Leçon 926 : Analyse des algorithmes : Complexité. Exemples.

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Carton, *Langages formels, calculabilité et complexité.*
[3] Cormen, *Algorithmique.*
[4] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Étude d'Union-Find pour la complexité

Algorithme de Dijkstra

Plan de la leçon

Motivation

Défense

Lors de la conception, puis de l'étude d'un algorithme, deux notions sont extrêmement importantes :

— la correction de l'algorithme : fait-il ce que l'on souhaite ?

— l'efficacité de l'algorithme : à quelle vitesse s'exécute-t-il ? ; est-il optimal ?

La complexité (temporelle ou spatiale) intervient alors pour comparer deux algorithmes corrects répondant à la même question.

Ce qu'en dit le jury

Il s'agit ici d'une leçon d'exemples. Le candidat prendra soin de proposer l'analyse d'algorithmes portant sur des domaines variés, avec des méthodes d'analyse également variées : approche combinatoire ou probabiliste, analyse en moyenne ou dans le pire cas.

Si la complexité en temps est centrale dans la leçon, la complexité en espace ne doit pas être négligée. La notion de complexité amortie a également toute sa place dans cette leçon, sur un exemple bien choisi, comme union find (ce n'est qu'un exemple).

1 Quantifier la complexité

Définir la complexité d'un algorithme n'est pas facile. Intuitivement la complexité d'un algorithme est un indicateur de la difficulté pour résoudre le problème traité par l'algorithme. Mais cette vue de l'esprits n'est pas simple à quantifier. Nous allons alors définir la complexité comme une fonction qui en fonction des entrées sur notre programme donnera la consommation d'une certaine ressource.

1.1 Qu'est-ce que la complexité ?

— *Définition* : donnée d'entrée d'un algorithme (= ensemble des variables externes à l'algorithmes sur lesquelles on exécute celui-ci)

— *Exemple* : un algorithme de tri sur un tableau prend un tableau en entrée

— *Définition* : taille d'une entrée : $f : \{entree\} \to \mathbb{N}^k$

— *Exemples* : mot : nombre de caractère ; tableau : nombre de cellule ou l'espace mémoire utilisé

— *Définition* : complexité comme fonction ϕ des entrées vers une ressources qui peut être temporelle, spatiale, le nombre d'accès à un disque externe, ...

— *Remarque* : en pratique on compte des unités de bases (peut être des cellules mémoires, des fonctions que l'on considère de bases)

— *Exemples* : complexité temporelle : tri bulle : $\frac{n(n-1)}{2}$; complexité disque dur : recherche B-arbre h où h hauteur ; complexité temporelle : plus longue sous-séquence commune

1.2 Mesurer la complexité [3, p.40]

On ne peut pas toujours calculer la complexité exacte (avec les constantes) car généralement, elle dépend de l'implémentation que nous utilisons. Pour cette même raison le calcul de la complexité exacte peut s'avérer inutile.

— *Définition* Notation de Landau (O et Θ) + abus de notation
— *Proposition* : Équivalence des notations + illustration
— *Exemples* : Tri bulle $O(n^2)$; B-arbre $O(\log n)$
— *Proposition* : L'ordre de croissance

1.3 Différentes approches de la complexité [4, p.19]

— *Définition* : Complexité pire cas, meilleur cas en moyenne (avec les probabilités)
— *Remarque* : Distribution uniforme : simplification de l'écriture mais pas toujours vrai Attention
— *Définition* : complexité constante

2 Techniques de calcul de la complexité

Maintenant que nous avons donné un cadre à notre théorie de la complexité, nous souhaitons calculer les complexités d'algorithmes usuels. Pour cela, nous allons présenter quelques techniques de calcul élémentaires.

2.1 Le calcul direct

On peut parfois calculer directement la complexité en dénombrant les opérations que l'on doit calculer. Efficace dans le cas d'algorithmes itératifs
Exemple : Recherche de maximum dans un tableau / CYK (Algorithme 1) $O(n^3)$ [2, p.198] / Marche de Jarvis (Algorithme 2) $O(hn)$ [1, p.389]

2.2 La résolution de récurrence [1, p.20]

On peut parfois exprimer la complexité pour une donnée de taille n par rapport à une donnée de taille strictement inférieure. Résoudre l'équation ainsi obtenue nous donne la complexité. Efficace dans le cas d'algorithmes récursifs.

— *Proposition* suite récurrente linéaire d'ordre 1 + *Exemples* Factorielle et Euclide
— *Proposition* suite récurrente linéaire d'ordre 2 + *Exemple* Fibonacci
— *Proposition* : Master theorem + *Exemples* Tri fusion et Strassen
— *Remarque* : Ne capture pas toutes les équations par récurrences.

2.3 Le calcul de la complexité moyenne par l'espérance

— *Remarque* : on a une distribution uniforme donc $c_{moy} = \frac{1}{|D_n|} \sum_{x \in D_n} \phi(x)$
— *Exemple* : tri rapide randomisé
— *Remarque* : on fait souvent l'hypothèse d'avoir une distribution uniforme sur les données de l'entrée, cependant c'est généralement faux et cela nous introduit des erreurs.

3 Raffinement de l'étude de la complexité : la complexité amortie

La complexité amortie n'est pas une complexité moyenne ! La complexité amortie est une amélioration de l'analyse dans le pire cas s'adaptant (ou calculant) aux besoins de performance des structures de données.

— *Définition* : la complexité amortie : $c_{amo} = \max_{op_1,\dots,op_k} \sum_{i=1}^{k} \frac{c(op_i)}{k}$

— *Exemple* : table dynamique où on souhaite la complexité d'un élément qu'on insère en nombre d'allocation et d'écriture [3, p.428]

3.1 Méthode de l'agrégat

Dans cette méthode, le coût amortie est le même pour toutes les opérations de la séquence même si elle contient différentes opérations

— *Principe* : On calcul la complexité dans le pire cas pour cette séquence. La complexité amortie d'une opération est donc cette complexité divisée par le nombre d'opérations de la séquence [3, p.418].

— *Exemples* : table dynamique, Union find + Kruskal DEV, balayage de Gram

3.2 Méthode comptable

Cette méthode se différencie de la précédente en laissant la possibilité que toutes les opérations ait un coup amortie différent.

— *Principe* : On attribut à chaque opération un crédit et une dépense (qui peuvent être différent de leur coût réel). Ces crédits sont plus importants pour les opérations coûteuses afin de rattraper leur dépenses importantes. Cependant, elles doivent vérifier la propriété suivante : $\sum_{i=1}^{k} cred(op_i) - \sum_{i=1}^{k} dep(op_i) \geq 0$. On a alors : $c_{amo}(op) = c_{reel}(op) + cred(op) - dep(op)$.

— *Exemples* : table dynamique, KMP

3.3 Méthode du potentiel

Cette méthode a été popularisé lors de la preuve de la complexité amortie de la structure de données Union Find, implémentée à l'aide d'une forêt et des heuristiques qui vont bien. Elle est moins facile que les autres à mettre en place.

— *Principe* : Au lieu d'assigner des crédits à des opérations, on va associer une énergie potentielle φ à la structure elle-même. Cette énergie vérifie les propriétés suivantes :$\varphi(\text{structure vide}) = 0$ et $\forall T, \varphi(T) \geq 0$. On a alors $c_{amo}(op) = c_{reel}(op) + \varphi(T) - \varphi(T')$ lorsque l'opération op permet de passer de T à T' [3, p.424].

— *Exemples* : table dynamique

4 Amélioration de l'étude de la complexité

Plusieurs pistes existe afin d'améliorer la complexité d'un algorithme : utiliser une structure de données plus adaptée, utiliser un peu plus de mémoire ou au contraire se souvenir de moins de choses, ... Cependant, quelques fois nous sommes capable de mettre une borne minimale sur la complexité d'une famille de problème : quand nous avons atteint cette borne on sait que nos algorithmes sont optimaux.

4.1 Borne minimale de la complexité sur une classe de problème

— *Proposition* : Borne minimal d'un algorithme de tri
— *Exemple* : Tri par insertion $O(n^2) \geq O(n \log n)$; Tri fusion $O(n \log n)$ atteint cette borne
— *Remarque* : même si on atteint la borne optimale asymptotique, on peut vouloir optimiser les constantes (tri rapide est en moyenne plus rapide que le tri fusion).

4.2 Utilisation de structures de données adaptées

Une piste pour améliorer un algorithme : utiliser une bonne structure de données
— *Exemple* : tri par tas + *Remarque* : dépend de la manière dont on implémente la structure
— *Exemple* : représentation d'un graphe + *Remarque* : utilisation de ces représentation
— *Exemple* : Prim (liste d'adjacence) + Floyd Warshall (matrice d'adjacence)
— On peut changer de structure de données
— *Exemple* : Dijstra DEV

4.3 Compromis espace/temps

— *Principe* : On peut vouloir améliorer le temps au détriment de l'espace ou vise-versa.
— *exemple* : Fibonacci
  — récursif : $O(n^2)$ en temps et $O(1)$ en espace (amélioration de l'espace)
  — programmation dynamique : $O(n)$ pour le temps et l'espace (amélioration du temps)
  — utilisation de deux variables : $O(n)$ pour le temps et $O(1)$ pour l'espace (gagnant sur les deux tableaux)
— *Principe* : le mémoïsation [3, p.338] + *Exemple* : découpage de barre

Ouverture

Évaluer la complexité d'un algorithme (hors implémentation) n'est pas une tâche facile. Souvent plusieurs astuces sont nécessaire pour trouver la meilleure borne sur notre complexité possible. Quelque fois, un approximation grossière de notre complexité (évaluation de la complexité pour le tri par tas) suffit.

Quelques notions importantes

Notation de Landau

On ne peut pas toujours calculer la complexité exacte (avec les constantes) car généralement, elle dépend de l'implémentation que nous utilisons. Pour cette même raison le calcul de la complexité exacte peut s'avérer inutile. On quantifie alors la complexité asymptotiquement. On utilise pour cela la notation de Landau [3, p.40].

Définition. Soit $f, g : \mathbb{N} \to \mathbb{R}_+$. On note

$$
\begin{array}{lcll}
O(g) & = & \{f \mid \exists n_0, c \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\} & \text{Majorant asymptotique} \\
\Theta(g) & = & \{f \mid \exists n_0, c_1, c_2 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} & \text{Encadrement asymptotique} \\
\Omega(g) & = & \{f \mid \exists n_0, c \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\} & \text{Minorant asymptotique}
\end{array}
$$

Remarque (Abus de notation). On écrira $f = O(g)$ (respectivement $f = \Theta(g)$) pour $f \in O(g)$ (respectivement $f \in \Theta(g)$).

Par définition, on a $f = O(g)$ si et seulement si $g = \Omega(f)$.

Proposition. *Soient $f, g : \mathbb{N} \to \mathbb{R}_+$, on a*

$$\begin{aligned} [f = O(g) \text{ et } g = O(f)] &\Leftrightarrow f = \Theta(g) \\ g = \Theta(f) &\Leftrightarrow f = \Theta(g) \end{aligned}$$

Par exemple, on a $n^2 + n = O(n^3)$. De plus, $n^2 + n = O(n^2)$ et $n^2 = O(n^2 + n)$. Par la proposition précédente, on a $n^2 + n = \Theta(n^2)$, mais $n^2 + n \neq \Theta(n^3)$.

Remarque. Ces notations sont transitives et réflexives.

Proposition. *L'ordre de croissance des O est : $O(1)$; $O(\log n)$; $O(n)$; $O(n \log n)$; $O(n^2)$; $O(n^k)$; $O(k^n)$; $O(n!)$. Cet échelle de croissance est utile pour comparer l'efficacité de deux algorithmes corrects résolvant le même problème.*

Quelques analyses d'algorithmes

Nous donnons ici une liste d'exemples d'algorithmes pour lesquels nous calculons leur complexité en mettant en place les différentes méthodes étudier dans la leçon sur la complexité. Ce sont de bonnes illustrations du calcul de complexité.

L'algorithme Cocke–Younger–Kasami L'algorithme CYK (Cocke–Younger–Kasami) [2, p.198] est un algorithme qui décide en temps cubique si un mot est engendré par une grammaire en forme normale quadratique. Il donne alors un certificat que tout langage algébrique est dans la classe P : comme toute grammaire algébrique est équivalente à une grammaire en forme normale quadratique, tout langage algébrique peut être décidé en temps cubique. Il faut faire attention au fait que la grammaire est fixée et qu'elle ne fait pas partie de l'entrée car la mise en forme normale d'une grammaire algébrique peut être exponentielle en sa taille.

Définition. Le problème du mot pour une grammaire algébrique :
entrée : une grammaire algébrique $G = (\Sigma, T, R)$, $w \in \Sigma^*$ un mot
sortie : oui si $w \in L_G$ le langage engendré par G ; non sinon

Remarque : En général répondre à ce problème est coûteux $O(|w|^3|G|^{|w|})$.

Définition. Soit G une grammaire algébrique. On dit que :

- G est sous forme normale si les règles sont sous la forme $A \to BCD \dots Z$ où $A \in T$ (alphabet de travail) et $B, \dots, Z \in \Sigma \cup T$.
- G est sous forme normale de Chomsky si les règles sont sous la forme : $A \to \alpha$ pour, $A \in V, \alpha \in \Sigma$ $V \cap \Sigma = \emptyset$; $S \to \epsilon$ et $A \to BC$ pour $A \in V$ et $B, C \in V \setminus \{S\}$

Théorème. *Le problème du mot pour une grammaire algébrique sous forme forme normale de Chomsky est décidable et un algorithme de programmation dynamique le décide en temps $O(|w|^3|G|)$ et en mémoire $O(|w|^3)$.*

Démonstration. Cet algorithme est un algorithme de programmation dynamique. Soit $w = a_1 \dots a_n$ un mot.

On note :

— $1 \leq i \leq j \leq n$, $w[i,j] = a_i \dots a_j$

— $1 \leq i \leq j \leq n$, $E_{i,j} = \{S \text{ des variables telles que } w[i,j] \in L_G(S)\}$. L'algorithme calcul tous ces $E_{i,j}$.

Nous allons maintenant énoncer quelques propriétés d'appartenance à ces ensembles $E_{i,j}$:

— $w \in L_G(S_0)$ si $S_0 \in E_{1,n}$

— S appartient à $E_{i,i}$ si et seulement si $S \to a_i$ est une règle de G ($w[i,i] = a_i$ et on a une grammaire en forme normale quadratique)

— S appartient à $E_{i,j}$ ($i < j$) si et seulement s'il existe une règle $S \to S_1 S_2$ et $k \in \mathbb{N}$ tels que $w[i,k] \in L_G(S_1)$ et $w[k+1,j] \in L_G(S_2)$.

Les ensembles $E_{i,j}$ peuvent être calculés à partir des ensembles $E_{i,k}$ et $E_{k+1,j}$ pour $i \leq k < j$. L'algorithme les calcul par récurrence sur la différence $j - i$.

Algorithm 1 Algorithme CYk (Cocke–Younger–Kasami).

```
1:  function CYK(w)                          ▷ w = a_1 ... a_n : mot
2:      for 1 ≤ i ≤ j ≤ n do                 ▷ Initialisation ;
        Complexité : O(|w|^2) (double boucle)
3:          E_{i,j} ← ∅
4:      end for
5:      for i = 1 à n do                     ▷ Cas i = j ; Complexité :
        O(|w||G|) (double boucle : une sur w, une sur G)
6:          for tout règle S → a do
7:              if a_i = a then
8:                  E_{i,i} ← E_{i,i} ∪ {a}
9:              end if
10:         end for
11:     end for
12:     for d = 1 à n do                     ▷ Cas i < j ; d = j − i ;
        Complexité : O(|w|^3|G|) (triple boucle sur w ; une sur G)
13:         for i = 1 à n − d do
14:             for k = i à i + d do
15:                 for tout règle S → S_1 S_2 do
16:                     if S_1 ∈ E_{i,k} et S_2 ∈ E_{k+1,i+d} then
17:                         E_{i,i+d} ← E_{i,i+d} ∪ {S}
18:                     end if
19:                 end for
20:             end for
21:         end for
22:     end for
23: end function
```

Complexité : Comme la taille de la grammaire est fixé et ne fait pas partie de l'entrée (c'est une constante), on a bien la complexité annoncé en $O(|w|^3)$ □

L'algorithme de la marche de Jarvis La marche de Jarvis [1, p.389] (ou algorithme du papier cadeau) permet de calculer l'enveloppe convexe d'un ensemble de points. Le but de cet algorithme n'est pas de déterminer les sommets de l'enveloppe convexe mais ces arêtes. Or $[p,q]$ est une arête de l'enveloppe si et seulement si tous les points de l'ensemble sont du même côté de la droite. On peut alors décider en temps $O(n)$ si un segment est une arête de l'enveloppe. Comme on a $O(n^2)$ segment à examiner, on obtient une complexité en $O(n^3)$ pour cet algorithme naïf.

La marche de Jarvis améliore cet algorithme en remarquant que le point suivant p_{k+1} d'une enveloppe convexe est le point minimal dans l'ensemble pour l'ordre polaire relativement à la droite $[p_k, \overrightarrow{p_{k-1}p_k})$. On obtient alors l'algorithme suivant :

Complexité : On a donc une complexité en $O(hn)$ qui est intéressante si h est petit devant n mais en $O(n^2)$ dans le pire cas.

L'algorithme du tri rapide randomisé Le tri rapide (Algorithme 3) est un tri en place dont la complexité moyenne est en $O(n \log n)$ (elle est optimale) et dont la complexité au pire cas est en $O(n^2)$. Il applique le paradigme diviser pour régner en séparant l'entrée en deux sous tableau

Algorithm 2 Algorithme calculant une enveloppe convexe par la marche de Jarvis.

function JARVIS(S) ▷ S : ensemble des points
  p_1 le point d'ordonnée minimale de S (et d'abscisse si plusieurs) ▷ Complexité : $O(n)$
  p_2 le point minimal de $S \setminus \{p_1\}$ pour l'ordre polaire relativement à la droite $[p_i, \overrightarrow{i})$ ▷ Complexité : $O(n)$
  $k \leftarrow 2$
  while $p_k \neq p_1$ **do** ▷ Complexité : $O(h)$ où h est le nombre de points dans l'enveloppe connexe
    Calculer q le point minimal de S pour l'ordre polaire relativement à la droite $[p_k, \overrightarrow{p_{k-1}p_k})$ ▷ Complexité : $O(n)$
    $k \leftarrow k + 1$
    $p_k \leftarrow q$
  end while
end function

dont les valeurs sont respectivement inférieures (ou supérieures) à un pivot. L'algorithme de tri se rappelle alors sur ces deux sous tableau.

Algorithm 3 Algorithme du tri rapide.

function TRI-RAPIDE(A) ▷ A : tableau à trier
  if $A.taille \geq 2$ **then**
    $pivot \leftarrow A[0]$
    $i \leftarrow 0$
    $j \leftarrow A.taille - 1$
    while $i < j$ **do**
      if $A[i+1] \geq pivot$ **then**
        Échanger $A[i+1]$ et $A[j]$
        $j \leftarrow j - 1$
      else
        Échanger $A[i+1]$ et $A[i]$
        $i \leftarrow i + 1$
      end if
    end while
    Tri-Rapide($A[0 \dots i-1]$)
    Tri-Rapide($A[i+1 \dots A.taille - 1]$)
  end if
end function

Algorithm 4 Algorithme du tri rapide randomise.

function TRI-RAPIDE(A) ▷ A : tableau à trier
  if $A.taille \geq 2$ **then**
    $pivor \leftarrow$ Random(A) ▷ On prend un élément au hasard dans A
    $i \leftarrow 0$
    $j \leftarrow A.taille - 1$
    while $i < j$ **do**
      if $A[i+1] \geq pivot$ **then**
        Échanger $A[i+1]$ et $A[j]$
        $j \leftarrow j - 1$
      else
        Échanger $A[i+1]$ et $A[i]$
        $i \leftarrow i + 1$
      end if
    end while
    Tri-Rapide($A[0 \dots i-1]$)
    Tri-Rapide($A[i+1 \dots A.taille - 1]$)
  end if
end function

Théorème (Correction)**.** *L'algorithme du tri rapide (Algorithme 3) est correct.*

Démonstration. content... □

Analyse de la complexité du tri rapide

— Dans le pire cas :

— Dans le cas favorable

Remarque. L'équilibre du découpage du tableau en deux sous tableau se répercute dans la complexité d'exécution.

Tri rapide randomisé Pour étudier la complexité moyenne de ce tri, nous allons utiliser une version randomisé qui simplifiera notre étude (Algorithme 4). On remarque que la correction du nouvel algorithme ne change pas car elle ne dépend pas du pivot choisi mais uniquement des actions autour de celui-ci.

L'algorithme du balayage de Graham Le balayage de Graham résout le problème de l'enveloppe connexe en conservant une pile de candidat aux sommets de cette dite enveloppe [3, p.948]. Chacun des points est empiler une fois, puis tous ceux qui ne sont pas un sommet de l'enveloppe seront dépilé. L'algorithme utilise deux sous-fonctions sommet qui retourne le haut de la pile sans le modifier et sous-sommet qui retourne le deuxième élément de la pile sans la modifier.

Algorithm 5 Algorithme calculant une enveloppe convexe par le balayage de Graham.

1: **function** GRAHAM(S) ▷ S : ensemble des points
2: p_0 le point d'ordonnée minimale de S (et d'abscisse si plusieurs)
3: $p_1, \dots, p_m$ les points de S triés par angle polaire relativement à p_0 (si égalité, on garde le plus loin)
4: $P \leftarrow \emptyset$ ▷ P est une pile
5: Empiler(p_0, P)
6: Empiler(p_1, P)
7: Empiler(p_2, P)
8: **for** $i = 3$ à m **do**
9: **while** l'angle formé les les points sous-sommet(P), sommet(P) et p_i fond un tour à gauche **do**
10: Dépiler(P)
11: **end while**
12: Empiler(p_i, P)
13: **end for**
14: **end function**

Théorème (Correction)**.** *Si Graham est exécuter sur un ensemble S de points tel que $|S| \geq 3$, alors, à la fin de la procédure la pile P contient les du bas vers le haut les sommets de l'enveloppe convexe pris dans l'ordre inverse des aiguilles d'une montre.*

Arguments de la preuve.
— Les points qui sont retire lors du tri ne sont pas dans l'enveloppe convexe.
— Invariant : "A chaque itération de la boucle Pour, la pile P devient de bas en haut, les sommets de l'enveloppe convexe Q_{i-1} pris dans l'ordre inverse des aiguilles d'une montre." où Q_i est l'ensemble des i premiers éléments donné par le tri.
Initialisation La pile contient Q_2 (trois points) qui sont bien leur enveloppe connexe pris dans l'ordre inverse des aiguilles d'une montre.
Conservation Après la boucle tant que on fait apparaître l'invariant de l'itération précédente (la pile contient la même chose). On montre ensuite que l'enveloppe convexe de $Q_j \cup \{p_i\}$ est la même que celle de Q_i soit p_i en fait partie, soit il est à l'intérieur.
Terminaison Ok

□

Théorème (Complexité). *Le temps d'exécution de* Graham *est* $O(n \log n)$ *si* $n = |S|$.

Arguments de la preuve. — Choisir le point minimal $\Theta(n)$

— Trier les autres points $O(n \log n)$ (si on utilise un algorithme de tri optimal (tri fusion ou tri par tas) et la suppression des points d'angles polaires égaux se fait en $O(n)$)

— La boucle pour nécessite un temps en $O(n)$ sans compter la boucle tant que.

— La boucle tant que nécessite au total (pour n boucle pour) un temps en $O(n)$. On utilise la méthode de l'agrégat.

  — On empile au plus un fois chaque points de S

  — On dépile au plus autant que l'on empile : au plus $m - 2$ opérations de dépiler.

  — Le test et l'opération de dépiler en $O(1)$.

□

L'algorithme de Knuth–Morris–Pratt Les algorithmes Morris–Pratt et Knuth–Morris–Pratt sont des algorithmes qui utilisent la notion de bord. On utilise le bord des préfixes du motif pour être plus astucieux et rapide quand une comparaison échoue : il nous donne le décalage que l'on doit réaliser.

Hypothèse : Le motif est fixe et connu à l'avance.

Quelques notions sur le bord [1, p.340] Le bord a un rôle essentiel dans ces algorithmes : il permet de calculer le décalage que l'on va effectuer lors d'un échec de comparaison. Bien définir cette notion est donc primordiale.

Définition. Un bord de x est un mot distinct de x qui est à la fois préfixe et suffixe de x. On note $Bord(x)$ le bord maximal d'un mot non vide.

Exemple Le mot *ababa* possède les trois bords ϵ, a et aba. De plus $Bord(ababa) = aba$.

Proposition. *Soit* x *un mot non vide et* $k \in \mathbb{N}$*, le plus petit, tel que* $Bord^k(x) = \epsilon$.

1. *Les bords de* x *sont les mots* $Bord^i(x)$ *pour tout* $i \in \{1, \dots, k\}$.
2. *Soit* a *une lettre. Alors,* $Bord(xa)$ *est le plus long préfixe de* x *qui est dans l'ensemble* $\{Bord(x)a, \dots Bord(x)^k a, \epsilon\}$.

Arguments de la preuve. 1. z est un bord de $Bord(x)$ ssi z est un bord de x (+ récurrence).

2. z est un bord de za ssi $z = \epsilon$ ou $z = z'a$ avec z' un bord de x.

□

Corollaire. *Soit* x *un mot non vide et soit* a *une lettre. Alors,*

$$Bord(xa) = \begin{cases} Bord(x)a & \text{si } Bord(x)a \text{ est préfixe de } x \\ Bord(Bord(x)a) & \text{sinon} \end{cases}$$

Arguments de la preuve. — Si $Bord(x)a$ est préfixe de x alors $Bord(x)a = Bord(xa)$.

— Sinon, $Bord(xa)$ est préfixe de $Bord(x) = y$ et le plus long préfixe de x dans $\{Bord(y)a, \dots Bord(y)^k a, \epsilon\}$.

□

Définition. Soit x un mot de longueur m. On pose $\beta : \{0, \dots, m\} \to \{-1, \dots, m-1\}$ la fonction qui est définie comme suit : $\beta(0) = -1$ et pour tout $i > 0$, $\beta(i) = |Bord(x_1, \dots, x_i)|$. On pose $s : \{1, \dots, m-1\} \to \{0, \dots, m\}$ la fonction suppléance de x définie comme suit : $s(i) = 1 + \beta(i-1), \forall i \in \{1, \dots, m\}$.

Remarque : on a bien entendu $\beta(i) \leq i - 1$.

Corollaire. *oit x un mot non vide de longueur m. Pour $j \in \{0, \ldots, m-1\}$, on a $\beta(1+j) = 1 + \beta^k(j)$ où $k \geq 1$ est le plus petit entier vérifiant l'une des deux conditions suivantes :*

1. $1 + \beta^k(j) = 0$
2. $1 + \beta^k(j) \neq 0$ *et* $x_{1+\beta^k(j)} = x_{j+1}$

Arguments de la preuve. Algorithme 6 □

Algorithm 6 Calcul de la fonction β donnant la taille des bords maximaux d'un mot x.

```
1:  function BORD-MAXIMAUX(x, β)        ▷ x mot de taille m
2:      β[0] ← −1
3:      for j = 1 à m do
4:          i ← β[j − 1]
5:          while i ≥ 0 et x[j] ≠ x[i + 1] do
6:              i ← β[i]
7:          end while
8:          β[j] ← i + 1
9:      end for
10: end function
```

Algorithm 7 Calcul de la fonction suppléance s du mot x.

```
1:  function SUPPLÉANCE(x, s)           ▷ x mot de taille m
2:      s[1] ← 0
3:      for j = 1 à m − 1 do
4:          i ← s[j]
5:          while i ≥ 0 et x[j] ≠ x[i] do
6:              i ← s[i]
7:          end while
8:          s[j + 1] ← i + 1
9:      end for
10: end function
```

L'algorithme de Morris–Pratt [3, p.916] L'algorithme de Morris–Pratt utilise un automate des occurrences pour calculer les bords du motif. On définit l'automate des occurrences associé à $P[1 \ldots m]$ comme suit : $\mathcal{Q} = \{1, \ldots m\}$; $q_0 = 0$; $F = \{m\}$; $\delta(q, a) = \sigma(P_q a)$ pour tout $a \in \Sigma$ et $q \in \mathcal{Q}$.

Définition. La fonction suffixe associé au motif P $\sigma : \Sigma^* \mapsto \{0, 1, \ldots, m\}$ donne la longueur de $Bord(x)$ pour $x \in \Sigma^*$.

Algorithm 8 Calcul de la fonction de transition δ de l'automate des occurrences.

```
1:  function CALCUL-δ(P, Σ)             ▷ P motif ; Σ alphabet
2:      p ← P.longueur
3:      for q = 0 à p do
4:          for a ∈ Σ do
5:              k ← min(p + 1, q + 2)
6:              repeat
7:                  k ← k + 1
8:              until P_k préfixe de P_q a      ▷ P_q est le préfixe de P de longueur q.
9:              δ(q, a) ← k
10:         end for
11:     end for
12:     Retourner δ
13: end function
```

Complexité temporelle du prétraitement dans le pire cas : $O(p|\Sigma|)$ Ce fait une fois par motif.

Algorithm 9 Calcul de la recherche dans l'automate des occurrences.

```
1:  function MORRIS–PRATT(T, δ, p)      ▷ T texte ; δ fonction transition ; p la longueur du motif
2:      t ← T.longueur
3:      for i = 1 à t do
4:          q ← δ(q, T[i])
5:          if q = p then
6:              Retourner i − m
7:          end if
8:      end for
9:      Retourner 0
10: end function
```

Complexité temporelle dans le pire cas : $\Theta(t)$

Remarque : La validité de cet algorithme provient de la validité de la construction de l'automate des occurrences.

L'algorithme de Knuth–Morris–Pratt [1, p.343] Cet algorithme utilise une méthode plus astucieuse afin de calculer les bords du préfixes. On s'épargne ainsi un calcul de l'automate des occurrences et on calcul la fonction δ puisse à la volée. On exploite les propriétés de la fonction de bord.

Définition. La fonction préfixe associé au motif P $\pi : \{1, 2, \ldots, p\} \mapsto \{0, 1, \ldots, p-1\}$ donne la longueur du plus long préfixe de P qui est suffixe propre de P_q où $q \in \{1, 2, \ldots, p\}$.

Si on ne fait pas l'hypothèse d'un préfixe propre alors le plus long suffixe d'un mot qui est aussi son préfixe est le mot en question. De plus, sans cette hypothèse l'algorithme de Knuth-Morris-Pratt serait soit incorrect soit non terminal.

Algorithm 10 Algorithme de Knuth–Morris–Pratt.

```
1:  function KMP(T, P)  ▷ T texte ; P motif
2:      t ← T.longueur
3:      p ← P.longueur
4:      i ← 1
5:      j ← 1
6:      s ← SUPPLÉANCE(x, P)
7:      while i ≤ p et j ≤ t do
8:          if i ≥ 1 et t[j] ≠ x[i] then
9:              i ← s[i]
10:         else
11:             i ← i + 1
12:             j ← j + 1
13:         end if
14:     end while
15:     if then i > m
16:         Renvoie j − m
17:     else
18:         Renvoie 0
19:     end if
20: end function
```

Remarque : La validité de l'algorithme de Knuth–Morris–Pratt (Algorithme 10) provient des propriétés sur le bord.
Complexité du prétraitement Calcul de la fonction suppléance s : $\Theta(p)$ On applique une méthode de l'agrégat : dans la boucle pour la variable ne peut augmenter plus de p fois et comme elle est toujours strictement positive, la boucle tant que ne peut pas s'exécuter plus de p fois.
Complexité : $\Theta(t)$ On applique une méthode de l'agrégat (exactement le même raisonnement).
Remarque : Il existe encore une version améliorer de cet algorithme qui consiste à ne pas vouloir se retrouver dans la même situation qu'au début (on teste toujours une situation différente). Cette amélioration donne les même complexité asymptotique mais semblerait plus rapide en pratique.

L'exemple d'une table dynamique Lorsqu'on manipule un tableau [3, p.428], on ne connaît pas toujours la place dont on va avoir besoin. On veut alors pouvoir ré-allouer une table plus grande dans le cas où on ajoute une nouvelle valeur et que notre table est déjà pleine. Nous allons montre que la complexité amortie de l'opération insérer est en $O(1)$ lorsque insérer consiste à ajouter l'élément dans la table si elle est non pleine ou de créer une table de taille double, de recopier les valeurs de la première dans la nouvelle et d'ajouter la valeur dans cette table.

Analysons la complexité d'une séquence de n opérations insérer sur une table vide. On note c_i le coût de la $i^{\text{ième}}$ opération. Si la table n'est pas pleine, on a $c_i = 1$ sinon $c_i = i$. Si on effectue n opérations insérer, le coût défavorable d'une opération est $O(n)$, ce qui donne une complexité en $O(n^2)$ pour la séquence. Cette borne n'est pas optimale, on va calculer cette complexité amortie, à l'aide des trois méthodes exposées dans la leçon.

Calcul par la méthode de l'agrégat On se donne une séquence de n opérations insérer sur une table vide. On souhaite affiner la borne précédente. On remarque que la $i^{\text{ième}}$ opération déclenche une extension si et seulement si $i-1$ est une puissance de deux. Le coût de la $i^{\text{ième}}$ opération est alors :

$$c_i = \begin{cases} i & \text{si } i-1 \text{ est une puissance de 2} \\ 1 & \text{sinon} \end{cases}$$

Le coût total de n opération insérer sur une table vide est :

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=1}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$$

Le coût amorti d'une opération insérer est au plus trois.

Calcul par la méthode comptable La méthode comptable permet de donner l'intuition de la valeur trois. En effet, lorsqu'on insère un élément, on lui donne un crédit correspondant à son insertion, son mouvement et la recopie d'un élément présent dans le tableau avant lui (comme on double la taille du tableau le nombre d'éléments qui ne viennent pas d'être ajouté à déplacer lorsque le tableau est plein est la moitié. Donc les ajouts financent bien la recopie des autres éléments). Plus formellement, on fixe :

$$\begin{array}{ccc} T \text{ est non plein} & cred(op) = 2 & dep(op) = 0 \\ T \text{ est plein} & cred(op) = 2 & dep(op) = |T| \end{array}$$

Ce qui nous donne

$$c_{amo}(op) = \begin{cases} 1+2-0 \\ (|T|+1)+2-|T| \end{cases} = 3$$

D'où la complexité amortie de trois.

Calcul par la méthode du potentiel On va maintenant appliquer la méthode du potentiel pour le calcul de cette complexité amortie. On souhaite une fonction potentielle qui vaut 0 lorsque la table vient d'être étendu et qui vaut la taille de la table lorsque celle-ci est pleine. La fonction $\varphi(T) = 2T.num - T.taille$ où $T.num$ est le nombre d'élément dans la table et $T.taille$ est sa taille, est une possibilité. On considère la $i^{\text{ième}}$ opération. On distingue deux cas.

— Si elle ne déclenche pas d'extension de la table, alors $taille_i = taille_{i-1}$ et

$$\begin{aligned} c_{amo}(i) &= c_i + \varphi_i - \varphi_{i-1} = 1 + (2num_i - taille_i) - (2num_{i-1} - taille_{i-1}) \\ &= 1 + (2num_i - taille_i) - (2(num_i - 1) - taille_i) = 3 \end{aligned}$$

— Si elle déclenche l'extension de la table, alors $taille_i = 2taille_{i-1}$, $taille_{i-1} = 2(num_i) - 1$ et

$$\begin{aligned} c_{amo}(i) &= c_i + \varphi_i - \varphi_{i-1} = 1 + (2num_i - taille_i) - (2num_{i-1} - taille_{i-1}) \\ &= 1 + (2num_i - taille_i) - (2(num_i - 1) - (num_i - 1)) = 3 \end{aligned}$$

Analyse de relation par récurrence

Nous allons maintenant étudier comment résoudre des récurrences [4, p.536]. Nous ne sommes pas obligé de tous détailler dans la leçon, mais en connaître quelques un peut être une bonne chose.

Les types d'équations récurrentes On peut classifier les suites récurrentes en plusieurs grandes familles :

— Les récurrences linéaires d'ordre k sont des équations du type : $T(n) = f(n, T(n-1), \ldots, T(n-k)) + g(n)$ où $k \in \mathbb{N}$ est fixé, f est une combinaison linéaire de ces éléments et g est une fonction quelconque. (Exemple : factorielle)

— Les récurrences de partitions sont des équations du type : $T(n) = aT\left(\frac{n}{b}\right) + d(n)$ où $a, b \in \mathbb{N}$ sont des constantes, d est une fonction quelconque et n est une puissance de d. (Elles apparaissent dans des algorithmes de partitions. Master theorem)

— Les récurrences complètes, linéaires ou polynomiales sont des équations du type : $T(n) = f(n, T(n-1), \ldots, T(0)) + g(n)$ où $k \in \mathbb{N}$ est fixé, f est une fonction linéaire ou polynomiale et g est une fonction quelconque.

Il existe de nombreuses méthodes (de la méthode du calcul à la main aux séries génératrices) pour résoudre ces équations. Nous allons donner l'exemple des récurrences linéaires d'ordre 1 et du master theorem.

Équations récurrentes linéaires d'ordre 1 On commence par les plus simple qui peuvent généralement être résolue à la main. Nous allons donner un résultat sur ces équations.

Proposition. *Pour une suite récurrente linéaire d'ordre 1* $(u_n)_{n\in\mathbb{N}}$ *avec* $u_{n+1} = au_n + b$ *pour tout* $n \geq 0$ *où* $u_0 \in \mathbb{R}$ *et* $a, b \in \mathbb{R}$. *Alors* $u_n = a^n(u_0 - l) + l$ *où* $l = \frac{b}{1-a}$ *si* $a \neq 1$ *et* $u_n = bn + u_0$ *sinon.*

Pour l'algorithme calculant la factorielle, si on compte le nombre de multiplication, on a une complexité en $c(n) = c(n-1) + 1$ avec $a = b = 1$ et $c(0) = 0$. Donc $c(n) = 1 * n + 0 = n$.

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] O. Carton. *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[4] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.