4 Analyse d'algorithmes

4.1 Temps de calcul

Pour caractériser une structure de données, on analysera souvent le temps de calcul pour les **opérations** différentes. Typiquement, on veut arriver à un résultat tel que «l'opération search prend $O(\log n)$ temps au pire cas» où n mesure la taille de la strucure. Ce qu'on veut dire par la «taille», dépend du contexte. Ici, n est le nombre d'éléments dans une structure qui implante un dictionnaire. De même façon, on caractérisera un **algorithme** en disant que «algorithme Tel-et-tel prend $O(n)$ temps au pire cas», où n mesure la taille de l'entrée. En général, on veut arriver à une borne asymptotique en fonction de la taille de l'entrée et celle de la sortie.

Exemple 4.1. On prend l'exemple de rechercher le maximum dans un tableau. L'algorithme MAX-TABLEAU utilise une boucle. MAX-REC utilise une récurrence terminale équivalente à la version itérative.

```
MAX-TABLEAU(x[0..n − 1])
Entrée : tableau x de taille n ≥ 0
Sortie : valeur maximale parmi les x[i]
B1  initialiser max ← −∞
B2  pour i ← 0, . . . , n − 1 faire
B3      si x[i] > max alors max ← x[i]
B4  retourner max
```

```
MAX-REC(x[0..n − 1], i, max)
// appel initiel avec i = 0, max = −∞
Sortie : valeur maximale parmi x[i..n − 1] et m
R1  si i = n alors retourner max
R2  sinon
R3      si x[i] > max alors max ← x[i]
R4      retourner MAX-REC(x[], i + 1, max)
```

Pour MAX-TABLEAU, le temps de calcul est $T(n) = O(1) + \big(O(1) + O(1)\big) \cdot O(n) + O(1)$, donc $T(n) = O(n)$ (v. règles de l'arithmétique avec O). L'algorithme prend un temps linéaire dans la taille de l'entrée (dans tous les cas). Pour analyser le temps de calcul de MAX-REC, on a besoin de trouver la solution à la récurrence $T(n) = O(1) + T(n-1)$. (Exemple 3A.4) □

Exercice 4.1. *Écrire un algorithme itératif qui calcule la somme des éléments dans un tableau $x[0..n-1]$. Montrer la version équivalente avec récurrence terminale. Analyser le temps de calcul asymptotique.*

4.2 La «taille» dépend du contexte.

En Exemple 4.1, on mesure la taille par le nombre d'éléments au lieu du nombre de bits, en supposant que les opérations numériques en Lignes 2 et 3 peuvent s'exécuter en un temps $O(1)$. Mais dans une application quelconque où on peut avoir des nombres entiers sans borne (pas seulement *int* $\leq 2^{31} - 1$ etc.), il faut considérer la dépendance du **nombre de bits utilisés dans l'encodage** de x.

Exemple 4.2. On prend l'exemple de l'**algorithme d'Euclid** qui trouve le plus grand commun diviseur de deux entiers positifs.

W(fr)

```
E1  Algo gcd(a, b) // avec b ≤ a      Taille de l'entrée est lg a + lg b bits.
E2  tandis que (b ≠ 0)                O(log b)
E3      c ← a mod b                   O(log^2 a)
E4      a ← b                         O(log b)
E5      b ← c                         O(log c)
E6  retourner a                       O(log a)

                                      O(log a+log b) fois
```

Le temps de calcul est **polynomiel dans la taille de l'entrée**.
Preuve : On a $a = kb + c \geq (k+1)c \geq 2c$ en Ligne E3. Donc $bc \leq ab/2$. Par conséquence, le nombre d'itérations est borné par $\lg(ab) = \lg a + \lg b$. Si calculer «$a \bmod b$» prend $O(\log^2 a)$ temps, alors l'exécution est en temps $O(\log^3 a)$. □

4.3 Récurrences

Si on travaille avec un algorithme récursif, on obtient des récurrences asymptotiques.

Exemple 4.3. On prend l'exemple de la recherche de valeur maximale sur une liste chaînée. Dans l'implantation Java montrée ici, la méthode **maxValue**() se sert d'une méthode auxiliaire **maxValue**(`Node` N).

```
public int maxValue()
{
  return maxValue(premier);
}

private int maxValue(Node N) // méthode récursive auxiliare
{
  if (N == null)
    return Integer.MIN_VALUE; // la plus petite valeur possible
  else
    return Math.max(N.valeur, maxValue(N.prochain)); // récursion
}
```

Soit $T(n)$ le temps de calcul de la méthode `maxValue` quand la liste est de longueur $n \geq 0$. On a la récurrence $T(n) = O(1) + T(n-1)$ pour $n > 0$, d'où on obtient que $T(n) = O(n)$ (Exemple 3A.4). □

Exemple 4.4. On prend l'exemple du calcul de puissances. On veut calculer x^n où $n \geq 0$ est un entier. La clé à une solution efficace est la récurrence suivante

$$x^n = \begin{cases} 1 & \{n = 0\} \\ x^{n/2} \cdot x^{n/2} & \{n > 0, n \text{ est pair}\} \\ x \cdot x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \{n > 0, n \text{ est impair}\} \end{cases}$$

On a donc l'algorithme

```
P1 Algo PUISSANCE(x, n)
P2 si n = 0 alors retourner 1
P3 sinon
P4     y ← PUISSANCE(x, ⌊n/2⌋)
P5     si n est pair
P6     alors retourner y^2
P7     sinon retourner x y^2
```

Ceci mène à la récurrence $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1)$ pour $n > 0$, dont la solution est connue : $T(n) = O(\log n)$ (Exemple 3A.5).

Preuve par **substitution de variables** : par induction dans le nombre de bits dans la représentation binaire de n. Si on dénote ce nombre par b, on a la récurrence $T'(b) = T'(b-1) + O(1)$ avec la solution $T'(b) = O(b)$. Or, $b = \lceil \lg(n+1) \rceil = O(\log n)$, donc $T(n) = T'\big(O(\log n)\big) = O(\log n)$. □