DIVISER POUR RÉGNER

À la fin de ce chapitre, je sais :

- ☞ énoncer le principe d'un algorithme de type diviser pour régner
- ☞ distinguer les cas d'utilisation de ce principe (sous-problèmes indépendants)
- ☞ évaluer la complexité d'un algorithme diviser pour régner

A Diviser pour régner

■ **Définition 1 — Algorithme de type diviser pour régner.** L'idée centrale d'un algorithme de type diviser pour régner est de décomposer le problème étudié en plusieurs sous-problèmes de taille réduite. Ces algorithmes peuvent éventuellement être exprimés récursivement et sont souvent très efficaces.

On distingue trois étapes lors de l'exécution d'un tel algorithme :

1. la division du problème en sous-problèmes qu'on espère plus simples à résoudre,
2. la résolution des sous-problèmes, c'est à cette étape que l'on peut faire appel à la récursivité,
3. la combinaison des solutions des sous-problèmes pour construire la solution au problème.

Ⓡ On devrait donc logiquement nommer ces algorithmes diviser, résoudre et combiner !

Vocabulary 1 — Divide and conquer ⇜⇝ diviser pour régner.

Très souvent[1], les algorithmes de type diviser pour régner sont exprimés récursivement. À partir d'une taille de problème $\mathcal{P}$ de taille n, la division en sous-problèmes aboutit soit à $n = 1$ soit à $n = s$, s étant alors une taille pour laquelle on sait résoudre le problème facilement et efficacement. Les étapes 7 et 9 de l'algorithme 1 sont facilement descriptibles à l'aide d'un arbre, comme le montre la figure 2. La hauteur de cet arbre peut être quantifiée. Elle sert notamment à calculer la complexité.

1. mais pas toujours

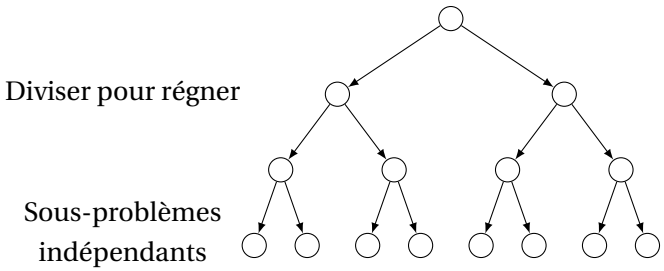

FIGURE 1 – Principe de la décomposition d'un problème en sous-problèmes indépendants pour un algorithme de type diviser pour régner.

Algorithme 1 Diviser, résoudre et combiner (Divide And Conquer)

```
1 : Fonction DRC(𝒫)                                   ▷ 𝒫 est un problème de taille n
2 :     r ← un entier ⩾ 1                             ▷ pour générer r sous-problèmes à chaque étape
3 :     d ← un entier > 1                             ▷ on divise par d la taille du problème
4 :     si n < s alors                                ▷ Condition d'arrêt, s est un seuil à déterminer
5 :         renvoyer RÉSOUDRE(n)
6 :     sinon
7 :         (𝒫_1, ..., 𝒫_r) ← Diviser 𝒫 en r sous-problèmes de taille n/d.
8 :         pour i de 1 à r répéter
9 :             S_i ← DRC(𝒫_i)                        ▷ Appels récursifs
10 :        renvoyer COMBINER(𝒮_1, ..., 𝒮_r)
```

Sur la figure 2, on a choisi de représenter un algorithme de type diviser pour régner dont le problème associé $\mathcal{P}$ est de taille n. L'étape de division en sous-problèmes divise par d le problème initial et nécessite r appels récursifs. Si $D(n)$ est la complexité de la partie division et $C(n)$ la complexité de l'étape de combinaison des résultats, alors on peut décrire la complexité $T(n)$ d'un tel algorithme par la relation de récurrence $T(n) = rT(n/d) + D(n) + C(n)$ et $T(s)$ une constante. Sur la figure 2 on a choisi $r = 3$ pour la représentation graphique.

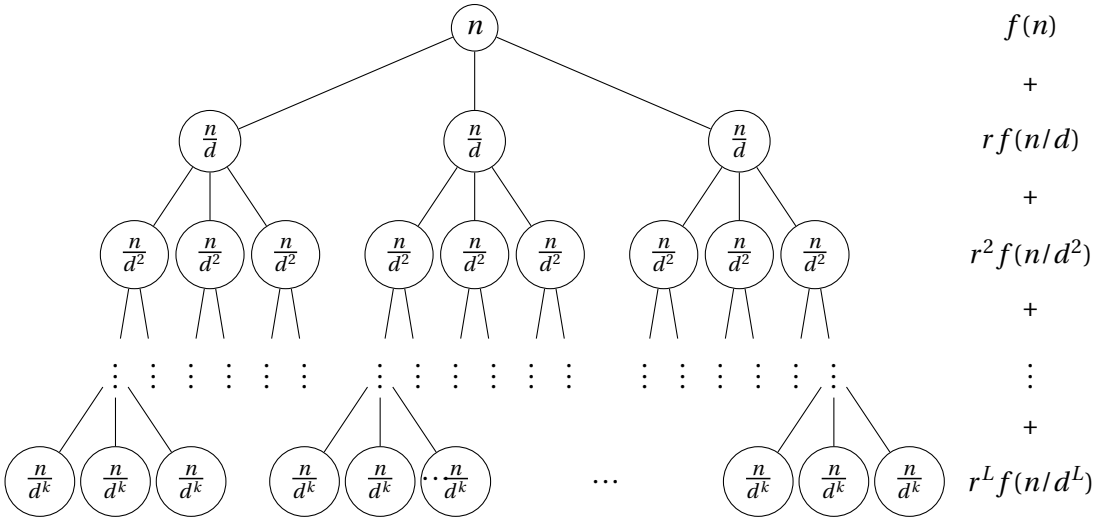

FIGURE 2 – Structure d'arbre et appels récursifs pour la récurrence : $T(n) = rT(n/d) + f(n)$ avec $n/d^k = s$. On a choisi $r = 3$ pour l'illustration, c'est-à-dire chaque nœud possède trois enfants au maximum : on opère trois appels récursifs à chaque étape de l'algorithme. La hauteur de l'arbre est en $\log_d n$.

B Exemple de la recherche dichotomique

■ **Exemple 1 — Recherche dichotomique.** L'algorithme de recherche dichotomique 2 est un exemple d'algorithme de type diviser pour régner : la division du problème en sous-problèmes est opérée via la ligne 5. La résolution des sous-problèmes est effectuée par des appels récursifs. La combinaison des résultats n'est pas explicite mais s'effectue sur le tableau lui-même grâce aux indices g et d.

R La recherche dichotomique est donc bien un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est-à-dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux.

Algorithme 2 Recherche récursive d'un élément par dichotomie dans un tableau trié

```
1 : Fonction REC_DICH(t, g, d, elem)
2 :     si g > d alors                                  ▷ Condition d'arrêt
3 :         renvoyer l'élément n'a pas été trouvé
4 :     sinon
5 :         m ← (g+d)//2                                ▷ Diviser
6 :         si t[m] = elem alors
7 :             renvoyer m
8 :         sinon si elem < t[m] alors
9 :             REC_DICH(t, g, m-1, elem)               ▷ résoudre
10 :        sinon
11 :            REC_DICH(t, m+1, d, elem)               ▷ résoudre
```

Grâce à la figure 3, on peut calculer le nombre d'opérations élémentaires $T(n)$ nécessaires à l'exécution de l'algorithme 2. On fait l'hypothèse que, hors appel récursif, la fonction REC_DICH nécessite un nombre constant d'opérations c. On peut expliciter formellement la relation de récurrence qui existe entre $T(n)$ et $T(n/2)$: on a $T(n) = T(n/2) + c$. on peut donc écrire :

$$\begin{aligned} T(n) &= T(n/2) + c && (1)\\ &= T(n/4) + c + c = T(n/4) + 2c && (2)\\ &= T(n/8) + 3c && (3)\\ &= \dots && (4)\\ &= T(n/2^k) + kc && (5)\\ &= T(1) + kc && (6) \end{aligned}$$

D'après l'algorithme 2, la condition d'arrêt s'effectue en un nombre constant d'opérations : $T(1) = O(1)$. Donc on a $T(n) = O(k)$. Or, on a $\frac{n}{2^k} = 1$. Donc $k = \log_2 n$ et $T(n) = O(\log n)$.

On peut également le montrer plus mathématiquement en considérant $k = \log_2 n$ et la suite $(u_k)_{k\in\mathbb{N}^\star}$ telle que $u_k = u_{k-1} + c$ et $u_1 = c$. C'est une suite arithmétique, $u_k = kc$. D'où le résultat.

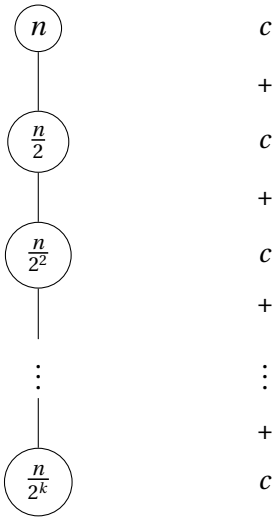

FIGURE 3 – Structure d'arbre et appels récursifs pour la récurrence de la recherche dichotomique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d'opérations c.

C Exemple de l'exponentiation rapide

L'algorithme naïf de l'exponentiation (cf. algorithme 3) qui permet d'obtenir a^n en multipliant a par lui-même n fois n'est pas très efficace : sa complexité étant en $O(n)$.

Algorithme 3 Exponentiation naïve a^n

```
1 : Fonction EXP_NAIVE(a,n)
2 :     api ← 1
3 :     pour i de 0 à n − 1 répéter
4 :         api ← api × a
5 :     renvoyer api
```

Or, l'exponentiation est une opération très récurrente qu'il est nécessaire de pouvoir exécuter le plus rapidement possible. L'exponentiation rapide (cf. algorithme 4) propose une version récursive de type diviser pour régner dont la complexité est en $O(\log n)$.

L'analyse de l'algorithme 4 montre que :

— c'est un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est-à-dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux[2],

— l'évolution du coût ne dépend pas de a mais de n, c'est-à-dire l'exposant.

On peut procéder de la même manière qu'avec l'algorithme 2 pour calculer la complexité et s'appuyer sur l'arbre de la figure 3. Pour simplifier le calcul, on peut considérer que la taille du

2. à une unité près si n est impair

Algorithme 4 Exponentiation rapide a^n

```
1 : Fonction EXP_RAPIDE(a,n)
2 :     si n = 0 alors                                  ▷ Condition d'arrêt
3 :         renvoyer 1
4 :     sinon si n est pair alors
5 :         p ← EXP_RAPIDE(a, n//2)                     ▷ Appel récursif
6 :         renvoyer p × p
7 :     sinon
8 :         p ← EXP_RAPIDE(a, n//2)                     ▷ Appel récursif
9 :         renvoyer p × p × a
```

problème est divisée par deux. Le coût hors appel récursif est constant car il s'agit de multiplications. On a donc $T(n) = O(\log n)$.

R L'algorithme 4 n'est pas à récursivité terminale. Sa version itérative est donnée par l'algorithme 5

Algorithme 5 Exponentiation rapide a^n (version itérative)

```
1 : Fonction EXP_RAPIDE_ITE(a, n)
2 :     resultat ← 1
3 :     base ← a
4 :     tant que n > 0 répéter
5 :         si n est impair alors                       ▷ Si le bit de poids faible est 1
6 :             resultat ← resultat × base
7 :         base ← base × base                          ▷ Élévation au carré pour le bit suivant
8 :         n ← n//2                                    ▷ Décalage vers la droite (passage au bit suivant)
9 :     renvoyer resultat
```