

Diviser pour régner

Lundi 12 Mars 2018

Michael FRANÇOIS

francois@esiea.fr



Méthode diviser pour régner

- Beaucoup d'algorithmes pratiques sont d'essence récursive : pour résoudre le problème, ils s'appellent eux-mêmes, une ou plusieurs fois pour traiter des sous-problèmes très similaires.
- Ces algorithmes suivent généralement une approche **diviser-pour-régner** :
 - c'est-à-dire, ils séparent le problème en plusieurs sous-problèmes semblables au problème initial mais de taille moindre, résolvent les sous-problèmes de façon récursive, puis combinent toutes les solutions pour produire la solution du problème original.

Le paradigme **diviser-pour-régner** implique trois étapes à chaque niveau de la récursivité :

- ① **Diviser** : le problème en un certain nombre de sous-problèmes qui sont des instances plus petites du même problème.
- ② **Régner** : sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un sous-problème est suffisamment réduite, on peut toutefois le résoudre directement.
- ③ **Combiner** : les solutions correspondantes aux sous-problèmes pour produire la solution du problème original.

Questions / Réponses

Q : Comment choisir les "tailles" des sous-problèmes ?

R : Il est intuitif qu'il vaut mieux choisir des sous-problèmes de taille sensiblement égale (lorsque c'est possible bien-sur), par exemple :

- si N est pair, on peut prendre deux sous-problèmes de taille $N/2$
- si N est impair, on peut prendre deux sous-problèmes de taille semblables, soit $(N-1)/2$ et $(N+1)/2$ (C'est en général le plus efficace).

Q : Comment résoudre les sous-problèmes ?

R : Il suffit d'appliquer de manière récursive le principe de la méthode diviser-pour-régner, jusqu'à ce que le sous-problème puisse être résolu de manière simple.

Tri fusion

L'algorithme du tri fusion

- **Tri fusion = Tri + fusion**

- Le tri par fusion (*merge sort* en anglais) implémente une approche de type diviser pour régner très simple : la suite à trier est tout d'abord scindée en deux suites de longueurs égales à un élément près. Ces deux suites sont ensuite triées séparément avant d'être fusionnées.

- L'efficacité du tri par fusion vient de l'efficacité de la fusion : le principe consiste à parcourir simultanément les deux suites triées dans l'ordre croissant de leur éléments, en extrayant chaque fois l'élément le plus petit.

- L'algorithme du **tri par fusion** agit de la manière suivante :
 - **Diviser** : la suite de n éléments à trier en deux sous-suites de $n/2$ éléments chacune.
 - **Régner** : trier les deux sous-suites de manière récursive en utilisant le tri par fusion.
 - **Combiner** : fusionner les deux sous-suites triées pour produire la suite globalement triée.

NB : la récursivité s'arrête quand la séquence à trier a une longueur 1 (puisque'une suite de longueur 1 est déjà triée).

Fusion

- Comment fusionner deux listes $L1$ et $L2$ qui sont déjà ordonnées (triées) ?
- La fusion est très simple : on sélectionne le plus petit des deux premiers éléments pour le **déplacer** dans la nouvelle liste $L3$. Par exemple, si le premier élément de la liste $L1$ a été choisi, alors le second élément de cette liste devient le premier élément. On répète ce processus jusqu'à épuiser les deux listes $L1$ et $L2$.
À la fin de ce processus, la liste $L3$ sera triée.

Exemple :

T1	1	4	5	6	9
T2	2	3	4		
T					

T1	1	4	5	6	9
T2	2	3	4		
T	1				

T1	1	4	5	6	9
T2	2	3	4		
T	1	2			

T1	1	4	5	6	9
T2	2	3	4		
T	1	2	3	4	

T1	1	4	5	6	9
T2	2	3	4		
T	1	2	3	4	

T1	1	4	5	6	9
T2	2	3	4		
T	1	2	3		

T1	1	4	5	6	9
T2	2	3	4		
T	1	2	3	4	4


T1	1	4	5	6	9			
T2	2	3	4					
T	1	2	3	4	4	5	6	9

Exemple :

T1	1	4	5	6	9	<i>n1 termes</i>
T2	2	3	4			<i>n2 termes</i>

Tant qu'il reste des éléments dans les deux tableaux
on sélectionne le plus petit

T1	1	4	5	6	9
T2	2	3	4		
T	1	2	3	4	4



Quand on est au bout de l'un des tableaux
on recopie le reste de l'autre.

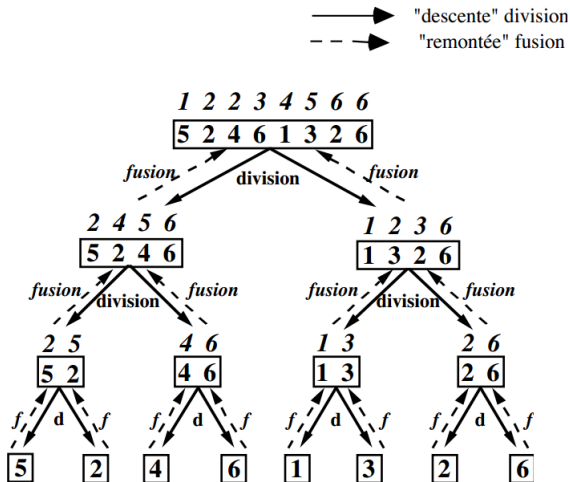
T1	1	4	5	6	9			
T2	2	3	4					
T	1	2	3	4	4	5	6	9

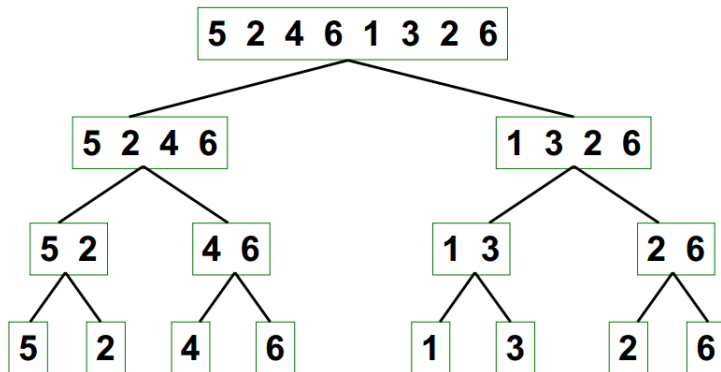
Tri fusion

Procédure récursive :

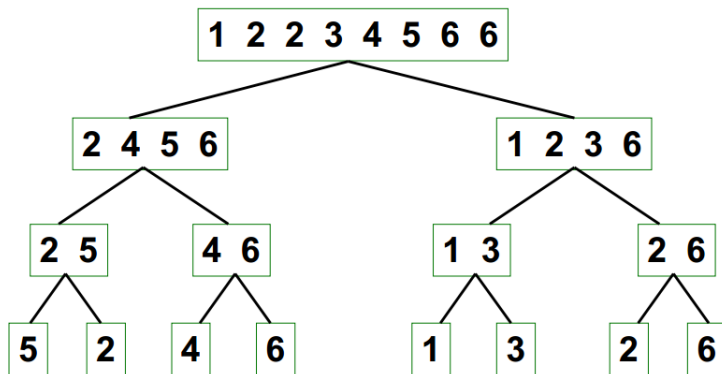
- diviser la séquence de n éléments en 2 sous-séquences de $n/2$ éléments (ou $n/2$ et $n/2 + 1$) ;
- trier chaque sous-séquence avec tri-fusion ;
- fusionner les 2 sous-séquences triées.

Exemple :



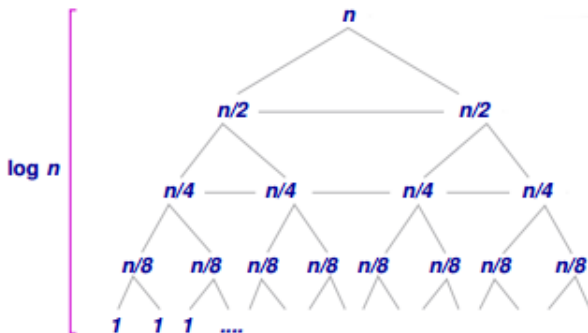


Division



Fusion

Complexité du tri fusion



Chaque nœud correspond à un appel de la fonction, ses fils correspondent aux deux appels récursifs. La hauteur de l'arbre est de $\log_2 n$ et à chaque niveau le cumul des traitements locaux est en $O(n)$, ce qui fait un coût total de $O(n) \times \log_2 n = O(n \log_2 n)$.

Ce qui fait du tri fusion, un très bon tri.

Recherche dichotomique

Recherche dichotomique

Soit T un tableau "déjà" trié de n éléments de type `TYPE` (`int`, `char` ou `float`), soit X , un élément de même type `TYPE` :

- ① trouver et renvoyer l'indice i s'il existe, tel que $T[i] == X$;
- ② ou alors renvoyer -1 si X n'appartient pas au tableau T .

Principe de la recherche dichotomique

- On cherche X dans le tableau $\{T[G], \dots, T[D]\}$
- On calcule $M = (G + D)/2$ qui est le milieu du tableau.
- On teste si $T[M] < X$:
 - si $T[M] < X$ alors on recommence la recherche dans le sous-tableau $\{T[M + 1], \dots, T[D]\}$
 - Sinon (comme on a $X \leq T[M]$), on recommence la recherche dans $\{T[G], \dots, T[M]\}$

- [WIKI] La **dichotomie** (« couper en deux » en grec) est en algorithmique, un processus itératif ou récursif de recherche où, à chaque étape l'espace de recherche est restreint à l'une des deux parties.
- [WIKI] On suppose bien sûr qu'il existe un test relativement simple permettant à chaque étape de déterminer l'une des deux parties dans laquelle se trouve la solution.
- L'algorithme s'applique typiquement à la recherche d'un élément dans un ensemble **fini ordonné** et organisé en séquence. À chaque étape, on coupera l'espace de recherche en deux parties de même taille (à un élément près) de part et d'autre de l'élément médian.

On peut décliner la méthode sous plusieurs variantes (peu nombreuses finalement) :

- La négation de $y < x$ c'est $y \geq x$
- La négation de $y \leq x$ c'est $y > x$
- La négation de $y > x$ c'est $y \leq x$
- La négation de $y \geq x$ c'est $y < x$

(Quelques exemples dans la suite)

Variante :

- On cherche X dans le tableau $\{T[G], \dots, T[D]\}$
- On calcule $M = (G + D)/2$ qui est le milieu du tableau.
- On teste si $T[M] \leq X$:
 - si $T[M] \leq X$ alors on recommence la recherche dans le sous-tableau $\{T[M], \dots, T[D]\}$
 - Sinon (comme on a $X < T[M]$), on recommence la recherche dans $\{T[G], \dots, T[M-1]\}$

Problème : reste à traiter la situation où $X == T[M]$: en clair où placer le test ?

- On cherche X dans le tableau $\{T[G], \dots, T[D]\}$
- On calcule $M = (G + D)/2$ qui est le milieu du tableau.
- On teste si $T[M] \leq X$:
 - On teste si $T[M] == X$, si oui alors on stoppe en renvoyant M ,
 - sinon on recommence la recherche dans le sous-tableau $\{T[M + 1], \dots, T[D]\}$
- Sinon (comme on a $T[M] > X$), on recommence la recherche dans $\{T[G], \dots, T[M - 1]\}$

Exemple [WIKI] :

Julie propose à Paul le jeu suivant :

« choisis en secret un nombre compris entre 0 et 100 ; je vais essayer de le deviner le plus rapidement possible, en ne pouvant que te poser des questions auxquelles tu réponds par oui ou par non ».

Paul choisit 66 et attend les questions de Julie :

- Julie sait que le nombre est entre 0 et 100 ; au milieu se trouve 50, elle demande donc : « Est-ce que le nombre est plus grand que 50 ? ». Paul répond « Oui ».
- Julie sait maintenant que le nombre est entre 51 et 100 ; au milieu se trouve 75, elle demande donc : « Est-ce que le nombre est plus grand que 75 ? » Paul répond « Non »
- Et ainsi de suite : « Plus grand que 63? ($63 = (51+75)/2$) », « Oui », « Plus grand que 69? ($69 = (63+75)/2$) », « Non », « Plus grand que 66? ($66 = (69+63)/2$) », « Non », « Plus grand que 65? ($65 \approx (63+66)/2$) », « Oui ».
- Julie sait maintenant que le nombre est entre 66 et 66, autrement dit qu'il s'agit de 66 : elle a trouvé le nombre choisi par Paul.

Remarques :

- Cette méthode itérative permet à Julie de trouver le nombre en posant en moyenne moins de questions que si elle procédait par des questions du type « Est-ce que le nombre est plus petit que 10 ?, 20?, 30 ?, etc. »
- Le nombre maximal M de questions à poser afin d'obtenir la réponse est la valeur du premier exposant entier de 2 supérieur ou égal au nombre N de réponses possibles, ou encore M est le premier entier supérieur ou égal au logarithme en base 2 de N .
- $M = \lceil \log_2 N \rceil$

Dans cet exemple il y a $N = 101$ réponses possibles, ce qui nous donne :

$$M = \lceil \log_2 101 \rceil = \lceil 6.658 \rceil = 7$$

Exemple en C :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void Recherche_Dicho_Nbre(int Nbre)
{
    int n, compt;
    n=101; compt=1;
    printf("saisir n :\n");
    scanf("%d", &n);
    while (n != Nbre)
    {
        if (n < Nbre)
        {printf("Le nombre mystère est plus grand que %d\n", n); }
        else
        {printf("Le nombre mystère est plus petit que %d\n", n); }
        printf("saisir n :\n");
        scanf("%d", &n);
        compt++;
    }
    if (n == Nbre)
    {printf("Nombre mystère trouvé au bout de %d coups\n", compt);}
}

int main(int argc, char** argv)
{
    int Nbre;
    srand(time(NULL));
    Nbre = rand() %101;
    Recherche_Dicho_Nbre(Nbre);
    return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
saisir n :
50
Le nombre mystère est plus grand que 50
saisir n :
75
Le nombre mystère est plus petit que 75
saisir n :
66
Le nombre mystère est plus grand que 66
saisir n :
70
Le nombre mystère est plus grand que 70
saisir n :
73
Le nombre mystère est plus grand que 73
saisir n :
74
Nombre mystère trouvé au bout de 6 coups
```

Un autre exemple en C :

```
#include <stdio.h>

int Recherche_Dichotomique(int L, int TAB[L], int ELT)
{
    int G, D, M;
    G=0; D=L;
    while (D >= G)
    {
        M = (D+G)/2;
        if (ELT < TAB[M])
            {D = M-1;}
        else
            {G = M+1;}
        if (ELT == TAB[M])
            {return M;}
    }
    return -1;
}

int main(int argc, char** argv)
{
    int ELT;
    int TAB[16] = {0, 1, 3, 5, 7, 8, 9, 11, 12, 18, 25, 26, 49, 77, 78, 94};
    while (scanf("%d", &ELT) != EOF)
    {
        printf("indice = %d\n", Recherche_Dichotomique(15, TAB, ELT));
    }
    return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
0
indice = 0
77
indice = 13
94
indice = 15
18
indice = 9
24
indice = -1
1
indice = 1
$
```

Complexité (Rappel)

Complexité (Rappel)

- La théorie de la complexité étudie formellement la difficulté intrinsèque des problèmes algorithmiques :
 - la théorie de la complexité s'attache à connaître la difficulté (ou la complexité) d'une réponse par l'algorithme à un problème dit algorithmique, posé de façon mathématique.
 - la théorie de la complexité vise à savoir si la réponse à un problème peut être donnée très efficacement, efficacement ou au contraire être inatteignable en pratique (et en théorie), avec des niveaux intermédiaires de difficulté entre les deux extrêmes ; pour cela elle se fonde sur une estimation théorique des temps de calcul et des besoins en mémoire informatique.

Recherche du minimum dans un tableau

- Soit un ensemble constitué de n éléments.
 - Quel est le nombre de comparaisons pour trouver le minimum ?
 - ???????
 - Ainsi la complexité en temps (*i.e.* nombre de comparaisons) de la recherche du minimum est : exactement ??????? itérations.
 - La complexité en temps dans le meilleur des cas est égale à la complexité dans le pire des cas donc, égale à la complexité en moyenne en temps : **très rare !!!**
- NB** : pourquoi se préoccuper de la complexité de la recherche du minimum d'un tableau (*a priori* non trié) ? parce que c'est une borne supérieure de la complexité de la recherche dichotomique dans un tableau trié.

Recherche du minimum dans un tableau

- Soit un ensemble constitué de n éléments.
 - Quel est le nombre de comparaisons pour trouver le minimum ?
 - facile : $n - 1$ fois.
 - Ainsi la complexité en temps (*i.e.* nombre de comparaisons) de la recherche du minimum est : exactement $n - 1$ itérations.
 - La complexité en temps dans le meilleur des cas est égale à la complexité dans le pire des cas donc, égale à la complexité en moyenne en temps : **très rare !!!**
- NB** : pourquoi se préoccuper de la complexité de la recherche du minimum d'un tableau (a priori non trié) ? parce que c'est une borne supérieure de la complexité de la recherche dichotomique dans un tableau trié.

Recherche dichotomique

- Soit un ensemble de n éléments, combien de fois peut-on :
 - séparer en deux sous-ensembles approximativement de même taille l'ensemble de départ ?
 - ce qui donne deux sous-ensembles, on en garde un et on oublie l'autre.
- Réponse : $p = \log_2 n$ fois, car il existe p tel que :

$$2^{p-1} < n \leq 2^p$$

- Donc :

$$p - 1 < \log_2 n \leq p$$

- Ainsi la complexité maximale en temps de la recherche dichotomique (dans le cas pire) en nombre de comparaisons est : au maximum $\log_2 n$ comparaisons, soit $O(\log_2 n)$.
- La complexité en temps dans le meilleur de cas : 1 comparaison !
- La complexité en moyenne est en $O(\log_2 n)$ comparaisons.

Exercice extrait du contrôle du 18 Mars 2015

```
fonction Ma_FONCTION (U, V, E, F)
  /* U, V, E sont des réels et F une fonction de  $\mathbb{R} \rightarrow \mathbb{R}^*$  /
  si (V - U ≤ E) alors
    | Retourner V
  fin
  m ← (U + V) / 2
  si (F(U) · F(m) ≤ 0) alors
    | Retourner MA_FONCTION (U, m, E, F)
  fin
  sinon
    | Retourner MA_FONCTION (m, V, E, F)
  fin
```

À quoi sert exactement la fonction MA_FONCTION ?

Solution :

MA_FONCTION sert à calculer via la méthode de dichotomie, le zéro d'une fonction réelle $F(x)$ sur l'intervalle $[U, V]$, avec une précision E . La fonction F ne s'annule qu'une seule et unique fois sur $[U, V]$.

Bibliographie

- R. ERRA, "Cours 6 d'informatique", 1A-S2 2013-2014 ESIEA.
- Cormen, Leiserson, Rivest, Stein "Algorithmique", 3ème éd. DUNOD.
- <http://pageperso.lif.univ-mrs.fr/~francois.denis/algoL2/chap4.pdf>
- <http://deptmedia.cnam.fr/new/spip.php?pdoc3352>