Matthieu Boyer
Planche A

Notation polonaise inverse et évaluation d'expressions

La notation polonaise inverse (NPI) représente les expressions arithmétiques sans parenthèses : $(3+4)*(2-1)$ devient $3\ 4\ +\ 2\ 1\ -\ *$. On travaille avec des entiers et les opérateurs $+$, $-$, $*$, $/$.

1. Rappeler ce qu'est une pile.
2. Décrire un algorithme évaluant une expression en NPI à l'aide d'une pile. Prouver sa correction et analyser sa complexité.
3. Décrire un algorithme qui traduit une expression infixe (notation habituelle) en NPI à l'aide d'une pile d'opérateurs, en traitant correctement les priorités ($*$ et $/$ avant $+$ et $-$, associativité à gauche) et les parenthèses.
4. Montrer que toute expression bien formée admet un arbre d'expression dont les feuilles sont les opérandes et les nuds internes les opérateurs. Que peut-on en déduire sur les algorithmes précédents ?
5. On remplace les opérateurs arithmétiques par les connecteurs logiques binaires $\wedge$, $\vee$ et l'opérateur unaire $\neg$. Identifier les difficultés spécifiques à $\neg$ dans chacun des deux algorithmes et décrire les adaptations nécessaires.

Solution : Notation polonaise inverse et évaluation d'expressions

1. Algorithme d'évaluation NPI : On lit l'expression de gauche à droite. Si le symbole lu est un opérande (nombre), on l'empile. Si c'est un opérateur binaire, on dépile les deux premiers éléments (le premier dépilé est l'opérande droit, le second l'opérande gauche), on applique l'opérateur, puis on empile le résultat. À la fin, la pile contient un unique élément : le résultat.

   Correction : Se prouve par récurrence structurelle. Une expression NPI valide est soit un entier, soit la concaténation de deux expressions NPI valides suivie d'un opérateur. L'algorithme évalue récursivement les sous-expressions et les remplace par leur valeur sur la pile.

   Complexité : $\mathcal{O}(n)$ en temps (chaque symbole est empilé et dépilé au plus une fois) et $\mathcal{O}(n)$ en espace (la taille de la pile dans le pire cas).

2. Algorithme de Shunting-Yard (Dijkstra) : On utilise une file pour la sortie (la NPI) et une pile pour les opérateurs en attente. En lisant de gauche à droite :
   - Un nombre va directement dans la file de sortie.
   - Une parenthèse ouvrante est empilée.
   - Une parenthèse fermante dépile les opérateurs vers la sortie jusqu'à trouver l'ouvrante correspondante (qu'on dépile et jette).
   - Un opérateur *op* dépile vers la sortie tous les opérateurs au sommet de la pile ayant une priorité strictement supérieure, ou une priorité égale et associatifs à gauche (comme $+, -, *, /$). Puis on empile *op*.

   À la fin de la lecture, on dépile tout le reste vers la sortie.

3. Arbre d'expression : L'arbre est construit en plaçant l'opération principale à la racine, et en décomposant récursivement (feuilles = nombres, nuds = opérateurs).

   Déduction : L'évaluation de l'expression NPI (Q2) correspond exactement à un parcours postfixe (ou suffixe) de cet arbre. L'algorithme de Shunting-Yard correspond à la traduction d'un parcours infixe (la notation habituelle, qui nécessite des parenthèses) en parcours postfixe. La composition des deux traduit donc l'arbre vers sa forme évaluable linéairement.

4. Spécificités du $\neg$ (unaire) :
   - Pour l'évaluation : Au lieu de dépiler deux opérandes, le $\neg$ ne doit en dépiler qu'un seul. L'algorithme doit vérifier l'arité de l'opérateur pour adapter le nombre de `pop()`.
   - Pour Shunting-Yard : Le $\neg$ a la priorité maximale et s'associe souvent à droite. Lors de son arrivée, il ne dépile pas d'autres opérateurs (ou seulement d'autres unaires) et s'empile directement.

Matthieu Boyer
Planche B

Structure et robustesse d'un réseau

Un réseau de machines est modélisé par un graphe non orienté $G = (V, E)$ représenté par liste d'adjacence. Un virus infecte simultanément un sous-ensemble $S_0 \subseteq V$ à $t = 0$ et se propage à chaque instant à tous les voisins non encore infectés des machines contaminées au pas précédent.

1. Concevoir un algorithme calculant pour chaque machine son temps d'infection ou $+\infty$ si elle est inatteignable. Justifier la structure de données utilisée, analyser la complexité, et préciser en quoi le cas multi-sources diffère structurellement du cas à source unique.
2. Un sommet v est un point d'articulation si $G \setminus \{v\}$ possède strictement plus de composantes connexes que G. Proposer un algorithme les identifiant tous en $\mathcal{O}(|V| + |E|)$.
3. Pour isoler une machine cible t de la source s, on veut supprimer une unique arête. Caractériser précisément les arêtes dont la suppression atteint cet objectif, proposer un algorithme les identifiant toutes, et analyser sa complexité.

Si aucune arête unique ne permet d'isoler t de s, on cherche à retirer un ensemble d'arêtes $C \subseteq E$ pour y parvenir. On dit que C sépare s de t si s et t se retrouvent dans deux composantes connexes différentes après la suppression de C.

Une façon naturelle de construire un tel ensemble est de partitionner les sommets de G en deux ensembles distincts A et B tels que $s \in A$ et $t \in B$. L'ensemble des arêtes ayant une extrémité dans A et l'autre dans B s'appelle la **coupe** associée à la partition (A, B).

4. Montrer que toute coupe associée à une telle partition déconnecte bien s et t. Montrer que la taille minimale d'une telle coupe (en nombre d'arêtes) est égale au nombre maximal de chemins disjoints (ne possédant aucune arête en commun) reliant s à t.
5. En déduire un algorithme calculant une coupe minimale.

Solution : Structure et robustesse d'un réseau

1. Algorithme (BFS multi-sources) : On utilise un Parcours en Largeur (BFS). On initialise un tableau `temps` à $+\infty$ pour tous les sommets. Pour chaque $v \in S_0$, on fixe `temps`$[v] = 0$ et on l'ajoute à une file (FIFO). Tant que la file n'est pas vide, on défile u. Pour chaque voisin w de u, si `temps`$[w] == +\infty$, on fixe `temps`$[w] =$ `temps`$[u] + 1$ et on enfile w. La file garantit le traitement par niveau de proximité. Complexité : $\mathcal{O}(|V| + |E|)$.
2. Points d'articulation : L'algorithme optimisé repose sur un Parcours en Profondeur (DFS) instrumenté (algorithme de Tarjan/Hopcroft). On maintient un compteur de découverte. Pour chaque sommet v, on calcule :
   - `num[v]` : son ordre de découverte lors du DFS.
   - `low[v]` : le `num` minimum atteignable depuis le sous-arbre de v en utilisant au plus une arête de retour (back-edge).

   Un sommet v (non-racine du DFS) est un point d'articulation s'il possède un enfant u tel que `low`$[u] \geq$ `num`$[v]$ (l'enfant ne peut pas "remonter" plus haut que v sans passer par v). La racine du DFS est un point d'articulation si et seulement si elle a au moins deux enfants dans l'arbre DFS. Complexité : $\mathcal{O}(|V| + |E|)$.
3. Ces arêtes s'appellent des isthmes (ou ponts) et elles doivent de surcroît appartenir à un chemin entre s et t.

   Algorithme : 1. Identifier tous les isthmes du graphe avec un DFS instrumenté (similaire à la question précédente : l'arête (u, v) est un isthme si `low`$[v] >$ `num`$[u]$). 2. Pour chaque isthme identifié, vérifier si sa suppression déconnecte bien s et t (par exemple en retirant l'arête temporairement et en lançant un BFS depuis s pour chercher t).

   Complexité : Trouver les isthmes prend $\mathcal{O}(|V| + |E|)$. La vérification prend $\mathcal{O}(|V| + |E|)$ par isthme testé, soit au pire $\mathcal{O}(|V| \times (|V| + |E|))$.
4. Preuve du Théorème de Menger :
   - Preuve de déconnexion : Soit une partition (A, B) avec $s \in A$ et $t \in B$. La coupe associée C est l'ensemble des arêtes ayant une extrémité dans A et l'autre dans B. Supposons par l'absurde qu'il existe un chemin entre s et t après suppression de C. Ce chemin débute en A et finit en B. Il contient donc nécessairement au moins une transition entre un sommet $u \in A$ et $v \in B$. L'arête (u, v) appartient à C par définition, or toutes les arêtes de C ont été retirées, ce qui est contradictoire.
   - Preuve du Théorème de Menger (par récurrence sur $|E|$) : L'inégalité (max chemins disjoints $\leq$ taille coupe min k) est évidente, car chaque chemin disjoint doit emprunter une arête distincte d'une coupe quelconque. Prouvons que si la coupe minimale vaut k, alors il existe k chemins arête-disjoints.

     Initialisation : Si $|E| = k$, le graphe possède exactement k arêtes reliées directement de s à t (multi-graphe). Elles forment immédiatement k chemins arête-disjoints.

     Hérédité : Supposons la propriété vraie pour tout graphe ayant strictement moins de $|E|$ arêtes.

     Cas 1 : Il existe une coupe minimale C de taille k non triviale (c'est-à-dire que la partition (V_s, V_t) associée vérifie $V_s \neq \{s\}$ et $V_t \neq \{t\}$). Construisons G_s en contractant tous les sommets de V_t en un unique sommet cible t', et G_t en contractant tous les sommets de V_s en un unique sommet source s'. La coupe minimale de G_s (entre s et t') vaut encore k, et comme $V_t \neq \{t\}$, G_s a strictement moins d'arêtes que G. Par hypothèse de récurrence, il y a k chemins arête-disjoints de s à t' dans G_s. Ces chemins se terminent sur les k arêtes de la coupe C. De même, il existe k chemins arête-disjoints de s' à t dans G_t, débutant sur ces k arêtes. En raccordant ces deux familles de chemins au niveau des arêtes de la coupe C, on obtient k chemins arête-disjoints de s à t dans G.

     Cas 2 : Toutes les coupes minimales de taille k sont triviales (incidentes uniquement à s ou à t). S'il existe un chemin $s \to u \to t$ de longueur 2, on peut retirer ses deux arêtes, appliquer l'hypothèse de récurrence sur le graphe réduit (dont la coupe minimale baisse de 1), obtenant $k - 1$ chemins, puis rajouter le chemin initial pour obtenir k chemins. Si aucun chemin n'est de longueur 2, on choisit une arête $e = (u, v)$ non incidente à s ni t. La coupe minimale de $G \setminus \{e\}$ vaut nécessairement k (sinon e appartiendrait à une coupe de taille k, obligatoirement triviale, ce qui contredit le choix de e). Par récurrence, $G \setminus \{e\}$ contient k chemins arête-disjoints, valides dans G.
5. En vertu du théorème, on modélise le problème par un réseau de flot et on applique un algorithme type Ford-Fulkerson.

Matthieu Boyer
Planche C

Arbre binaire de recherche augmenté

On maintient un ABR dans lequel chaque nud stocke, en plus de sa clé, la taille de son sous-arbre.

1. Décrire les algorithmes d'insertion et de suppression en maintenant le champ taille à jour. Analyser la complexité.
2. Exploiter le champ taille pour concevoir un algorithme `kième(racine, k)` renvoyant la k-ième plus petite clé en $\mathcal{O}(h)$. Montrer que sans ce champ, $\mathcal{O}(h)$ est inatteignable dans le pire cas.
3. Montrer que la hauteur d'un ABR peut atteindre $\Theta(n)$ et exhiber la suite d'insertions correspondante.
4. Décrire le principe des arbres AVL (propriété d'équilibre, impact sur la hauteur, surcoût sur les opérations) sans implémenter les rotations.
5. Ajouter une opération `intervalle(a, b)` retournant toutes les clés dans $[a, b]$ en ordre croissant. Concevoir un algorithme et analyser sa complexité en $\mathcal{O}(h + k)$ où k est le nombre de clés retournées.
6. Montrer que $\log n + k$ est une borne inférieure dans le cas équilibré.

Solution : Arbre binaire de recherche augmenté

1. Mise à jour lors de l'insertion et suppression :
   - Insertion : Lors de la descente récursive pour insérer la nouvelle feuille, on incrémente le champ `taille` de 1 pour chaque nud traversé.
   - Suppression : Lors de la suppression (qu'il s'agisse d'une feuille, d'un nud à un enfant, ou du remplacement par le successeur), on décrémente `taille` de 1 sur tout le chemin allant de la racine jusqu'au nud physiquement supprimé.

   Complexité : Ces opérations font un aller-retour sur la branche. La complexité reste $\mathcal{O}(h)$ où h est la hauteur de l'arbre.
2. Algorithme `kième(noeud, k)` : Soit $t =$ `taille(noeud.gauche)` (avec $t = 0$ si pas d'enfant gauche).
   - Si $k = t + 1$: la clé cherchée est celle du nud courant. On la retourne.
   - Si $k \leq t$: la clé est dans le sous-arbre gauche. On retourne `kième(noeud.gauche, k)`.
   - Si $k > t + 1$: la clé est dans le sous-arbre droit. On retourne `kième(noeud.droite, k - (t + 1))`.

   L'algorithme descend d'un étage à chaque appel sans remonter, d'où $\mathcal{O}(h)$. Sans ce champ, on serait forcé de parcourir in-extenso les sous-arbres pour compter les nuds (parcours infixe), ce qui prend $\Theta(n)$ dans le pire cas.
3. Dégénérescence et AVL : La hauteur atteint $\Theta(n)$ (arbre filiforme) si l'on insère des éléments déjà triés, par exemple la suite : 1, 2, 3, 4, 5.

   Les arbres AVL imposent la propriété suivante : pour tout nud, la différence de hauteur entre son sous-arbre gauche et droit vaut au plus 1. Cette contrainte stricte garantit une hauteur $h = \mathcal{O}(\log n)$. Le surcoût réside dans la vérification de l'équilibre à chaque modification et l'exécution de rééquilibrages locaux (rotations simples ou doubles) en temps $\mathcal{O}(1)$ par nud mis à jour, maintenant ainsi la complexité globale des opérations à $\mathcal{O}(\log n)$.
4. Opération `intervalle(noeud, a, b)` :
   - Si `noeud.cle` $> a$, il y a potentiellement des éléments valides à gauche : appel récursif sur l'enfant gauche.
   - Si $a \leq$ `noeud.cle` $\leq b$, on ajoute la clé au résultat.
   - Si `noeud.cle` $< b$, il y a potentiellement des éléments valides à droite : appel récursif sur l'enfant droit.

   Complexité : On descend dans l'arbre pour trouver les bornes a et b (chemins de longueur au plus $\mathcal{O}(h)$), puis on visite exhaustivement les sous-arbres contenus entre ces chemins pour récolter k nuds. Temps total : $\mathcal{O}(h + k)$.
5. Borne inférieure : Dans un arbre parfaitement équilibré, trouver simplement l'élément a (ou son plus proche supérieur) requiert de traverser la hauteur de l'arbre, ce qui donne une borne de $\Omega(\log n)$. Une fois à ce point de départ, il faut obligatoirement retourner ou afficher k éléments, ce qui nécessite $\Omega(k)$ opérations. La borne théorique pour l'opération complète est donc indéniablement $\Omega(\log n + k)$.

Matthieu Boyer
Planche D

Système de types et parsing

Un système de types Ω est défini par un alphabet fini Σ, un ensemble fini $\mathcal{T}$ de types, des règles terminales $\text{type}(a) \subseteq \mathcal{T}$ pour chaque $a \in \Sigma$, des règles de composition $R \subseteq \mathcal{T}^3$ notées $\tau \to \tau_1 \cdot \tau_2$, et un type de départ τ_S.
Un arbre d'analyse pour $s = a_1 \dots a_n$ est un arbre binaire dont les feuilles, lues de gauche à droite, reconstituent s ; chaque feuille a_i porte un type de $\text{type}(a_i)$; chaque nud interne de type τ a ses fils de types τ_1, τ_2 avec $\tau \to \tau_1 \cdot \tau_2 \in R$.
La chaîne s est acceptable s'il existe un tel arbre de racine τ_S. On dit aussi que le système Ω reconnaît s.
On définit un système de référence avec $\Sigma = \{a, b\}$, $\mathcal{T} = \{A, B, S\}$, $\text{type}(a) = \{A\}$, $\text{type}(b) = \{B\}$, $R = \{S \to A \cdot B, S \to A \cdot S, S \to S \cdot B\}$, $\tau_S = S$.

1. Lister tous les arbres d'analyse de `aabb` dans le système de référence. Montrer que ce système reconnaît exactement les chaînes $a^i b^j$ pour $i, j \geq 1$.
2. Considérer maintenant le système réduit $\Sigma' = \{a\}$, $\mathcal{T}' = \{A\}$, $\text{type}(a) = \{A\}$, $R' = \{A \to A \cdot A\}$. Calculer le nombre d'arbres d'analyse de a^n pour $n = 1, \dots, 5$, identifier la suite obtenue et conjecturer une formule close.
3. Montrer que l'algorithme récursif naïf pour décider si $s[i..j]$ est de type τ (essayer toutes les coupures et toutes les paires de types compatibles) est exponentiel en n.
4. Montrer que le nombre de sous-problèmes distincts est $\mathcal{O}(n^2|\mathcal{T}|)$. Montrer que donc on peut résoudre le problème en temps polynomial.
5. Modifier l'algorithme pour reconstruire un arbre d'analyse valide lorsqu'il en existe un, et compter le nombre total d'arbres d'analyse distincts. Calculer ce nombre pour `aabb` et vérifier la cohérence avec la question (a).
6. Montrer que la segmentation de texte, c'est-à-dire décider si s se décompose en une suite de mots d'un dictionnaire fini D, est un cas particulier du présent modèle : expliciter le système de types correspondant.
7. Construire ensuite un système reconnaissant exactement $\{a^n b^n \mid n \geq 1\}$. Montrer qu'aucun dictionnaire fini D ne satisfait
$$\{s \in \{a, b\}^* \mid s \in D^*\} = \{a^n b^n \mid n \geq 1\}$$
On pourra raisonner sur la structure nécessaire de D pour que $ab \in D^*$, puis montrer que cette structure entraîne inévitablement l'appartenance à D^* d'une chaîne $\notin \{a^n b^n\}$.

Solution : Système de types et parsing

1. Analyse de `aabb` et reconnaissance : Pour `aabb`, il existe 2 arbres d'analyse distincts correspondants aux dérivations suivantes :
   — $S \to A \cdot S \to A \cdot (S \cdot B) \to A \cdot ((A \cdot B) \cdot B)$
   — $S \to S \cdot B \to (A \cdot S) \cdot B \to ((A \cdot B) \cdot S)$ (ou structurellement $S \to (A \cdot S) \cdot B \to (A \cdot (A \cdot B)) \cdot B$)

   Le type S exige au moins un A (à gauche) et un B (à droite). Les règles $S \to A \cdot S$ permettent d'accumuler autant de A que souhaité à gauche, et $S \to S \cdot B$ accumule les B à droite. Le système ferme récursivement sur $S \to A \cdot B$. Ainsi, le système engendre exactement le langage a^+b^+, soit a^ib^j avec $i, j \geq 1$.
2. Système réduit $A \to A \cdot A$: Pour $n = 1$, 1 arbre (juste a). $n = 2 \Longrightarrow 1$. $n = 3 \Longrightarrow 2$. $n = 4 \Longrightarrow 5$. $n = 5 \Longrightarrow 14$. C'est la suite des nombres de Catalan. La formule close pour le nombre d'arbres de a^n est le $(n-1)$-ième nombre de Catalan : $C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1}$.
3. Algorithme récursif naïf : L'algorithme essaie toutes les coupures $k \in [i, j-1]$. Pour chaque coupure, il effectue 2 appels récursifs (pour le sous-mot gauche et le sous-mot droit). Le temps d'exécution $T(n)$ vérifie la récurrence $T(n) = \sum_{k=1}^{n-1}(T(k) + T(n-k))$, ce qui donne une complexité exponentielle $\Omega(2^n)$.
4. Complexité avec mémoïsation : Un sous-problème est entièrement défini par les indices (i, j) délimitant le sous-mot, et par le type cible τ. Il y a $\frac{n(n+1)}{2}$ sous-mots possibles, et $|\mathcal{T}|$ types. Il n'y a donc que $\mathcal{O}(n^2 |\mathcal{T}|)$ sous-problèmes distincts. En mémorisant les résultats dans un tableau (programmation dynamique, algorithme CYK), chaque sous-problème s'évalue en essayant $j - i$ coupures, d'où un temps total polynomial en $\mathcal{O}(n^3 |R|)$.
5. Reconstruction et comptage :
   — Reconstruction : Au lieu de stocker un simple booléen dans $\texttt{dp}[i][j][\tau]$, on stocke un triplet gagnant (k, τ_1, τ_2) ayant permis d'obtenir le type τ. On reconstruit l'arbre par retours arrière.
   — Comptage : On remplace l'opérateur booléen OU ($\vee$) par une somme ($+$) et le ET ($\wedge$) par une multiplication ($\times$). $\texttt{ways}[i][j][\tau] = \sum_k \sum_{\tau \to \tau_1 \tau_2} \texttt{ways}[i][k][\tau_1] \times \texttt{ways}[k+1][j][\tau_2]$.

   Pour $s = \texttt{aabb}$, le tableau $\texttt{ways}[1][4][S]$ renverrait bien 2, ce qui confirme l'énumération manuelle de la question (a).
6. Segmentation de texte : C'est un cas particulier. On pose $\mathcal{T} = \{W, S\}$ (W pour mot valide du dico, S pour segment valide). Pour chaque mot $d = x_1 x_2 \dots x_m \in D$, on crée des règles intermédiaires et terminales pour s'assurer que sa lecture exacte force l'arbre à se réduire en W. Ensuite, on ajoute simplement les règles de concaténation de phrases : $S \to W \cdot S$ et $S \to W \cdot W$.
7. Système pour $\{a^nb^n\}$ et impossibilité du dictionnaire :
   — Système : On pose $\mathcal{T} = \{A, B, X, S\}$ avec $\text{type}(a) = \{A\}, \text{type}(b) = \{B\}$. Règles : $S \to A \cdot B$ (cas de base ab), $X \to S \cdot B$ et $S \to A \cdot X$ (permettant la construction récursive de la forme "emboîtée" $A \cdot (S \cdot B)$).
   — Impossibilité de D : Supposons par l'absurde qu'un tel dictionnaire D fini existe. D contenant des chaînes finies, il existe une longueur maximale de mot $L = \max_{w \in D} |w|$. Considérons la chaîne $a^Nb^N \in \{a^nb^n\}$ avec $N > L$. Puisqu'elle appartient à D^*, elle se décompose en mots $w_1 w_2 \dots w_k$ du dictionnaire. Le premier mot w_1 ne peut pas contenir de b, car s'il était de la forme a^ib^j, sa longueur serait $> N > L$ (puisque tous les a doivent être consommés avant de toucher aux b). Donc w_1 ne contient que des a : $w_1 = a^i$ (avec $1 \leq i \leq L$). Puisque $w_1 \in D$, la chaîne $(w_1)^2 = a^{2i}$ appartient obligatoirement à D^*. Or $a^{2i} \notin \{a^nb^n\}$ (pas de b). Il y a donc une contradiction flagrante : le modèle D^* ne peut pas exprimer la symétrie centrale stricte des langages algébriques non réguliers.