

La technique « **diviser pour régner** » (*divide and conquer*) est un paradigme algorithmique basé sur une approche récursive, qui consiste à diviser un problème en plusieurs sous-problèmes identiques, de plus petites tailles, jusqu'à parvenir à un problème tellement simple que sa résolution est immédiate. **Cette méthode est efficace si la taille des sous-problèmes suit une loi géométrique (par exemple si elle est divisée par deux lors de chaque appel), la profondeur des appels est alors logarithmique en la taille du problème initial.** Le calcul récursif du terme u_n d'une suite à partir de u_{n-1} n'est pas du type diviser pour régner.

Quelques exemples classiques qui illustrent ce concept dans le cours d'algorithmique :

- Algorithme d'exponentiation rapide
- Recherche dichotomique dans une liste triée
- Recherche dichotomique d'une racine d'une équation transcendante $f(x) = 0$
- Algorithme de tri fusion
- Algorithme de tri rapide
- Produit de polynômes par la méthode de Karatsuba
- Produit de matrices par la méthode de Strassen
- Recherche de la plus petite distance dans un nuage de points
- Algorithme de transformée de Fourier rapide (FFT) de Cooley-Tukey
- Recherche d'une enveloppe convexe d'un nuage de points
- Comptage du nombre d'inversions

Typiquement lorsque le calcul de $f(mn)$ peut se ramener aux calculs de $f(m)$ et $f(n)$, la technique diviser pour régner peut être invoquée. Les cas les plus élémentaires conduisent à un seul sous-problème de taille inférieure : recherche dichotomique d'une racine, calcul du pgcd par l'algorithme d'Euclide. Ces algorithmes sont facilement dérécursifiés.

La correction d'un algorithme fondé sur le paradigme diviser pour régner est prouvée par récurrence et la complexité temporelle conduit à résoudre une relation de récurrence. C'est sa complexité intéressante qui fait préférer un tel algorithme, que l'on qualifie parfois de « rapide ».

Le paradigme diviser pour régner évoque l'idée de parcourir un arbre depuis la racine de manière à en découvrir toutes les feuilles, ces dernières représentant des cas de base, triviaux à traiter. La programmation récursive met directement en évidence cet **arbre des appels récursifs**.

La méthode de **recherche dichotomique** (*binary search, bisection*) dans un tableau fait partie des fondamentaux de l'algorithmique : elle consiste à appliquer la méthode « diviser pour régner », soit de manière itérative, soit de manière récursive. **La méthode suppose que le tableau initial est trié.** Cette contrainte est primordiale, il faut commencer par trier le tableau de données si besoin ; les meilleurs algorithmes de tri par comparaison ont une complexité temporelle dans le pire des cas en $O(n \ln n)$, quasi-linéaire en la taille du tableau.



Dangers de la recherche dichotomique

La recherche dichotomique est célèbre en algorithmique pour le nombre de programmeurs expérimentés qui se sont fait piéger en l'implémentant ! Il est impératif de représenter sur des schémas l'évolution des indices lors du déroulement de l'algorithme. Il est nécessaire de distinguer un tableau de taille pair d'un tableau de taille impair, même s'il est judicieux pour l'étude de complexité de regrouper ces deux cas en un seul.

Exercice 1 (Recherche dichotomique)

La recherche dichotomique s'applique à un tableau trié de taille n . Les éléments du tableau appartiennent donc à un type ordonné. La recherche dichotomique consiste à comparer l'élément cherché avec l'élément situé au milieu du tableau trié. S'ils sont différents, la recherche continue avec la moitié de tableau dans laquelle cet élément peut encore se trouver : on compare l'élément cherché avec l'élément central de la partie restante de tableau, de manière répétitive, jusqu'à ce que l'élément soit trouvé ou que la partie restante de tableau soit vide. On note $\llbracket a_k, b_k \rrbracket$ l'intervalle de recherche à la k -ième itération et n_k le nombre d'éléments dans cette partie du tableau. On choisit $m_k = \lfloor \frac{a_k + b_k}{2} \rfloor$ comme indice du milieu du tableau.

1. Quelles sont les expressions de a_0 , b_0 , n_0 et m_0 ? Exprimer n_k en fonction de a_k et b_k . Représenter les schémas utiles, en distinguant n_k pair ou impair et en déduire les valeurs possibles de n_{k+1} en fonction de n_k pour l'algorithme 1.1. de la page suivante. Même question pour l'algorithme 1.2. Majorer finalement n_k en fonction de n et k . En déduire que les algorithmes proposés terminent.
2. Montrer la correction des algorithmes 1.1 et 1.2. en exhibant des invariants de boucles.
3. Écrire une fonction impérative `recherche1(t, x)` qui retourne un booléen indiquant si l'élément x est présent dans le tableau trié t , en utilisant l'algorithme 1.1.
4. Quel est l'intérêt de l'algorithme 2 par rapport à l'algorithme 1?
5. Écrire une fonction impérative `recherche2(t, x)` qui retourne un booléen indiquant si l'élément x est présent dans le tableau trié t , en utilisant l'algorithme 1.2.
6. Écrire une fonction récursive `recherche3(t, x)` qui retourne un booléen indiquant si l'élément x est présent dans le tableau trié t , en utilisant l'algorithme 1.1. On veillera à ne pas transmettre de tableau mais seulement 2 indices lors des appels récursifs.
7. Écrire une fonction récursive `recherche4(t, x)` qui retourne un booléen indiquant si l'élément x est présent dans le tableau trié t , en utilisant l'algorithme 1.2. On veillera à ne pas transmettre de tableau mais seulement 2 indices lors des appels récursifs.
8. Montrer que dans tous les cas $b_k - a_k \leq \frac{n}{2^k} - 1$ pour l'algorithme 1.1 et en déduire le nombre d'itérations dans le pire des cas, en fonction de n .
9. En déduire la complexité temporelle de la recherche dichotomique dans un tableau trié de taille n .
10. Quelle est la complexité spatiale de la recherche dichotomique?
11. Écrire une fonction récursive `recherche5(t, x)` qui retourne la position d'une occurrence de x dans le tableau t ou qui renvoie (-1) si l'élément x est absent de t .

Du grec *dichotomia*, on retient l'idée de couper en deux parties. Une méthode dichotomique (*bisection method*, *binary search method*) consiste à diviser itérativement par deux (bisection) la largeur de l'intervalle de recherche.

La résolution d'une équation transcendante de type $f(x) = 0$ est une autre application classique du principe dichotomique :

It is possible to trap a root between bracketing values, and then hunt it down like a rabbit. [Numerical Recipes]

Algorithme 1.1 Recherche dichotomique dans un tableau trié

ENTRÉES : Un tableau T trié contenant n éléments et une valeur x recherchée

SORTIES : True si x est dans T et False sinon.

INITIALISATION

1. $a \leftarrow 0$
2. $b \leftarrow n - 1$

BOUCLE PRINCIPALE

3. **TANT QUE** $a \leq b$ **FAIRE**
 4. $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
 5. $y \leftarrow T[m]$
 6. **SI** $y = x$ **ALORS**
 7. **RETOURNE** TRUE
 8. **FIN SI**
 9. **SI** $y > x$ **ALORS**
 10. $b \leftarrow m - 1$
 11. **SINON**
 12. $a \leftarrow m + 1$
 13. **FIN SI**
 14. **FIN TANT QUE**
 15. **RETOURNE** FALSE
-

Algorithme 1.2 Recherche dichotomique dans un tableau trié

ENTRÉES : Un tableau T trié contenant n éléments et une valeur x recherchée

SORTIES : True si x est dans T et False sinon.

INITIALISATION

1. $a \leftarrow 0$
2. $b \leftarrow n - 1$

BOUCLE PRINCIPALE

3. **TANT QUE** $a < b$ **FAIRE**
 4. $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
 5. $y \leftarrow T[m]$
 6. **SI** $y \geq x$ **ALORS**
 7. $b \leftarrow m$
 8. **SINON**
 9. $a \leftarrow m + 1$
 10. **FIN SI**
 11. **FIN TANT QUE**
 12. **RETOURNE** $\text{BOOL}(T[a] = x)$
-