5 Techniques de récursion

5.1 Diviser pour régner

La technique de «**diviser pour régner**» (*divide-and-conquer*) exploîte la nature récursive de solutions optimales à une classe de problèmes. La démarche générale est de (1) couper le problème dans des sous-problèmes similaires, (2) chercher la solution optimale aux sous-problèmes par récursion, (3) combiner les résultats. Quand un problème de taille n est coupé en m sous-problèmes de taille n_i: $i = 1, \ldots, m$, le temps de calcul est déterminé par la récurrence $T(n) = \tau_{\text{couper}}(n) + \sum_{i=1,\ldots,m} T(n_i) + \tau_{\text{combiner}}(n)$, où $\tau_{\text{couper}}(n)$ et $\tau_{\text{combiner}}(n)$ sont les temps pour couper et combiner.

Exemple 5.1. On prend l'example de chercher le maximum dans un tableau.

```
    MAX-DR(x[0..n − 1], g, d) // trouve le max parmi x[g..d − 1]
    // appel initiel avec g = 0, d = n
D1  if d − g = 0 then return −∞                          // cas de base 0
D2  if d − g = 1 then return x[g]                        // cas de base 1
D3  mid ← ⌊(g + d)/2⌋                                    // diviser
D4  m1 ← MAX-DR(x, g, mid) ; m2 ← MAX-DR(x, mid, d)      // sous-problèmes par récurrence
D5  return max{m1, m2}                                   // combiner
```

Le temps de calcul de l'algorithme MAX-DR s'écrit par la récurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(1) \qquad \{n \geq 2\} \tag{5.1}$$

Théorème 5.1. *La solution de l'Équation* (5.1) *est* $T(n) = \Theta(n)$. ♠

Exemple 5.2. On a fourni une solution au Tours de Hanoï par le principe de «diviser pour régner» (Notes de cours, §1.6). Le temps d'exécution du réarrangement de n disques s'écrit par la récurrence

$$T(n) = 2T(n-1) + \Theta(1) \qquad \{n > 0\}$$

avec solution $T(n) = \Theta(2^n)$.

5.2 Récursion terminale

On peut toujours transformer une boucle en un algorithme récursif équivalent.

```
    MAX-ITER(x[0..n − 1])
M1  initialiser max ← −∞
M2  for i ← 0, . . . , n − 1 do
M3      if x[i] > max then max ← x[i]

M4  return max
```

```
    MAX-TERM(x[0..n − 1], i, max)        // variables locales de l'itération
    // appel initiel avec i = 0, max = −∞
T1  if i < n then                        // condition d'arrêt de l'itération
T2      if x[i] > max then max ← x[i]    // corps d'une itération
T3      return MAX-TERM(x, i + 1, max)   // boucler
T4  if i = n then return max             // après la boucle
```

La profondeur maximale de la pile d'exécution (§1.3, Notes 01) caractérise un aspect important de l'efficacité d'un algorithme récursif. La récurrence de MAX-DR (Exemple 5.1) nécessite une pile d'exécution de profondeur $\lceil \lg n \rceil$. Ce qui est pire, la récurrence naïve de MAX-REC de l'Exemple 4.3 peut mèner au

débordement de la pile d'exécution, à cause de n appels imbriqués. Par contre, si l'appel récursif est en **position terminale**, on n'a pas besoin d'attendre le retour de l'appel. On peut simplement transférer le contrôle sans sauvegarder le contexte : il suffit de remplacer le bloc d'activation (au lieu d'en empiler un nouveau). En conséquence, la profondeur maximale de la pile reste $\Theta(1)$. Les compilateurs modernes détectent des appels terminaux. Après compilation, MAX-ITER et MAX-TERM sont identiques. W(fr)

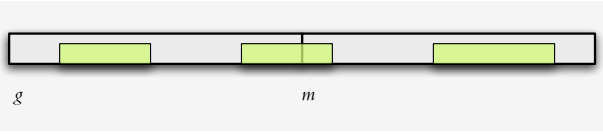
5.3 Programmation dynamique

Si les récurrences successives visitent des sous-problèmes identiques, il vaut mieux de se servir d'une approche de «récursion ascendante», ou de **programmation dynamique**, au lieu de recalculer la même valeur plus qu'une fois. W(en)

Exemple 5.3. Il serait très inefficace d'utiliser la récursion directement pour calculer le nombre Fibonacci $F(n)$. L'algorithme suivant calcule $F(n)$ en $\Theta(n)$ temps, avec un espace de travail de $O(1)$.

```
P1  Algorithme FIBPD(n)                    // algorithme efficace pour calculer F(n)
P2  f ← 0; i ← 0
P3  if n > i then f_prec ← f; f ← 1; i ← 1
P4  while n > i do                         // on a toujours f = F(i) et f_prec = F(i − 1)
P5      t ← f + f_prec; f_prec ← f; f ← t; i ← i + 1
P6  return f
```

Exemple 5.4. Dans le problème de la **sous-somme maximale** [J. Bentley, "Algorithm design techniques." *Communications of the ACM*, **27**(9) :865–871, 1984], on a un tableau $x[0..n-1]$ qui contient des valeurs négatives et positives. On définit la sous-somme $S(i,j)$ de chaque sous-tableau avec $0 \le i \le j \le n$: $S(j,j) = 0$ et $S(i,j) = \sum_{k=i}^{j-1} x[k]$ pour $i < j$. Comment doit-on trouver $\max_{i,j} S(i,j)$? Solution naïve : utiliser des boucles imbriquées $\rightarrow \Theta(n^2)$ temps. W(en)



Solution «diviser pour régner» : couper le tableau en deux, trouver les meilleurs sous-tableaux à gauche et droit par récursion, et comparer au meilleur sous-tableau qui traverse les deux côtés.

```
    SUBSUM-REC(x[0..n − 1], g, d)   // meilleur sous-somme en x[g..d − 1]
R1  if d − g = 0 then return 0
R2  if d − g = 1 then return max{0, x[g]}
R3  mid ← ⌊(g + d)/2⌋                                          // diviser
R4  s1 ← SUBSUM-REC(x, g, mid) ; s2 ← SUBSUM-REC(x, mid, d)
R5  s ← 0; s0 ← 0
R6  for i ← mid − 1, mid − 2, . . . , g do
R7      s ← s + x[i] ; if s > s0 then s0 ← s
R8  s ← s0
R9  for i ← mid, mid + 1, . . . , d do
R10     s ← s + x[i] ; if s > s0 then s0 ← s
R11 return max{s0, s1, s2}
```

Soit $T(m)$ le temps de calcul pour l'appel SUBSUM-REC$\big(x[0..n-1], g, d\big)$ avec $d - g = m$. On a la récurrence

$$T(m) = T(\lfloor m/2 \rfloor) + T(\lceil m/2 \rceil) + \Theta(m) \qquad \{m \ge 2\} \tag{5.2}$$

Théorème 5.2. *La solution de* (5.2) *est* $T(m) = \Theta(m \log m)$.

Sous-somme par programmation dynamique : Soit $S(j) = \max_{i=0,\ldots,j} S(i,j)$. Évidemment, $\max_{i,j} S(i,j) = \max_j S(j)$. La programmation dynamique utilise la récurrence $S(j) = \max\{0, S(j-1) + x[j-1]\}$ pour $j > 0$, avec la valeur initiale $S(0) = 0$. La programmation dynamique mène à un temps de calcul $\Theta(n)$.

```
SUBSUM-PD(x[0..n − 1])
S1  s ← 0; max ← 0
S2  for i ← 1, . . . , n do
S3      s ← s + x[i − 1]
S4      if s < 0 then s ← 0                          // glisser côté gauche
S5      if s > max then max ← s        // meilleure sous-somme finit par ici
S6  return max
```

5.4 Liste chaînée comme structure recursive

La liste (simplement) chaînée est une structure récursive. Une liste chaînée est construite par l'application des règles suivantes.

$$\langle liste \rangle \to \mathsf{null} \qquad \text{(liste vide)}$$
$$\langle liste \rangle \to \big(\langle noeud \rangle, \langle liste \rangle\big) \qquad \text{(liste commence par } \langle noeud \rangle\text{)}$$

Dans d'autres mots, une liste chaînée est soit une référence null, soit une paire d'un nœud (sa tête) et d'une liste.

On peut exploiter la structure récursive lors du parcours.

```
LENGTH(N)      // calcule le nombre de nœuds à partir de N
N1  if N = null then return 0
N2  else return 1 + LENGTH(N.next).
```

Exercice 5.1. *Compter le nombre de nœuds sur une liste chaînée par récursion terminale.*

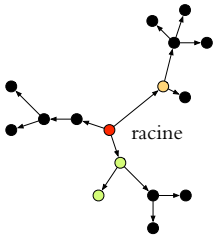
5.5 Arbres

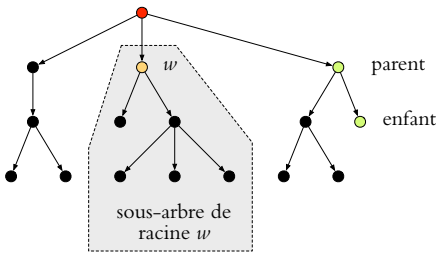
Un arbre est une structure récursive qui joue un rôle central dans la conception et analyse d'algorithmes :
- ⋆ structures de données explicites qui sont des réalisations concrètes d'arbres
- ⋆ arbres pour décrire les propriétés dynamiques des algorithmes récursifs
- ⋆ arbres de syntaxe

On considère des arbres enracinés (ou l'ordre des enfants n'est pas important) et les arbres ordonnés (comme l'arbre binaire, ou les enfants sont ordonnés dans une liste).

Définition 5.1. *Un **arbre enraciné** T ou arboresence est une structure définie sur un ensemble de nœuds qui*

1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et un ensemble d'arbres enracinés (les **enfants**)*





Définition 5.2. *Un **arbre ordonné** T est une structure définie sur un ensemble de nœuds qui*

1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et les arbres $T_0, T_1, T_2, \ldots, T_{d-1}$. La racine de T_i est apellé l'**enfant** de r étiqueté par i.*

*Le **degré** d'un nœud est le nombre de ses enfants : les nœuds externes sont de degré 0. Le degré de l'arbre est le degré maximal de ses nœuds. Un arbre k-aire est un arbre ordonné où chaque nœud interne possède exactement k enfants. Un* W(fr) *__arbre binaire__ est un arbre ordonné où chaque nœud interne possède exactement 2 enfants : les sous-arbres gauche et droit.*

5.6 Parcours d'un arbre

Dans un parcours, tous les nœuds de l'arbre sont visités. Dans un **parcours préfixe** (*preorder traversal*), chaque nœud est visité avant que ses enfants soient visités. Dans un **parcours postfixe** (*postorder traversal*), chaque nœud est visité après que ses enfants sont visités. Dans les algorithmes suivants, un nœud externe est null, et chaque nœud interne N possède les variables N.children (si arbre numeroté), ou N.left et N.right (si arbre binaire). L'arbre est stocké par une référence à sa racine root.

```
Algo PARCOURS-PRÉFIXE(x)
1 if x ≠ null then
2     «visiter» x
3     for y ∈ x.children do
4         PARCOURS-PRÉFIXE(y)
```

```
Algo PARCOURS-POSTFIXE(x)
1 if x ≠ null then
2     for y ∈ x.children do
3         PARCOURS-POSTFIXE(y)
4     «visiter» x
```

Maintenant PARCOURS-. . . (root) visite tous les nœuds dans l'ordre souhaité.

On peut parcourir un arbre binaire aussi dans l'ordre infixe. Dans un **parcours infixe** (*inorder traversal*), chaque nœud est visité après son enfant gauche mais avant son enfant droit.

```
Algo PARCOURS-INFIXE(x)
1 if x ≠ null then
2     PARCOURS-INFIXE(x.left)
3     «visiter» x
4     PARCOURS-INFIXE(x.right)
```

Un parcours préfixe ou postfixe peut se faire aussi à l'aide d'une pile. Si au lieu de la pile, on utilise une queue, alors on obtient un **parcours par niveau**.

```
Algo PARCOURS-PILE
1 initialiser la pile P
2 P.push(root)
3 while P ≠ ∅
4     x ← P.pop()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do P.push(y)
```

```
Algo PARCOURS-NIVEAU
1 initialiser la queue Q
2 Q.enqueue(root)
3 while Q ≠ ∅
4     x ← Q.dequeue()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do Q.enqueue(y)
```

Exercice 5.2. *Montrer comment faire un parcours postfixe, sans récursion, à l'aide d'une pile. **Indice :** placez des paires (x, e) sur la pile où e encode l'état du nœud x dans le parcours.*