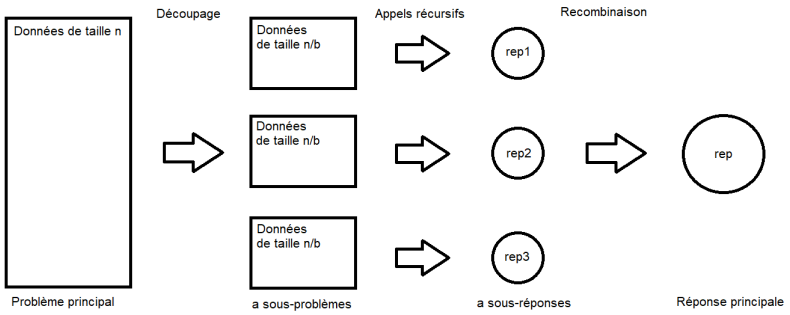
ans ce chapitre, on explore une méthode générale utilisée en algorithmique : la méthode diviser pour régner. Nous allons en voir le principe puis présenter deux exemples classiques d'algorithme mettant en oeuvre cette méthode.

1 Principe

a 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.



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.

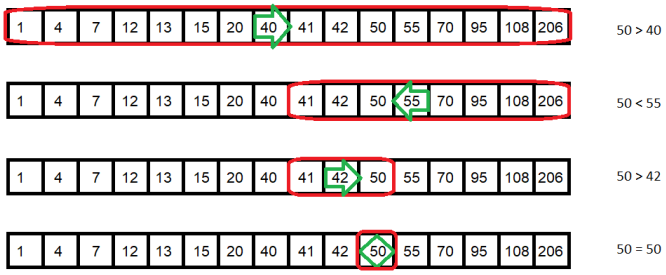
2 Retour sur la recherche dichotomique

2.1 Version impérative

On rappelle l'algorithme de recherche dichotomique (version impérative) dans un tableau trié qui est un exemple classique de stratégie diviser pour régner.

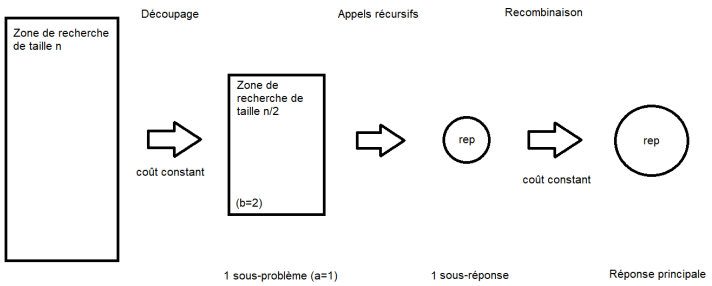
```
Recherche de x sur t :
    # t est supposé trié et de taille n
    a <- 0
    b <- n-1
    res <- FAUX
    Tant que a<b et non(B) :
        m <- (a+b)//2
        Si t[m]=x :
            res <- VRAI
        Si t[m]<x :
            a <- m+1
        Si t[m]>x :
            b <- m-1
    renvoyer res
```

Dans l'exemple suivant, on cherche la valeur 50 dans un tableau trié :



2.2 Version récursive

Pour donner une définition récursive à l'algorithme de recherche par dichotomie, dessinons-en le schéma récursif :



On obtient l'algorithme suivant :

```
Recherche de x sur t entre a et b :
    # t est supposé trié et de taille n
    Si a > b :
        renvoyer FAUX
    Sinon :
        m <- (a+b)//2
        Si t[m] = x :
            renvoyer VRAI
        Si t[m] < x :
            rechercher récursivement x entre m+1 et b
        Si t[m]>x :
            rechercher récursivement x entre a et m-1
```

et 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 !

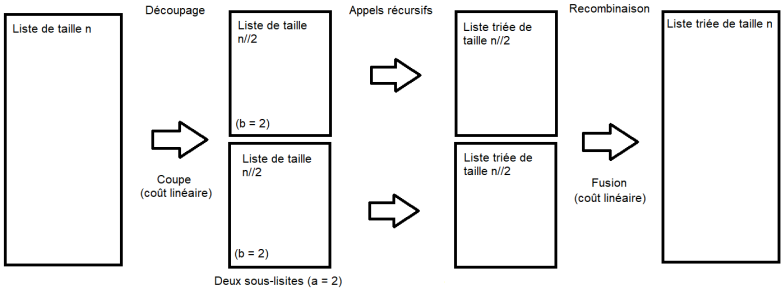
3 Tri fusion

3.1 Principe et exemple

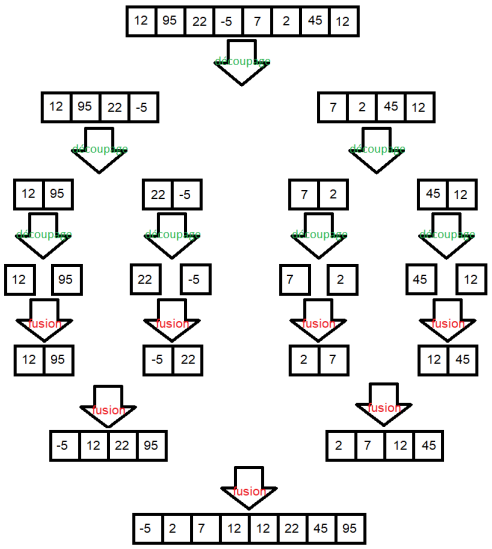
On s'intéresse au tri fusion (*merging sort*). Le principe du tri fusion est le suivant :

— Une liste vide ou de taille 1 est triée.
— Si la liste n'est pas vide, on la coupe en deux. Peu importe comment ce **découpage** à lieu. Cela est faisable avec un coût linéaire.
— On trie **récursivement** les deux sous-listes.
— On effectue la **fusion** des deux listes triées de sorte que la liste résultats en comporte toute les valeurs et soit toujours triée. Cela est faisable avec un coût linéaire.

On suit donc le schéma récursif suivant :

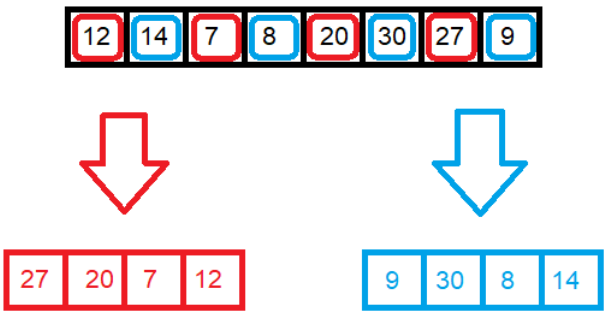


Exemple : tri fusion du tableau `[12, 95, 22, -5, 7, 2, 45, 12]`



3.2 Tri fusion : fonctions auxiliaires

Nous avons 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 :



L'algorithme est le suivant :

```
Couper une liste L en deux :
    L1 <- liste vide
    L2 <- liste vide
    Tant que L n'est pas vide :
        retirer v, la tête de L
        ajouter v à la liste L1 ou L2 en alternant
    renvoyer L1 et L2
```

Remarque : alterner entre `L1` et `L2` peut être fait à l'aide d'une variable `dest = 1 ou 2` changeant de valeur à chaque tour de la boucle et indiquant vers quelle liste on veut envoyer la valeur.

On peut aussi stocker les deux listes dans un tableau ou un tuple et insérer la valeur dans la liste de position `i%2` où i est un indice de boucle à maintenir.

La fusion de deux listes triées ce fait en suivant l'algorithme suivant :

```
Fusion de L1 et L2 triées
    L <- liste vide
    Tant que L1 et L2 sont non-vides :
        v1 <- la tête de L1
        v2 <- la tête de L2
        Si v1 < v2 :
            ajouter v1 à L
            retirer v1 de L1
        Sinon :
            ajouter v2 à L
            retirer v2 de L2
    Tant que L1 n'est pas vide :
        retirer la tête de L1 et l'ajouter à L
    Tant que L2 n'est pas vide :
        retirer la tête de L2 et l'ajouter à L
```

3.3 Algorithme et complexité

Nos fonctions auxiliaires nous permettent enfin d'écrire l'algorithme de tri fusion :

```
Tri fusion d'une liste L :
    si L est vide ou de taille 1 :
        renvoyer L
    sinon :
        couper L en L1 et L2
        par tri fusion, trier L1
        par tri fusion, trier L2
        renvoyer la fusion de L1 et L2
```

Comme on va le voir dans la prochaine partie, le tri fusion a une complexité en $\mathcal{O}(n\log_2(n))$. On parle d'algorithme **quasi-linéaire**.

Il s'agît d'une meilleure complexité que les tris par insertion et sélection (quadratiques) étudiés en classe de première.

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. 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*

4.1 Énoncé et applications

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 :
- $T(n)$ le nombre d'opérations réalisées par l'**algorithme complet** pour des données de taille n ;
- $a \geq 1$ le **nombre de sous-problèmes** appelés récursivement ;
- $b \geq 2$ le facteur de division de la **taille des sous-problèmes** ;
- $f(n)$ le nombre d'opérations effectuées par les **fonctions auxiliaires** (principalement les fonctions de découpage et recombinaison).

Avec ces notations, on peut établir une formule de récurrence donnant $T(n)$ en fonction de $T(\frac{n}{b})$:

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

On suppose de plus que les fonctions auxiliaires ont ensembles une complexité polynomiale $f(n) = \mathcal{O}(n^d)$ et on pose $c = \log_b(a)$.

Théorème (*master theorem*, version faible) – Avec les notations définies plus haut, on a :

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

Appliquons le *master theorem* à l'algorithme de recherche par dichotomie :

— **Appels récursifs :** On n'a qu'**un seul** appel récursif sur un tableau dont la taille a été divisée par **deux**, d'où :

$$a = 1 \; b = 2 \; c = \log_b(a) = \log_2(1) = 0$$

— **Fonctions auxiliaires :** Le découpage se fait en temps constant puisqu'il ne s'agît que que modifier les variables `a`, `b` ou `m`. Il n'y pas en outre pas de recombinaison, donc :

$$f(n) = \mathcal{O}(1) = \mathcal{O}(n^0) \implies d = 0$$

Nous devons maintenant comparer $d = 0$ à $c = 0$, soit ici $c = d$. Le *master theorem* nous dit donc que :

$$T(n) = \mathcal{O}(n^d \log_b(n)) = \mathcal{O}(n^0 \log_2(n)) = \mathcal{O}(\log_2(n))$$

Appliquons le *master theorem* à l'algorithme de tri fusion :

— **Appels récursifs :** On a **deux** appels récursif sur des listes dont la taille a été divisée par **deux**, d'où :

$$a = b = 2 \; c = \log_b(a) = \log_2(2) = 1$$

— **Fonctions auxiliaires :** On a vu que le découpage et la recombinaison (fusion) se font avec un coût linéaire. D'où :

$$f(n) = \mathcal{O}(n) = \mathcal{O}(n^1) \implies d = 1$$

Nous devons maintenant comparer $d = 1$ à $c = 1$ soit ici $c = d$. Le *master theorem* nous dit donc que :

$$T(n) = \mathcal{O}(n^d \log_b(n)) = \mathcal{O}(n^1 \log_2(n)) = \mathcal{O}(n \log_2(n))$$

Remarque 1 : Il existe une version plus précise de ce théorème, mais dont la formulation appelle des notations qu'on n'introduira pas cette année.

Remarque 2 : Pour comprendre le *master theorem*, il faut voir les nombres c et d respectivement comme des **niveaux de prédominance** des appels récursifs et des fonctions auxixiliaires dans la complexité de l'algorithme. Les différents cas du théorème peuvent ainsi se lire comme :

1. $c > d$: les appels récursifs sont prédominants :
$$T(n) = \mathcal{O}(n^c)$$
2. $c = d$: les appels récursifs sont équivalent aux fonctions auxiliaires ;
$$T(n) = \mathcal{O}(n^d \log_b(n))$$
3. $c < d$: les fonctions auxiliaires sont prédominantes.
$$T(n) = \mathcal{O}(n^d)$$

4.2 Le cas 3 : fonctions auxiliaires prédominantes

Pour comprendre le cas 3 du théorème, considérons une situation extrême en imaginant que l'on ait pas d'appels récursif et donc que des fonctions auxiliaires.

Dans ce cas, on ne peut définir b ni c mais le nombre d'opérations s'écrit bien :

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

4.3 Le cas 1 : appels récursifs prédominants

Pour comprendre le cas 1 du théorème, à l'extrême inverse, imaginons qu'on ait pas de fonctions auxiliaires ($f(n) = 0$) et donc que des appels récursifs.

Dans ce cas, la formule de récurrence devient :

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

Or, 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 $T(n)$ et 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 comme annoncé par le théorème :

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

4.4 Le cas 2 : équivalence entre les appels récursifs et les fonctions auxiliaires

On ne va pas détailler le cas 2. du *master theorem* (il y a assez de bonus comme ça !) mais on peut observer dans la complexité :

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

l'influence à la fois des fonctions auxiliaires (n^d) et des appels récursifs ($\log_b(n)$).