assure une complexité en $\mathcal{O}(n^1\log_2(n)) = \mathcal{O}(n\log_2(n))$.

Remarque 1 : Pour comprendre ce théorème, il faut voir les nombres c et d comme des niveaux de prédominance (respectivement) des appels récursifs et de la recombinaison. Les différents cas peuvent ainsi se lire comme 1. *les appels récursifs sont prédominants*, 2. *les appels récursifs et la recombinaison sont équivalents* et 3. *la recombinaison est prédominante.*

Remarque 2 : Il existe une version plus précise de ce théorème, mais dont la formulation appelle des notations qu'on n'introduira pas cette années.

4.1 Comprendre le *master theorem* : recombinaison prédominante

Prenons un cas extrême en imaginant que $a = 0$ (pas d'appel récursif). Dans ce cas, la formule de récurrence devient

$$T(n) = f(n) = \mathcal{O}(n^d)$$

On est bien dans le cas 3. du théorème.

4.2 Comprendre le *master theorem* : appels récursifs prédominants

À l'extrême inverse, si $f(n) = 0$ (pas de recombinaison) alors la formule de récurrence devient

$$T(n) = aT(\frac{n}{b})$$

Une telle relation de récurrence implique que $T(n) = \mathcal{O}(n^c)$. Démontrons-le :

Réduction du problème à l'étude de $T(b^k)$: Quelque soit $n \in \mathbb{N}$,il existe $k \in \mathbb{N}$ tel que

$$b^k \leq n < b^{k+1}$$

En effet, k peut être vu comme le nombre de chiffres de n dans son écriture en base b et plus précisément, $k = \lfloor \log_b(n) \rfloor$.

Appliquons la suite T qu'on admet croissante à cette inégalité. On obtient :

$$T(b^k) \leq T(n) < T(b^{k+1})$$

Le fait de pouvoir estimer $T(b^k)$ permettra donc d'estimer $T(n)$.

Étude de $T(b^k)$: On peut réécrire la formule de récurrence en fonction de b^k

$$T(b^k) = aT(\frac{b^k}{b}) = aT(b^{k-1})$$

La suite $(T(b^k))_{k\in\mathbb{N}}$ est géométrique de raison a, soit

$$T(b^k) = T(1)a^k$$

Retour à $T(n)$, conclusion : Des deux parties précédentes on peu déduire que

$$T(n) < T(b^{k+1}) = T(1)a^{k+1} = aT(1)a^k$$

Or $k = \lfloor \log_b(n) \rfloor \leq \log_b(n)$ donc

$$T(n) < aT(1)a^{\log_b(n)} = \mathcal{O}(a^{\log_b(n)})$$

Mais (un peu de gymnastique avec les puissances)

$$a^{\log_b(n)} = e^{\log_b(n)\ln(a)} = e^{\frac{\ln(n)}{\ln(b)}\ln(a)} = e^{\frac{\ln(a)}{\ln(b)}\ln(n)} = n^{\frac{\ln(a)}{\ln(b)}} = n^{\log_b(a)} = n^c$$

D'où

$$T(n) = \mathcal{O}(n^c)$$

On est bien dans le cas 1. du théorème.

4.3 Comprendre le *master theorem* : équivalence entre les appels récursifs et la recombinaison

On ne va pas détailler le cas 2. du *master theorem* mais on peut observer dans la complexité $\mathcal{O}(n^d \log_b(n))$ l'influence à la fois de la recombinaison (n^d) et des appels récursifs $\log_b(n)$.

Amusez-vous à utiliser le *master theorem* sur les algorithmes proposés dans les exercices !