Exercices d'oral en Informatique

Lycée Pierre de Fermat, MP*

2011-2012

Voici un recueil d'exercices d'informatique portant sur le programme des classes prépa dont l'objectif est de faciliter la préparation à l'oral d'informatique des Écoles Normales Supérieures. Ils proviennent principalement des colles d'informatique posées par M. Ginoux aux MP*, mais il y a également d'autre sources telles que le TaupIC, compétition de programmation destinée aux élèves de classe prépa.

Énoncés

1 Exercices posés aux Écoles Normales Supérieures, oral d'informatique fondamentale

1.1 Transformations de langages, ensembles parfaitement périodiques

Source : Officiel de la Taupe.

1. Soit A un alphabet (fini). Montrer que $\{ww \mid w \in A^*\}$ n'est pas rationnel.
2. Soit $B = (Q, q_0, F, \delta)$ un automate fini déterministe complet. Pour tout mot w de A^*, on note $f_w : q \in Q \mapsto \delta(q, w) \in Q$. Soit f une fonction de Q dans Q. Montrer que $L_f = \{w \mid f_w = f\}$ est rationnel.
3. Soit L un langage rationnel. Montrer que $R(L) = \{w \mid ww \in L\}$ est rationnel.
4. Montrer que $S(L) = \{w \mid w^{|w|} \in L\}$ est rationnel.
5. Montrer que $T(L) = \{w \mid w^{2^{|w|}} \in L\}$ est rationnel.
6. On dit qu'un sous-ensemble E de $\mathbf{N}$ est *parfaitement périodique* s'il existe $n_0, p \in \mathbf{N}$ tels que $\forall n > n_0, n + p \in E \Leftrightarrow n \in E$. À l'aide de cette notion, généraliser (montrer, par exemple, que $U(L) = \left\{w \mid w^{2^{2^{|w|}}} \in L\right\}$ est rationnel).

1.2 Fractions de langage et rationalité

Vu sur forum.prepas.org.

Soit L un langage sur l'alphabet X. On note $\frac{1}{2}L = \{u \in X^* \mid \exists v \in X^*/uv \in L \text{ et } |u| = |v|\}$.

1. Déterminer $\frac{1}{2}L$ pour $L = (a(aba + bab))^*$.
2. Montrer que si L est rationnel, $\frac{1}{2}L$ est rationnel.

3. On note, pour $r \in \mathbf{N}^*$, $\frac{1}{r}L = \left\{ u \in X^* \middle| \exists v \in X^*/uv \in L \text{ et } |u| = \frac{1}{r}|uv| \right\}$.
   Montrer que si L est rationnel, $\frac{1}{r}L$ est rationnel.
4. Même question en remplaçant $\frac{1}{r}$ par $\frac{p}{q}$, $p \wedge q = 1$.
5. Soit $f \in \mathbf{Q}[X]$, on note $f^{-1}(L) = \{u \in X^* | \exists v \in X^*/uv \in L \text{ et } f(|u|) = |uv|\}$.
   Montrer que si L est rationnel, $f^{-1}(L)$ est rationnel.

2 Exercices posés en colle d'informatique

2.1 Découpage d'un polygone

On considère un polygone à n sommets $A_1, \ldots, A_n$. On cherche à le couper en deux en reliant 2 sommets par un segment, de façons à ce que les périmètres des 2 sous-polygones soient les plus proches possibles. Écrire un algorithme qui résout le problème en temps linéaire.

2.2 Transformation itérée sur des mots

Soit l'alphabet $X = \{a, b\}$. On s'intéresse à la transformation sur les mots de X^* qui transforme un facteur ab en un facteur bba (c'est-à-dire $w_1abw_2 \to w_1bbaw_2$). Montrer qu'il n'existe pas de mot sur lequel cette transformation peut s'appliquer une infinité de fois.

2.3 Codes préfixes

Soient X et A 2 alphabets. Un *code* est défini comme un morphisme injectif de monoïdes $c\colon X^* \to A^*$. Un morphisme p est dit *préfixe* si pour tout $(x, y) \in X^2$, $x \neq y$, $p(x)$ n'est pas préfixe de $p(y)$.

1. Montrer qu'un morphisme préfixe est un code ; la réciproque est-elle vraie ?
2. Soit c un code, on pose $D = \operatorname{Card} A$, $s = \sum_{x \in X} D^{-|c(x)|}$. Montrer que $s \leq 1$.
   Indication : on montrera que $s^k = O(k)$.
3. Montrer que pour tout code $c\colon X^* \to A^*$, il existe un code $c'\colon X^* \to A^*$ préfixe tel que $\forall w \in X^*, |c(w)| = |c'(w)|$.

2.4 Jetons sur un graphe

Soit G un graphe non orienté connexe. On note d_s le degré du sommet s. On place une pile de jetons sur chaque sommet du graphe ; le nombre de jetons sur le sommet s sera noté c_s.

On joue à un jeu dont la règle est la suivante : si $c_s \geq d_s$ pour un sommet s, on peut retirer d_s jetons au-dessus de s et en donner un à chaque sommet voisin de s. La partie se poursuit tant qu'il est possible de jouer l'un des sommets (on peut avoir des parties infinies...), si plusieurs sont jouables il n'y a pas de contrainte sur lequel jouer en premier.

Question préliminaire : comment fonctionnerait un programme qui simule un tel jeu ?

Question sérieuses :

1. Montrer que si une partie est infinie, tous les sommets ont été joués une infinité de fois.
2. Montrer que si une partie est finie, au moins un sommet n'a pas été joué.
3. Condition nécessaire simple sur le nombre de jetons pour espérer une partie finie ?

4. Montrer que pour une partie finie, toutes les autres parties jouées à partir de la même configuration de départ sont finies et parviennent à la même configuration finale.

Note : une variante de ce système avec un sommet puits est traitée dans le sujet d'écrit ENS Maths-Info 2009 où on s'en sert pour représenter un groupe.

2.5 Langage miroir

Soit L un langage rationnel, on note $\widetilde{L}$ son langage miroir.

1. Montrer que $\widetilde{L}$ est rationnel avec deux méthodes différentes.
2. Dans toute la suite, on fixe $L = X^{n-1}aX^*$ (X est l'alphabet, $\{a, b\} \subset X$).
   Automate déterministe complet reconnaissant L ?
3. Majorer le nombre d'états nécessaires pour construire un automate déterministe complet reconnaissant $\widetilde{L}$.
4. Montrer que le nombre d'états atteint justement cette majoration.

2.6 Problème d'optimisation

On a n skieurs de taille $h_1, \ldots, h_n$, m paires de skis de taille $s_1, \ldots, s_m$ ($m \geq n$). On veut minimiser $\sum_{k=1}^{n} \left| s_{f(k)} - h_k \right|$ où f est la fonction (injective) d'association skieur $\to$ ski. Trouver un algorithme qui détermine f avec une complexité raisonnable. (On calculera la complexité de la solution proposée.)

2.7 Complémentaire d'un langage reconnaissable

Question préliminaire : montrer que si L est un langage reconnu par un automate fini déterministe complet à n états, alors son complémentaire L^c est aussi reconnu par un AFDC à n états.

On veut maintenant montrer que le nombre d'états des automates non-déterministes les plus petits reconnaissant un langage rationnel et son complémentaire n'ont pas forcément le même ordre de grandeur. On pose, pour $n \in \mathbf{N}$,

$$L_n = \{uxvyw \mid x, y \in \{a, b\},\ x \neq y,\ u, v, w \in \{a, b\}^*,\ |v| = n - 1\}$$

1. Expression rationnelle décrivant L_n ?
2. Donner un automate qui reconnaît L_n avec $O(n)$ états.
3. Montrer qu'un automate reconnaissant L_n^c a au moins 2^n états.
   Indication : si $|u| = n$, que peut-on dire de uu ?

2.8 Sous-tableau maximal

Soit A un tableau de n entiers relatifs, on cherche le sous-tableau connexe qui maximise la somme de ses éléments, *i.e.* $(i, j) \in \mathbf{N}^2$ avec $1 \leq i < j \leq n$ tels que $\sum_{k=i}^{j} A[k]$ soit maximale.

1. Algorithme simple qui résout le problème ? Complexité ?
2. Chercher une méthode dichotomique avec une meilleure complexité.
3. Algorithme en $O(n)$?

2.9 Motifs inévitables

Soit Σ un alphabet fini, $X \subset \Sigma^*$.

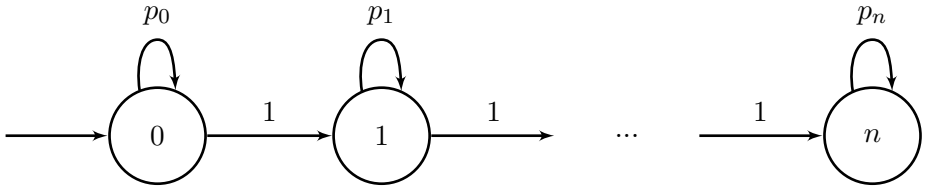
On dit qu'un mot u fini ou infini évite X si aucun mot de X n'est facteur de u. X est dit inévitable si aucun mot *infini* n'évite X.

(Rappel : les mots infinis sur Σ, parfois notés Σ^ω, sont les suites infinies à valeurs dans Σ. Σ^* est l'ensemble des mots *finis* sur Σ.)

1. Décrire un algorithme pour déterminer si $u \in \Sigma^*$ évite X, étant donné une fonction testant l'appartenance à X. Cas où X est fini et on a la liste de ses éléments. Discuter de la complexité.
2. Montrer que X est inévitable ssi $\Sigma^* \setminus (\Sigma^* X \Sigma^*)$ est fini.
3. Montrer que tout ensemble inévitable contient un ensemble inévitable fini.

2.10 Parcours sur un graphe

On considère le graphe orienté étiqueté suivant :



où les p_i sont des poids pris dans $\mathbf{N}^*$.

En partant de 0, on fait un parcours sur le graphe dont le poids est défini comme la somme des poids des arêtes qu'on emprunte (si on passe plusieurs fois par la même arête, on la compte plusieurs fois dans la somme).

Trouver un algorithme en temps polynomial qui dit si on peut circuler sur le graphe indéfiniment sans jamais avoir un poids cumulé d ($d \in \mathbf{N}^*$), c'est-à-dire qu'à tout moment, le poids du parcours déjà effectué ne vaut pas exactement d.

Indication : s'intéresser aux booléens $\lambda_{k,i}$ valant true ssi on peut arriver à l'état k avec un poids de i.

2.11 Langages rationnels coïncidant sur une partie finie

Soient L_1, L_2 deux langages rationnels sur un alphabet X. Montrer que pour certaines valeurs de N (à quantifier) on a $L_1 \cap X^N = L_2 \cap X^N \Rightarrow L_1 = L_2$. (Faire intervenir les automates reconnaissant les 2 langages).

On pourra d'abord montrer que si un automate à n états ne reconnaît aucun mot de longueur $\leq n-1$, alors le langage reconnu est vide.

2.12 Coupure équilibrée d'un tableau

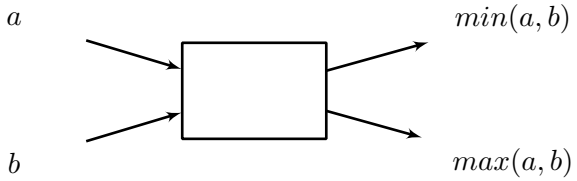
Soit un tableau A de taille n dont les cases prennent des valeurs dans $\{0, 1\}$. Écrire un algorithme qui trouve (i, j) tels que $|i - j|$ soit maximal et $A[i \dots j]$ contienne autant de 0 que de 1 (autrement dit, il faut trouver le sous-tableau maximal équilibré). Est-ce possible en $O(n)$?

2.13 Exo réflexe/astuce (langages rationnels)

Soit L un langage rationnel, définissons L' comme le langage des mots dont tous les suffixes sont dans L. Montrer que L' est rationnel.

2.14 Circuit de fusion de listes triées

Le but de l'exercice est de réaliser un circuit qui prend en entrée deux listes *triées* et dont la sortie est une liste triée qui contient les éléments de ces deux listes (comme dans l'algorithme de tri fusion). Pour cela, on dispose d'un composant comparateur-échangeur, qui permet d'obtenir le plus petit et le plus grand des deux nombres en entrée :



On se donne deux listes triées $a = [a_1; \ldots; a_n]$ et $b = [b_1; \ldots; b_n]$ où $n = 2^p$, $p \in \mathbf{N}^*$.

1. Circuit pour fusionner les listes dans les cas $p = 1$ et $p = 2$?
2. On pose :
   - $d = [d_1; \ldots; d_n]$ la fusion des listes $[a_1; a_3; \ldots; a_{n-1}]$ et $[b_1; b_3; \ldots; b_{n-1}]$
   - $e = [e_1; \ldots; e_n]$ la fusion des listes $[a_2; a_4; \ldots; a_n]$ et $[b_2; b_4; \ldots; b_n]$

   Montrer que la liste fusionnée à partir de a et b est
   $$f = [d_1; \min(d_2, e_1); \max(d_2, e_1); \ldots; \min(d_n, e_{n-1}); \max(d_n, e_{n-1}); e_n]$$
3. En déduire un circuit de fusion pour tout p. Quel est son nombre de portes ?
4. Comparer avec le nombre de comparaisons effectuées par l'algorithme traditionnel de fusion. Quel est l'avantage du circuit ?

3 Problèmes d'algorithmique du TaupIC

3.1 Exo débile

Comment calculer le plus court cycle d'un graphe orienté lorsqu'il existe ?

3.2 Jeu de Nim

(Vu dans Fort Boyard.) Les règles du jeu sont les suivantes : 2 joueurs retirent des allumettes sur un tas à tour de rôle. Lorsque c'est le tour d'un joueur, il peut prendre un certain nombre d'allumettes, puis c'est au tour de l'autre. Le gagnant est celui qui retire la dernière allumette.

Étant donné :

- la liste des coups permis *i.e.* l'ensemble (fini) des quantités d'allumettes qu'un joueur peut choisir de retirer lorsque c'est son tour
- la taille du tas initial d'allumettes

donner une procédure qui permet de décider quels sont, parmi les coups que peut jouer le joueur qui joue en premier, ceux qui sont le début d'une stratégie gagnante (c'est-à-dire quels sont ceux pour lesquels, en jouant celui-là en premier et parfaitement par la suite, le premier joueur est sûr de gagner).

3.3 Plus long palindrome

Étant donné un mot d'entrée, trouver le ou les sous-mot(s) qui forment les plus longs palindromes. Un sous-mot est obtenu en ne prenant que certaines lettres du mot d'origine et en conservant l'ordre (contrairement à un facteur, les lettres peuvent être dispersées dans le mot de départ).

3.4 Transvasement de liquide

On considère n récipients identiques, remplis de liquide à différents niveaux. On souhaite répartir le liquide de manière équilibrée dans les récipients en faisant un minimum de transvasements. Supposons par exemple qu'il y ait 3 récipients initialement remplis à 3L, 7L et 2L. Si on transvase 1L du deuxième vers le premier, on passe à l'état 4L / 6L / 2L. On transvase alors 2L du deuxième vers le troisième et on arrive à l'état 4L / 4L / 4L. Il nous a fallu deux transvasements.

Donner un algorithme qui, étant donné les volumes initiaux contenus dans les récipients, détermine le nombre minimal de transvasements nécessaires pour atteindre l'équilibre des volumes.

3.5 Polygone étoilé

Trouver un algorithme qui, à partir de la liste des sommets d'un polygone dans le plan (donnés dans l'ordre d'un parcours du périmètre dans le sens direct), détermine si l'intérieur du polygone est étoilé en un de ses points.

Corrections

1 Exos d'informatique fondamentale ENS

Corrections à venir...

2 Exercices posés en colle d'informatique

2.1 Découpage d'un polygone

1re question : structure de donnée à prendre en argument ? Il suffit d'un vecteur à n cases recensant les coordonnées de chaque point par exemple. En fait comme ce qui nous intéresse, ce sont les longueurs des côtés, on supposer qu'on dispose du tableau des distances $[\![a_1, \dots, a_n]\!]$ (où a_i est la longueur du segment $[A_iA_{i+1}]$).

Écriture d'un programme naïf : on remarque que minimiser l'écart entre les périmètres des 2 polygones découpés en traçant le segment $[A_iA_j]$ revient à minimiser l'écart de la longueur du contour comprise entre A_i et A_j au demi-périmètre du polygone de départ. Il suffit donc d'itérer sur les couples (i, j) avec $1 \leq i < j \leq n$ en gardant un compteur pour voir lequel minimise $\left|\sum_{k=i}^{j-1} a_k - \frac{p}{2}\right|$ où p est le périmètre.

Ce qui donne en pseudocode :

procedure DECOUPAGE($[\![a_1, \dots, a_n]\!]$)
  $p \leftarrow \sum_{k=1}^{n} a_k$ ▷ On précalcule le périmètre
  $min \leftarrow 9001 * p$, $i_{min} \leftarrow -1$, $j_{min} \leftarrow -1$
  for $i \leftarrow 1$ to $n-1$ **do** ▷ Boucle (1)
    for $j \leftarrow i$ to n **do** ▷ Boucle (2)
      $s \leftarrow \sum_{k=i}^{j-1} a_k$ ▷ Calculé en $O(j-i)$ avec une boucle
      if $\left|s - \frac{p}{2}\right| < min$ **then**
        $min \leftarrow \left|s - \frac{p}{2}\right|$, $i_{min} \leftarrow i$, $j_{min} \leftarrow j$
      end if
    end for
  end for
end procedure

Cet algorithme est en $O(n^3)$.

On peut déjà voir une amélioration possible : maintenir s dans une variable réinitialisée à chaque itération de la boucle (1) et incrémentée par la boucle (2) pour ne pas avoir à recalculer la somme à chaque fois : on obtient un algorithme en $O(n^2)$.

Autre remarque pertinente : à i fixé, il est inutile de regarder toutes les valeurs de j de i jusqu'à n. Les valeurs susceptibles de réaliser le minimum sont j_i et $j_i + 1$ telles que $s_{j_i} < \frac{p}{2}$ et $s_{j_i+1} \geq \frac{p}{2}$. On peut donc arrêter la 2e. boucle dès que s dépasse $\frac{p}{2}$, en utilisant une boucle while.

La dernière idée qui permet de faire baisser la complexité de $O(n^2)$ à $O(n)$ est que si on s'arrête à l'indice $j_i + 1$ avec $a_i + \ldots + a_{j_i-1} < \frac{p}{2}$, alors $a_{i+1} + \ldots + a_{j_i-1} < \frac{p}{2}$: pour chercher

l'indice j_{i+1} juste avant que s dépasse $\frac{p}{2}$ dans l'itération $i+1$ de la boucle (1), il suffit de commencer à $j_i + 1$, en retranchant a_i à s. Si on a directement $s - a_i \geq \frac{p}{2}$, alors $j_{i+1} = j_i$. On obtient au final le programme suivant :

procedure DECOUPAGE($[\![a_1, \ldots, a_n]\!]$)
  $p \leftarrow \sum_{k=1}^{n} a_k$
  $min \leftarrow 9001 * p$, $i_{min} \leftarrow -1$, $j_{min} \leftarrow -1$, $j \leftarrow 1$, $s \leftarrow 0$
  for $i \leftarrow 1$ to $n-1$ **do** ▷ Boucle (1')
    while $s < \frac{p}{2}$ **do** ▷ Boucle (2')
      $s \leftarrow s + a_j$
      $j \leftarrow j + 1$
    end while
    if $s - \frac{p}{2} < min$ **then** ▷ Test du couple (A_i, A_j)
      $min \leftarrow s - \frac{p}{2}$, $i_{min} \leftarrow i$, $j_{min} \leftarrow j$
    end if
    $s' \leftarrow s - a_{j-1}$
    if $\frac{p}{2} - s' < min$ **then** ▷ Test du couple (A_i, A_{j-1})
      $min \leftarrow \frac{p}{2} - s'$, $i_{min} \leftarrow i$, $j_{min} \leftarrow j - 1$
    end if
  end for
end procedure

Chaque tour de la boucle (2') fait augmenter j qui est compris entre 1 et n. Ainsi le nombre de tours de la boucle (2') au cours de toute l'exécution du programme est majoré par n. Comme le coût de chaque itération est $O(1)$, l'algorithme a une complexité $O(n)$.

2.2 Transformation itérée sur des mots

On peut remarquer que :
- le nombre de lettres a dans le mot est préservé par la transformation ;
- la transformation a tendance à « rapprocher » les a de la fin.

D'où l'idée de s'intéresser au n-uplet des positions des différents a dans le mot en partant de la fin (en supposant qu'il y ait n fois la lettre a dans le mot de départ). Soit une suite de mots $(u_1, u_2 \ldots)$, tels que $\forall i,\ u_i \to u_{i+1}$; alors la suite des n-uplets associée est strictement décroissante dans l'ordre lexicographique. Comme $\mathbf{N}$ est bien ordonné, l'ordre lexicographique sur $\mathbf{N}^n$ est bien fondé, donc la suite doit être finie. Ce qui apporte une réponse négative à la question posée.

2.3 Codes préfixes

1. Soit c un morphisme préfixe. Supposons par l'absurde qu'il ne soit pas injectif, alors $\exists u, v \in X^*,\ u \neq v \mid c(u) = c(v)$. Quitte à simplifier par un préfixe commun on peut supposer que la première lettre diffère : $u = u_1 u'$ et $v = v_1 v'$ avec $u_1 \neq v_1$. Sans perte de généralité supposons $|c(u_1)| \leq |c(v_1)|$. $c(u_1)c(u') = c(v_1)c(v')$ donc $c(u_1)$ est préfixe de $c(v_1)c(v')$ et la condition sur la longueur donne $c(u_1)$ préfixe de $c(v_1)$. Ce qui n'est pas possible pour un morphisme préfixe. Le cas où u ou v est le mot vide est trivial…

2. Prendre $X = A = \{a, b\}$, et le morphisme c tel que $c(a) = a$, $c(b) = ab$. Ce morphisme n'est évidemment pas préfixe. Par contre c'est un code : à w fixé, si v est un antécédent de

w par c, soit w commence par aa et alors v commence par a, soit w commence par ab et alors v commence par b ; par récurrence sur la longueur, l'antécédent est unique (sous réserve d'existence).

3. Faisons d'abord le lien entre l'indication et le résultat demandé : si $s > 1$, s^k croît géométriquement, $s^k \neq O(k)$.

On a en développant le produit

$$s^k = \sum_{(x_1,\dots,x_k)\in X^k} D^{-|c(x_1)|-\dots-|c(x_k)|} = \sum_{u\in X^k} D^{-|c(u)|}$$

Si on pose $\mu = \max_{x\in X} |c(x)|$, alors $\forall u \in X^*, |c(u)| \leq \mu|u|$. Donc

$$s^k \leq \sum_{\substack{w\in A^*\setminus\{\varepsilon\}\\ |w|\leq \mu k}} D^{-|w|} = \sum_{i=1}^{\mu k} \sum_{w\in A^i} D^{-i}$$

et comme $\operatorname{Card} A^i = (\operatorname{Card} A)^i = D^i$,

$$s^k \leq \sum_{i=1}^{\mu k} D^i \times D^{-i} = \mu k$$

CQFD.

4. Pour cette question, comme on vient de démontrer juste avant un résultat pas forcément intuitif dont on ne voit pas immédiatement l'utilité, on pense à le réutiliser ici.

Nous allons construire le code préfixe c' à partir de c. Si $x \in X$ est tel que $|c(x)| = 1$, il faut que $|c'(x)| = 1$, on peut prendre $c'(x) = c(x)$. Il faut ensuite assigner à chaque lettre x telle que $|c(x)| = 2$ une image par c' à 2 lettres. Problème : y a-t-il assez de paires de lettres différentes pour tous les x qui nous intéressent, sachant que la condition de code préfixe impose de prendre la première lettre parmi celles non utilisées pour les x avec $|c(x)| = 1$.

Notons k_n le nombre de lettres $x \in X$ telles que $|c(x)| = n$. En excluant k_n possibilités pour la première lettre, ça nous laisse $D(D - k_1)$ paire de lettres. On a

$$\sum_{x\in X} D^{-|c(x)|} \leq 1 \Rightarrow Dk_1 + k_2 + \sum_{n=2}^{\infty} k_n D^{-n+2} \leq D^2 \Rightarrow k_2 \leq D(D - k_1)$$

C'est bon, on peut définir l'image des lettres vérifiant $|c(x)|$ sans nuire au caractère préfixe du code c' ! On continue ainsi en examinant les lettres x par ordre croissant de $|c(x)|$, le résultat de la question 3 garantissant l'existence de mots de longueur $|c(x)|$ pouvant être définis comme les images des x par c', finalement on obtient un code préfixe c' qui vérifie bien $\forall x$, $c'(x) = c(x)$. La propriété étant vraie sur les lettres elle est vraie sur les mots.

2.4 Jetons sur un graphe

Programme de simulation

On numérote les sommets de 1 à n, le graphe est représenté par une des listes d'adjacence ou une matrice, peu importe. On précalcule les degrés d_s qu'on enregistre dans un tableau, les c_s sont stockés dans un tableau mutable.

Le programme est composé d'une boucle while qui tourne tant que la partie peut continuer. À chaque itération, on boucle sur les sommets jusqu'à en trouver un tel que $c_s \geq d_s$, on le joue alors en modifiant le tableau des c_s. Si on ne trouve pas de sommet jouable la partie est finie, on modifie une variable booléene qui signale la fin de la grande boucle. On peut parsemer de commandes d'affichage pour observer le déroulement de la partie...

Questions théoriques

1. Supposons qu'un des sommets soit joué un nombre fini de fois dans une partie. À partir d'un moment, ce sommet n'est plus joué. Comme le nombre total de jetons dans le graphe se conserve et est fini, le nombre de jetons sur ce sommet est borné. Or le nombre de jetons ne diminue pas sans que le sommet soit joué, et il augmente quand un sommet voisin est joué. On en déduit que chaque voisin ne peut être joué qu'un nombre fini de fois. En exploitant la connexité du graphe et en faisant une récurrence sur la distance au sommet qu'on a pris au départ, on montre alors que tout sommet est joué un nombre fini de fois. La partie est donc fini. On a montré la contraposée de la proposition demandée.

2. Par l'absurde, on suppose que tous les sommets sont joués au moins une fois. La partie étant finie, ils sont tous joués un nombre fini de fois. On prend le sommet qui est le premier à être joué pour la dernière fois : chacun de ses voisins va être joué au moins une fois après cette dernière fois. Après que cela s'est produit, ce sommet a acquis de ses voisins suffisamment de jetons pour être jouable (si chaque voisin lui donne un jeton ou plus, il se retrouve avec plus de jetons que son degré). Comme il n'est plus joué jusqu'à la fin de la partie, il est encore jouable lorsque la partie s'arrête. Or on joue jusqu'à ne plus pouvoir continuer ! On a une contradiction.

3. Si on prend le nombre de jetons total plus grand que la somme des degrés des sommets (qui s'écrit aussi comme 2 fois le nombre d'arêtes), il y aura toujours un sommet avec plus de jetons que son degré, qui sera jouable. La partie se poursuit alors à l'infini. On a une condition suffisante de partie infinie, sa négation est une condition nécessaire de partie finie.

4. Soit C_0 la configuration de départ (une configuration étant le n-uplet des c_s qui décrit l'état du graphe), la partie finie de référence consiste en la suite de configurations $C_0 \to C_1 \to \ldots \to C_f$. Considérons une autre partie qui diverge de la première à partir de la configuration k, sa suite de configurations est $C_0 \to \ldots \to C_k \to C'_{k+1} \to \ldots$ avec $C_{k+1} \neq C_{k+1}$.

Disons que l'on passe de C_k à C_{k+1} en jouant i, et à C'_{k+1} en jouant $j \neq i$. i et j sont tous les deux jouables dans la configuration C_k. Le sommet j est éventuellement joué dans la première branche : en effet, dans le cas contraire, la partie s'achève alors que j est toujours jouable (la branche est celle d'une partie finie). On montre aussi que i est éventuellement joué dans la deuxième branche : dans le cas contraire cette deuxième branche est finie (contraposée du 1.), i reste jouable jusqu'à la fin, contradiction.

La partie $C_k \xrightarrow{i} C_{k+1} \to \ldots \xrightarrow{j} C_{k+l} \to \ldots$ peut être réarrangée en $C_k \xrightarrow{i} C_{k+1} \xrightarrow{j} \widehat{C}_{k+2} \to \ldots \to \widehat{C}_{k+l-1} \to C_{k+l} \to \ldots$ en jouant plus tôt le sommet j sans modifier le déroulement de la partie à partir de C_{k+l}, en particulier la longueur (finie) et la configuration finale de la partie ne changent pas. On peut faire le même genre de transformation sur la deuxième partie qui n'influence pas son caractère fini ou infini.

Ainsi on peut avancer le moment k ou il y a bifurcation à $k+2$, et en itérant le processus on aura bifurcation à C_f. Or aucun sommet n'étant jouable à C_f toute branche qui en part est vide. Donc la deuxième partie est bien finie et débouche sur la configuration C_f.

On a en fait mieux : non seulement le caractère fini et la configuration finale sont indépendants de la partie, la longueur et le nombre de fois où chaque sommet est joué le sont aussi.

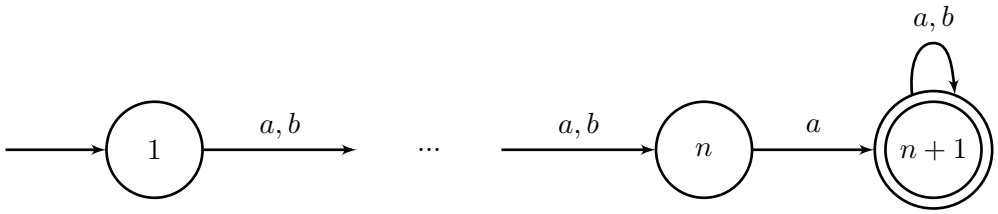
2.5 Langage miroir

1. Grâce à l'équivalence entre langages rationnels et reconnaissables (théorème de Kleene) on peut s'y prendre des 2 façons suivantes :

- 1re méthode : par induction structurelle. Le miroir de l'union est l'union des miroirs, même chose pour l'étoile, et $\widetilde{L_1 L_2} = \widetilde{L_2}\,\widetilde{L_1}$ ...

– 2e méthode : avec les automates finis. On prend un automate reconnaissant L, on renverse le sens des transitions et on échange états initiaux et états finaux pour avoir un automate reconnaissant $\widetilde{L}$.

2. L'automate suivant marche :



3. L'automate miroir de l'automate précédent a $n+1$ états, mais il est non-déterministe. En le déterminisant on obtient un automate avec $\leq 2^{n+1}$ états. On peut avoir mieux : quelque soit le mot lu, il est possible de boucler indéfiniment sur le premier état de l'automate miroir, donc celui-ci appartient à tous les états accessibles de l'automate des parties, ce qui permet de diviser par deux notre majoration du nombre d'états. Le majorant recherché est 2^n.

4. Dans un automate déterministe complet reconnaissant $\widetilde{L}$, il y a au moins autant d'états que de résiduels de $\widetilde{L}$ (cf. cours sur les automates minimaux et les résiduels). Nous allons montrer qu'il y a 2^n résiduels ce qui prouvera le résultat demandé.

$\widetilde{L}$ est le langage des mots qui se terminent par un suffixe de la forme au avec $|u| = n-1$, c'est-à-dire que la $n^{\text{ième}}$ lettre en partant de la fin du mot est a. Ainsi,

$$\forall k \in [\![0, n-1]\!],\ \forall w \in X^*,\ a^k \in w^{-1}\widetilde{L} \Leftrightarrow wa^k \in \widetilde{L} \Leftrightarrow \text{la } (n-k)^{\text{ième}} \text{ lettre de } w \text{ est } a$$

Ce critère permet de distinguer les résiduels de $\widetilde{L}$ par les mots de $\{a,b\}^n$. Il y a donc au moins 2^n résiduels différents, et comme on a un AFDC à 2^n états reconnaissant le langage il y a en fait exactement 2^n résiduels différents.

2.6 Problème d'optimisation

La méthode naïve serait d'énumérer toutes les fonctions f, pour chercher laquelle réalise le minimum. Les fonctions convenables sont les fonctions injectives de $[\![1,n]\!]$ dans $[\![1,m]\!]$, et il y en a $\dfrac{m!}{(m-n)!} \geq n!$: ça croît très rapidement, on ne peut pas espérer avoir une complexité raisonnable avec cette méthode...

On va donc changer de stratégie. D'abord, on se ramène au cas où les listes h et s sont croissantes, en effectuant un tri en $O(n \log n)$ (pour l'instant, c'est pas trop coûteux...). Ensuite, dans le doute, on essaie un algorithme de programmation dynamique : on va donc s'intéresser au problème de minimiser la somme en ne prenant en compte que les i premiers skieurs et les j premiers skis, et noter $f_{i,j}$ la fonction de répartition solution du problème.

Comment obtenir la solution en fonction des valeurs plus petites de i et j ?

– Si la répartition optimale ne fait pas intervenir le j-ième ski, alors $f_{i,j} = f_{i,j-1}$;
– Sinon, a priori, c'est plus compliqué...

On se dit que ça serait bien qu'on aie forcément, dans le 2ème cas, $f_{i,j}(i) = j$, vu qu'on a trié les listes...mais comment le prouver ? En prenant une fonction f telle que j soit dans l'image de f mais ne soit pas l'image de i, et en trouvant une autre fonction g avec $g(i) = j$ qui rende la somme qu'on cherche à minimiser plus petite que f.

Posons $j_0 = f(i)$ et $i_0 = f^{-1}(j)$. Le but va être de montrer que la fonction

$$g(k) = \begin{cases} f(k) & \text{si } k \neq i, i_0 \\ j_0 & \text{si } k = i_0 \\ j & \text{si } k = i \end{cases}$$

est une meilleure répartition que f.

On sait qu'on a $h_{i_0} \leq h_i$ et $s_{j_0} \leq s_j$ vu que les (h_i) et les (s_j) sont dans l'ordre croissant. Reste donc 6 cas à distinguer :

- $s_{j_0} \leq s_j \leq h_{i_0} \leq h_i$:
  $|h_{i_0} - s_j| + |h_i - s_{j_0}| = h_{i_0} - s_j + h_i - s_{j_0} = h_i - s_j + h_{i_0} - s_{j_0} = |h_i - s_j| + |h_{i_0} - s_{j_0}|$
  donc échanger i et i_0 ne change rien
- $h_{i_0} \leq h_i \leq s_{j_0} \leq s_j$: la même chose avec un changement de signe
- $s_{j_0} \leq h_{i_0} \leq s_j \leq h_i$:
  $|h_{i_0} - s_j| + |h_i - s_{j_0}| = s_j - h_{i_0} + h_i - s_{j_0} = h_i + s_j - h_{i_0} - s_{j_0}$
  et comme $s_j \geq s_{j_0}$,
  $|h_{i_0} - s_j| + |h_i - s_{j_0}| \geq h_i - s_j - h_{i_0} + s_{j_0} = |h_i - s_j| + |h_{i_0} - s_{j_0}|$
  échanger i et i_0 diminue la somme des écarts de taille entre ski et skieur
- $h_{i_0} \leq s_{j_0} \leq h_i \leq s_j$: même chose...
- $s_{j_0} \leq h_{i_0} \leq h_i \leq s_j$: laissé au lecteur, tout comme le même cas en échangeant h et s...

cqfd.

Une fois ceci fait, on exploite le fait que si $f_{i,j}$ est une répartition optimale telle que $f_{i,j}(i) = j$, alors elle induit une distribution optimale des $j-1$ skis aux $i-1$ skieurs, donc on aura

$$f_{i,j}(k) = \begin{cases} f_{i-1,j}(k) & \text{si } k \geq i-1 \\ j & \text{si } k = i \end{cases}$$

Il suffit ensuite de comparer la fonction ainsi construite à la solution où le j-ième ski n'intervient pas à savoir $f_{i,j-1}$ pour avoir la solution optimale *i.e.* $f_{i,j}$. Si on ne cherche qu'à connaître la valeur minimale de la somme, notons-la $a_{i,j}$, pour les i premiers skieurs et j premiers skis, alors elle se calcule par la simple relation de récurrence

$$a_{i,j} = \min\left(a_{i,j-1}, a_{i-1,j-1} + |h_i - s_j|\right)$$

De toute façon, c'est une bonne idée de stocker ça pour ne pas avoir à recalculer les valeurs de $a_{i-1,j-1}$ et $a_{i,j-1}$ avant d'effectuer la comparaison permettant de sélectionner la bonne fonction $f_{i,j}$.

Ensuite, c'est de la programmation dynamique tout à fait ordinaire, on remplit le tableau des $f_{i,j}$...mais il n'y a pas besoin de toutes les valeurs de $f_{i,j}$ pour $1 \leq i \leq n$, $1 \leq j \leq m$.

En effet, dans la relation de récurrence, on observe qu'on fait appel à des valeurs de f pour des couples dont l'écart $j - i$ est plus petit que celui du couple actuel. De plus, si $i > j$, il n'y a pas assez de skis pour tout le monde. Comme l'écart initial est $m - n$, on en déduit qu'il suffit de remplir le tableau pour $1 \leq i \leq n$, $i \leq j \leq i + m - n$, ce qui donne $n(m - n + 1)$ cases à remplir en $O(1)$ chacune.

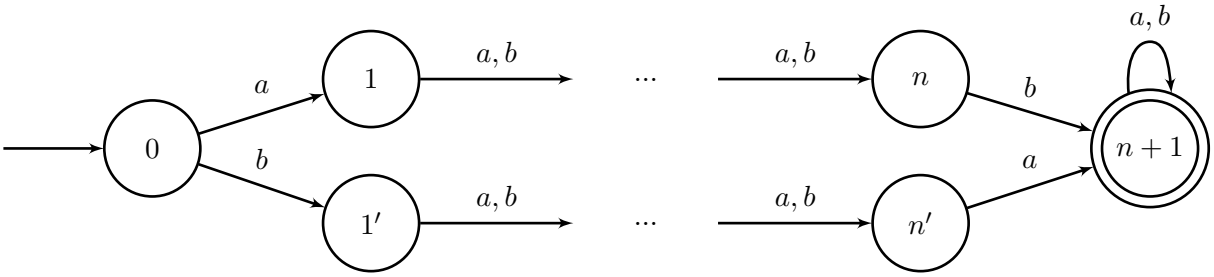
L'algorithme final est d'une complexité $O(n(m - n + 1))$, ce qui est très satisfaisant...

2.7 Complémentaire d'un langage reconnaissable

Question préliminaire : soit un AFDC reconnaissant L, le même automate où les états finaux sont considérés non finaux et vice-versa reconnaît L^c et cette transformation préserve évidemment le nombre total d'états.

1. $L_n = X^*aX^{n-1}bX^* + X^*bX^{n-1}aX^*$.

2. L'automate suivant reconnaît L_n avec $2n + 2$ états :



3. Si $|u| = n$, uu est un mot n-périodique.

L_n est le langage des mots dans lesquels on peut trouver 2 lettres distantes de n différentes. Son complémentaire est composé des mots où deux lettres distantes de n sont toujours égales, c'est-à-dire des mots n-périodiques. Donc $\forall u \in \{a,b\}^n, uu \in L_n^c$. Ainsi, si $\mathcal{A}$ est un automate reconnaissant L_n^c, il accepte tous les tels uu.

Inversement, si $|u| = n$, $|v| = n$, $u \neq v$ alors uv n'est pas n-périodique (si $u_i \neq v_i$, $(uv)_i \neq (uv)_{n+i}$), donc $uv \notin L_n^c$. Deux chemins réussis dans l'automate étiquetés par uu et vv (qui existent forcément car $uu, vv \in L_n^c$) ont donc des n-ièmes états différents (sinon, en recollant les chemins, uv serait accepté par $\mathcal{A}$). Comme Card $\{a,b\}^n = 2^n$, il y a au moins 2^n états différents dans $\mathcal{A}$.

2.8 Sous-tableau maximal

1. C'est le même principe que l'exo « découpage d'un polygone » sauf qu'on essaie de maximiser la somme plutôt que de la rapprocher d'une valeur donnée... Se référer donc à cet exercice pour avoir un algorithme naïf en $O(n^3)$.

2. On distingue 3 cas :

- Soit le sous-tableau maximal est inclus dans la moitié gauche du tableau (entre 1 et $\left\lfloor \frac{n}{2} \right\rfloor$)
- Soit il est inclus dans la moitié droite
- Soit il chevauche les deux : alors c'est la réunion d'un sous-tableau de la forme $A\left[i \dots \left\lfloor \frac{n}{2} \right\rfloor\right]$ et d'un autre de la forme $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 1 \dots j\right]$, et il est maximal lorsque ses 2 moitiés sont maximales

On calcule le maximum pour les différents cas. Les 2 premiers consistent en des appels récursifs et le 3ème se fait en $\Theta(n)$ (c'est le coût pour maximiser $A\left[i \dots \left\lfloor \frac{n}{2} \right\rfloor\right]$ en décrémentant i à partir de $\left\lfloor \frac{n}{2} \right\rfloor$ et en mettant à jour la somme en $O(1)$ à chaque décrémentation...). D'où la formule de la complexité

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

qui se résout en $C(n) = \Theta(n \log n)$ (c'est la même chose que le tri fusion).

Note : pour implémenter ça en Caml, introduire une fonction auxiliaire récursive qui prend en argument les bornes du sous-tableau sur lesquelles on opère.

3. Pour j fixé on pose

$$g_j = \max_{i \leq j} \sum_{k=i}^{j} A[k]$$

On calcule $g_1, \dots, g_n$ avec la relation de récurrence

$$g_{j+1} = \max\left(g_j + A[j+1], A[j+1]\right)$$

ce qui se fait en $O(n)$. Ensuite, on peut trouver simplement le maximum, qui est le maximum pris sur les g_j.

Pour trouver le couple (i, j) qui réalise le maximum on peut retenir le i qui réalise le maximum pour chaque j.

2.9 Motifs inévitables

1. Algorithme simple : on teste tous les facteurs de u. La complexité dépend évidemment du coût du prédicat qui teste si un facteur appartient à X. Dans le cas où celui-ci est en $O(1)$, la complexité est en $O(|u|^2)$ car un mot de longueur n a $\frac{n(n+1)}{2}$ facteurs. Si X est fini et qu'on a la liste de ses éléments, chercher si un mot est dans X prend un temps $O(|X|)$, d'où un temps $O(n^2|X|)$ pour l'algorithme (on pourrait améliorer ça en triant la liste des mots dans l'ordre lexicographique et en utilisant une recherche dichotomique en $O(\log |X|)$).

Autre solution si X est fini : on construit un automate reconnaissant les facteurs de u et on lui fournit en entrée les mots de X.

2. Notons $L = \Sigma^* \setminus (\Sigma^* X \Sigma^*)$. $\Sigma^* X \Sigma^*$ étant le langage des mots (finis) qui ont un facteur dans X, son complémentaire L est le langage des mots qui évitent X.

Le sens le plus facile est le sens réciproque. Si L est fini, on note l la longueur de son plus grand mot. Soit $\mu \in \Sigma^\omega$, alors $\mu_1 \dots \mu_{l+1}$ est un mot fini qui n'est pas dans L, il contient donc un facteur dans X. Par conséquent, μ n'évite pas X, et on peut même dire qu'on a un facteur de μ qui est dans X « au début » du mot (important pour le (3.) !).

Passons maintenant au sens direct, que dont nous montrerons la contraposée en partant de l'hypothèse L infini. Alors il existe une infinité de mots commençant par la même lettre (en effet, Σ est fini). Notons a_1 une lettre qui convient, et posons $L_1 = a_1^{-1} L$, il est infini. On peut encore trouver a_2 tel que $L_2 = a_2^{-1} L_1$ soit infini. Par récurrence on construit une suite infinie $(a_1, a_2, \dots)$ telle que $\forall n \in \mathbf{N}^*$, $L_n = (a_1 \dots a_n)^{-1} L$ soit infini. Posons $a = a_1 a_2 \dots$, c'est un mot infini. De plus, tout facteur fini de a se trouve dans un préfixe fini $a_1 \dots a_n$ de a, préfixe commun à a et à un mot de $a_1 \dots a_n L_n \subset L$ (infini, donc non vide !) ; c'est donc c'est un facteur d'un mot qui évite X, il n'est pas dans X. En d'autres termes, a évite X, donc X n'est pas inévitable.

3. Reprenons la preuve du sens réciproque du 2. En notant $l = \max_{w \in \Sigma^* \setminus (\Sigma^* X \Sigma^*)} |w|$, on sait qu'on peut trouver un facteur dans X dans le préfixe de taille $l+1$ de tout mot infini. Cela signifie, en particulier, que le facteur en question est de taille $\leq l+1$: on obtient que l'ensemble des mots de X de taille $\leq l+1$ est inévitable.

En fait, le 2. permet d'avoir les implications suivantes :

$$X \text{ inévitable } \Rightarrow \Sigma^* \setminus (\Sigma^* X \Sigma^*) \text{ fini } \Rightarrow \exists l \in \mathbf{N} \mid X \cap (\Sigma \cup \{\varepsilon\})^{l+1} \text{ inévitable}$$

Comme $X \cap (\Sigma \cup \{\varepsilon\})^{l+1}$ est fini est inclus dans X, on a bien le résultat voulu.

2.10 Parcours sur un graphe

La première remarque qu'on peut faire est qu'au fur et à mesure qu'on avance sur le graphe, le poids cumulé augmente (il n'y a pas de poids négatifs sur les arêtes). La question est donc : peut-on dépasser d sans passer par d ? Une fois cela fait, il n'y aura pas de problème pour continuer le parcours...Évidemment, si on emprunte une arête d'étiquette 1, on ne pourra pas passer d'une valeur $< d$ à une valeur $> d$. Le saut éventuel se fera donc en empruntant la boucle d'un sommet k tel que $p_k > 1$; on se ramène ainsi au problème de savoir s'il est possible d'arriver à un sommet k avec un poids compris entre $d - p_k + 1$ et $d - 1$.

Pour y répondre, on introduit les variables booléennes $\lambda_{k,i}$ définies comme dans l'indication. Sachant que dans un chemin aboutissant au sommet k, la dernière arête empruntée était soit la liaison $(k-1,k)$, soit la boucle de poids p_k, on a la relation :

$$\lambda_{k,i} = \lambda_{k-1,i-1} \vee \lambda_{k,i-p_k}$$

Ceci permet de calculer les $\lambda_{k,i}$ avec de la programmation dynamique, c'est-à-dire de remplir le tableau des $\lambda_{k,i}$, $0 \leq k \leq n$, $0 \leq i \leq d-1$, en partant des petites valeurs de k et de i et en accédant aux valeurs déjà calculées pour obtenir les nouvelles valeurs par récurrence.

Le résultat cherché ensuite est $\bigvee_{k=0}^{n} \left(\bigvee_{i=d-p_k+1}^{d-1} \lambda_{k,i} \right)$.

2.11 Langages rationnels coïncidant sur une partie finie

Commençons par le lemme préliminaire. Soit un automate $\mathcal{A}$ à n états qui reconnaît le langage L. Supposons que $L \neq \emptyset$, et choisissons $u \in L$ quelconque. u étiquette un chemin d'un état initial à un état final dans $\mathcal{A}$. En éliminant toutes les boucles de ce chemin pour obtenir un chemin simple, où chaque état de l'automate n'apparaît qu'une fois, on obtient un chemin réussi de longueur $\leq n-1$. Son étiquette est un mot de L, donc L contient bien un mot de longueur $\leq n-1$. Par contraposition, si $\mathcal{A}$ ne reconnaît aucun mot de longueur $\leq n-1$, $L = \emptyset$.

Autre façon de procéder si on veut faire le malin (c'est la même idée) : lorsque $|u| \geq n$, il existe $x, y, z \in X^*$ avec $y \neq \varepsilon$ tels que $u = xyz$ et $xy^*z \subset L$, et en utilisant $\varepsilon \in y^*$, $xz \in L$ et $|xz| < u$. Donc le mot de longueur minimale dans L est forcément de longueur $\leq n-1$.

Passons maintenant à la question principale. On veut $L_1 = L_2$, c'est-à-dire $L_1 \triangle L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = \emptyset$. Or, si on dispose d'automates déterministes de taille n_1 et n_2 reconnaissant respectivement L_1 et L_2, alors on sait construire un automate reconnaissant la différence symétrique à partir de l'automate produit avec $n_1 \times n_2$ états (voir cours). En appliquant le lemme préliminaire à cet automate, $N \geq n_1 \times n_2$ convient.

2.12 Coupure équilibrée d'un tableau

Algorithme naïf : évident, voir les autres exos sur les tableaux...

Pour avoir un algorithme en $O(n)$, on introduit la notion de déséquilibre d'un sous-tableau : c'est la différence entre le nombre de 1 et le nombre de 0 qui se trouvent dans ce tableau (c'est donc un entier relatif). On pose $d(i)$ le déséquilibre de $A[1 \dots i]$. On cherche le sous-tableau maximal dont le déséquilibre est nul, et comme le déséquilibre de $A[i \dots j]$ vaut $d(j) - d(i-1)$, la dernière condition se traduit en $d(i-1) = d(j)$.

Il y a évidemment un nombre limité de valeurs possibles pour le déséquilibre. Les cas extrémaux sont lorsque le tableau ne contient que des 0, le déséquilibre valant alors $-n$, ou lorsqu'il ne contient que des 1, le déséquilibre dans ce cas étant $+n$. Il y a $2n+1$ valeurs différentes pour le déséquilibre.

On veut trouver i et j le plus éloignés possibles avec la même valeur de d. Pour cela, on initialise 2 tableaux à $2n+1$ cases, une pour chaque valeur possible du déséquilibre. Une boucle parcourant le tableau A une fois (évidemment en $O(n)$) permet de stocker dans ces tableaux le plus petit et le plus grand indice réalisant chaque valeur de d. Il suffit ensuite de regarder chaque valeur possible trouver le couple (i,j) d'écart maximal.

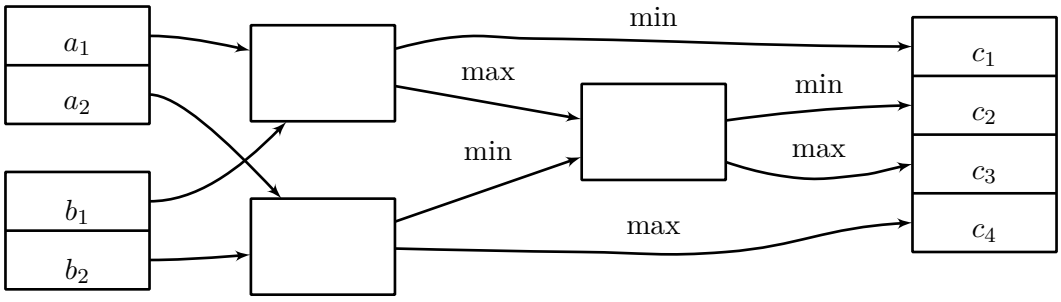
2.13 Exo réflexe/astuce

On se souvient que la classe des langages rationnels est stable par passage au complémentaire, et on écrit : $X^* \setminus L' = X^*(X^* \setminus L)$ (en effet, le complémentaire du langage des mots dont tous

les suffixes sont dans L est le langage des mots dont un des suffixes n'est pas dans L, donc est dans $X^* \setminus L$). La rationalité de L' est donc assurée...(et oui, c'est débile !)

2.14 Circuit de fusion de listes triées

1. Cas $p = 1 \Rightarrow n = 2$: soient 2 listes $[a_1; a_2]$ et $[b_1; b_2]$ triées ($a_1 \leq a_2$ et $b_1 \leq b_2$). Le plus petit élément est forcément $\min(a_1, b_1)$, et le plus grand est $\max(a_2, b_2)$. Il reste 2 éléments à ordonner pour les 2 positions au milieu, ce que l'on peut faire avec un comparateur-échangeur. Cela mène finalement au circuit suivant :



$[c_1; c_2; c_3; c_4]$ étant la liste fusionnée. Pour le cas $p = 2$, c'est fastidieux...

2. Notons déjà que f, fusion de a et b, est aussi la fusion de d et e : chaque élément d'indice impair (resp. pair) dans a ou b apparaît exactement une fois dans d (resp. e).

Il est déjà facile de justifier le premier et le dernier élément de la fusion. d_1 est le plus petit élément parmi les éléments d'indice impair de a et de b, parmi lesquels on trouve a_1 et b_1 qui atteignent le minimum des listes a et b. Donc $d_1 = \min(a_1, b_1)$ et c'est en fait le plus petit parmi tous les éléments de a et de b, même en comptant ceux d'indice pair. Sa place est donc à la tête de la liste f. Le même genre d'argument est valable pour e_n.

Pour les autres cas, fixons-nous $i \in \mathbf{N}$, et cherchons des contraintes sur la position de d_i dans f. d_i est un élément d'indice impair soit de a, soit de b. Par symétrie, on peut prendre sans perte de généralité $d_i = a_{2k-1}$. $d_1, \ldots, d_i$ sont les i plus petits éléments d'indice impair dans a et b.

Les éléments de a qui en font partie sont $a_1, a_3, \ldots a_{2k-1}$. En effet, si a_{2k-1} est le i-ème plus petit élément de d, comme les a_{2j-1} avec $j < k$ sont dans d et sont encore plus petits que a_{2k-1} par croissance de la liste a, ils font donc partie des i plus petits éléments de d. Inversement, les a_{2j-1}, $j > k$, sont trop grands pour en faire partie.

Cela nous donne k éléments parmi $d_1, \ldots, d_i$, il en manque donc $i - k$. Dans le cas où $i > k$, les éléments manquants doivent provenir de la liste b. Comme celle-ci est triée et qu'on prend les plus petits, ces éléments sont forcément $b_1, \ldots, b_{2(i-k)-1}$. On a donc $b_{2(i-k)-1} \leq d_i$.

Par croissance de a, $\forall j \in [\![1, 2k-2]\!], a_j \leq a_{2k-1} = d_i$. De même sur b : $\forall j \in [\![1, 2(i-k)-1]\!], b_j \leq b_{2(i-k)-1} \leq d_i$. Donc dans la liste fusionnée finale f, il y a au moins $2k-2+2(i-k)-1 = 2i-3$ éléments qui précèdent d_i. Donc la position de d_i dans f est $\geq 2i-2$.

De façon analogue, en raisonnant à partir de la fin de la liste d et des $n - i$ plus grands éléments, on montre que la position de d_i dans la fusion de a et b doit être $\leq 2i-1$. Donc d_i occupe dans f soit la place $2i-2$, soit la place $2i-1$. On montre de même que e_{i-1} doit occuper une de ces deux places dans f.

Comme f est croissante, on a forcément $f_{2i-2} = \min(d_i, e_{i-1})$, et $f_{2i-1} = \max(d_i, e_{i-1})$. C'est le résultat attendu.

3. Le circuit pour p se déduit de celui pour $p-1$: on se sert de deux circuits pour $p-1$ pour fusionner les termes pairs/impairs de a et b et obtenir les listes d et e de la question 2.

Ensuite, il suffit d'employer la formule de la question précédente, en rajoutant un niveau avec $n-1$ comparateurs-échangeurs pour placer les différents éléments.

Le nombre de portes N_p obéit à la relation de récurrence :

$$N_p = 2^p - 1 + 2N_{p-1} \text{ avec } N_0 = 1$$

Cette récurrence admet pour solution :

$$N_p = p2^p + 1 = n \log_2 n + 1$$

4. Dans le pire des cas, l'algorithme traditionnel effectue $2n$ comparaisons pour fusionner deux listes de taille n. C'est bien moins que le nombre de comparaisons effectuées par le circuit lorsque n est grand. Non seulement le circuit est plus compliqué, il fait plus de comparaisons !

Cependant, le temps mis pour effectuer le calcul ne dépend pas du nombre de portes du circuit, mais de sa profondeur, puisque les portes peuvent calculer en parallèle. La profondeur du circuit étant $\log_2 n$, on a une complexité avantageuse par rapport à l'algorithme traditionnel...

3 Problèmes TaupIC

À venir...