Feuille d'exercices n°14 - Révisions de MP2I

Notions abordées

- recherche de motif : algorithme de Boyer-Moore, de Rabin-Karp, automate des motifs
- compression de texte : algorithme de Huffman, algorithme LZW
- programmation dynamique (plusieurs exemples)
- diviser pour régner et calcul de complexité
- logique propositionnelle : mise sous FNC, mise sous FND
- graphe de flot de contrôle

Exercice 1 : Algorithme de Boyer-Moore (simplifié)

Q. 1 Quel problème résout l'algorithme de Boyer-Moore ?

Q. 2 Donner le pseudo-code d'un algorithme naïf qui résout ce problème. Préciser sa complexité.

Q. 3 Rappeler l'idée de l'algorithme de Boyer-Moore.

Q. 4 On suppose que la fenêtre de texte $t[\![i, i+|x|[\![$ ne coïncide pas avec le motif x du fait d'un mauvais caractère à l'indice j, *i.e.* parce que $t_{i+j} \neq x_j$. En se concentrant sur ce caractère, quel est l'indice i' qui définit la prochaine fenêtre du texte qui est susceptible de coïncider avec le motif ?

Q. 5 Appliquer l'algorithme découlant de la remarque précédente sur le texte `tatatututoto` et pour le motif `tuto`.

Q. 6 Identifier quelles données peuvent être précalculées à partir du motif x pour rendre plus efficace le calcul du décalage proposé à la **Q.4**. Donner alors le pseudo-code de ce pré-calcul et préciser sa complexité.

Q. 7 On suppose que la fenêtre de texte $t[\![i, i+|x|[\![$ ne coïncide pas avec le motif x du fait d'un mauvais caractère à l'indice j, *i.e.* parce que $t_{i+j} \neq x_j$. On suppose que ce caractère est le premier en partant de la droite où la fenêtre du texte et le motif ne coïncident plus, ainsi $t[\![j+1, i+|x|[\![= x[\![j+1, |x|[\![$. En se concentrant sur ce suffixe, quel est l'indice i' qui définit la prochaine fenêtre du texte qui est susceptible de coïncider avec le motif ?

Q. 8 Appliquer l'algorithme découlant de la remarque précédente♣ sur le texte `babbababbbaab` et pour le motif `babaab`.

Q. 9 Identifier quelles données peuvent être précalculées à partir du motif x pour rendre plus efficace le calcul du décalage proposé à la **Q.7**. Donner alors le pseudo-code de ce pré-calcul et préciser sa complexité.

Q. 10 Appliquer l'algorithme de Boyer-Moore combinant les deux remarques précédentes sur le texte `taratatauturlututu` et pour le motif `turlututu`.

Q. 11 Proposer le pseudo-code complet de l'algorithme de Boyer-Moore. Quelle est la complexité de cet algorithme en fonction de la taille du motif et de celle du texte ?

♣. et de la remarque précédente uniquement

Exercice 2 : Recherche de motif avec un automate

On fixe Σ un alphabet, et un mot $x = x_1x_2 \ldots x_m \in \Sigma^+$. Pour un mot $t \in \Sigma^\star$ de longeur $n \in \mathbb{N}$ et $j \in [\![1, n]\!]$, on dira qu'une **occurence** de x se termine à l'indice j dans t si et seulement si $x_1x_2 \ldots x_m = t_{j-m+1} \ldots t_{j-1}t_j$. Dans ce cadre là, les mots x et t ont des rôles très différents, c'est pourquoi on appellera x le **motif**, c'est-à-dire le mot dont on cherche des occurences, et on appellera **texte** un mot $y \in \Sigma^\star$ dans lequel on cherche des occurences du motif. Le but de cet exercice est de construire, à partir du motif seulement, un automate déterministe qui permette de calculer efficacement toutes les occurences du motif dans un texte.

Pour tout mot $w \in \Sigma^\star$, on note $\mathscr{P}(w)$, resp. $\mathscr{S}(w)$, l'ensemble des préfixes, resp. suffixes, de w.

$$\forall w \in \Sigma^\star,\ \mathscr{P}(w) = \{p \in \Sigma^\star \mid \exists \overline{w} \in \Sigma^\star, w = p \cdot \overline{w}\} \quad \text{et} \quad \mathscr{S}(w) = \{s \in \Sigma^\star \mid \exists \underline{w} \in \Sigma^\star, w = \underline{w} \cdot s\}$$

Q. 1 Soit $u \in \Sigma^\star$. Pour $s \in \mathscr{S}(u)$ que vaut $\mathscr{S}(s)$ en fonction de $\mathscr{S}(u)$? *Une brève justification suffit.*

Solution

On a $\mathscr{S}(s) = \mathscr{S}(u) \cap \Sigma^{\leqslant |s|}$. En effet si $w \in \mathscr{S}(s)$ il existe $\underline{s} \in \Sigma^\star$ tel que $s = \underline{s} \cdot w$. De plus comme $s \in \mathscr{S}(u)$, il existe $\underline{u} \in \Sigma^\star$ tel que $u = \underline{u} \cdot s$, donc $u = \underline{u} \cdot s = \underline{u} \cdot (\underline{s} \cdot w) = (\underline{u} \cdot \underline{s}) \cdot w)$, donc $w \in \mathscr{S}(u)$ par définition.

Q. 2 Soit $u \in \Sigma^\star$. $\mathscr{P}(u) \cap \mathscr{S}(u)$ peut-il être vide ?

Solution

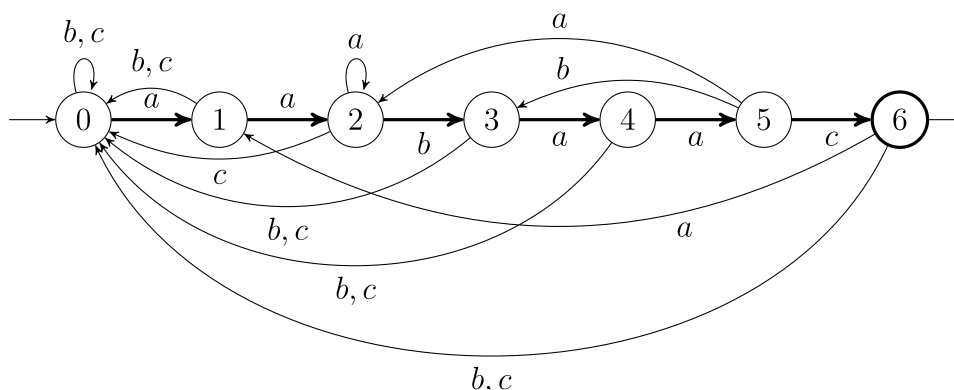
Non car cet ensemble contient nécessairement ε qui est à la fois préfixe et suffixe de tout mot.

Q. 3 Soit $u \in \Sigma^\star$. Donner un majorant de $\{|w| \mid w \in \mathscr{P}(u)\}$ et montrer qu'il est atteint (c'est donc un maximum).

Solution

$\forall w \in \mathscr{P}(u), |w| \leqslant |u|$ et $u \in \mathscr{P}(u)$ donc ce majorant est atteint.

L'automate du motif x est un automate déterministe à $m + 1$ états qui servent à représenter la progression dans le motif. Par exemple l'automate du motif $aabaac$ est le suivant.



Afin de définir formellement un tel automate pour un motif x quelconque, on introduit la fonction f qui à une longueur $i \in [\![0, m]\!]$ et une lettre $a \in \Sigma$ associe la longueur du plus long suffixe de $x_1 \ldots x_i \cdot a$

qui soit aussi un préfixe de x.

$$f = \left(\begin{array}{ccc} [\![0,m]\!]\times\Sigma & \to & [\![0,m]\!] \\ (i,a) & \mapsto & \max\limits_{w\in\mathcal{P}(x)\cap\mathcal{S}(x_1\dots x_i a)} |w| \end{array} \right)$$

Q. 4 Pour $i \in [\![0, m-1]\!]$, que vaut $f(i, x_{i+1})$?

Solution

$f(i, x_{i+1}) = i + 1$.
En effet $w^\star = x_1 \dots x_i x_{i+1}$ est à la fois un préfixe de x et un suffixe de $\mathcal{S}(x_1 \dots x_i \cdot x_{i+1})$, donc $f(i, x_{i+1}) \geqslant |w^\star| = i + 1$. Or puisque $\mathcal{S}(x_1 \dots x_i \cdot x_{i+1}) \subseteq \mathcal{S}(x_1 \dots x_i \cdot x_{i+1})$, on a d'après la question précédente $\forall w \in \mathcal{P}(x) \cap \mathcal{S}(x_1 \dots x_i \cdot x_{i+1}), |w| \leqslant |x_1 \dots x_i \cdot x_{i+1}| = i + 1$, et en passant au max sur w on en déduit $f(i, x_{i+1}) \leqslant i + 1$, d'où $f(i, x_{i+1})$.

On définit alors $\mathcal{A}_x = (\Sigma, Q = [\![0, m]\!], I = \{0\}, F = \{m\}, \delta)$ où $\delta = \{(i, a, f(i, a)) \mid i \in [\![0, m]\!], a \in \Sigma\}$

Q. 5 Pour le motif $x = babba$ sur l'alphabet $\Sigma = \{a, b\}$ (on a donc $m = 5$), dresser la table des $f(i, z)$ pour $(i, z) \in [\![0, m]\!] \times \Sigma$ puis dessiner $\mathcal{A}_x$.

Solution

i	$x_1 \dots x_i$	$f(i,a)$	$f(i,b)$
0	ε	0	1
1	b	2	1
2	ba	0	3
3	bab	2	4
4	$babb$	5	1
5	$babba$	0	3

Q. 6 Donner le pseudo-code d'un algorithme naïf construisant l'automate $\mathcal{A}_x$ à partir du motif x. On attend un algorithme naïf en $\mathcal{O}(|\Sigma| m^3)$.

Solution

Algorithme 1 : Algorithme construction de l'automate

Entrée : Un motif $x =_1 x_2 \dots x_m$, $m \in \mathbb{N}$
Sortie : L'automate $\mathcal{A}_x$

```
1  δ une matrice initialisée à 0 indicée par ⟦0,m⟧ × Σ ;
2  pour q = 0 à m faire
3  |  pour tout z ∈ Σ faire
4  |  |  k ← 1 ;
5  |  |  tant que δ[q][z] = 0 et k ⩽ q faire
6  |  |  |  si x_k x_{k+1} ... x_q z = x_1 x_2 ... x_{q-k+2} alors
7  |  |  |  |  δ[q][z] ← q − k + 2 ;
8  |  |  |  sinon
9  |  |  |  |  k ← k + 1 ;
10 |  |  si δ[q][z] = 0 et z = x_1 alors
11 |  |  |  δ[q][z] ← 1 ;
12 retourner (Σ, ⟦0,m⟧, {0}, {m}, {(q, z, δ(q,z)) | q ∈ ⟦0,m⟧, z ∈ Σ})
```

Puisque l'automate $\mathcal{A}_x$ est complet et déterministe, pout tout état $q \in Q$ et toute lettre $a \in \Sigma$, il existe un unique état $q' \in Q$ tel que $(q, a, q') \in \delta$. On note cet état $\delta^1(q, a)$, ce qui revient à noter δ^1 la fonction de transition de $\mathcal{A}_x$. De plus on note $\delta^\star$ la fonction de transition étendue de $\mathcal{A}_x$.

$$\delta^\star = \left(\begin{array}{lll} Q \times \Sigma^\star & \longrightarrow & Q \\ (q, \varepsilon) & \mapsto & q \\ (q, z \cdot u) & \mapsto & \delta^\star\Big(\delta^1(q, z), u\Big) \\ \text{où } z \in \Sigma & & \end{array} \right) = \left(\begin{array}{lll} Q \times \Sigma^\star & \longrightarrow & Q \\ (q, \varepsilon) & \mapsto & q \\ (q, u \cdot z) & \mapsto & \delta^1\Big(\delta^\star(q, u), z\Big) \\ \text{où } z \in \Sigma & & \end{array} \right)$$

On a affirmé en introduction que les états de l'automate $\mathcal{A}_x$ servent à représenter la progression dans le motif x. Cette affirmation vague peut maintenant être reformulée de manière plus précise. Pour tout mot $u \in \Sigma^\star$, sur l'automate $\mathcal{A}_x$, la lecture de u mène à un état qui indique la longueur du plus long préfixe de x qui est un suffixe de u. Avec les notations introduites plus haut, cet invariant peut être reformulé comme suit.

$$\forall u \in \Sigma^\star,\ \delta^\star(0, u) = \max_{w \in \mathcal{P}(x) \cap \mathcal{S}(u)} |w|$$

Q. 7 Démontrer la propriété précédente par récurrence sur la taille de u.

Solution

Puisque l'automate auquel on s'intéresse a un seul état initial, 0, et qu'on ne s'intéresse qu'à des éxecutions, on se permet dans cette preuve de noter, pour tout mot u, $\delta^\star(u)$ au lieu $\delta^\star(0, u)$. Pour $n \in \mathbb{N}$, on note H_n : $\forall u \in \Sigma^n,\ \delta^\star(u) = \max_{w \in \mathcal{P}(x) \cap \mathcal{S}(u)} |w|$.

• Si $u \in \Sigma^0$, $u = \varepsilon$, donc $\delta^\star(u) = \delta^\star(\varepsilon) = 0$ car $I = \{0\}$. Par ailleurs $\mathcal{S}(\varepsilon) = \{\varepsilon\}$ et $\varepsilon \in \mathcal{P}(x)$, donc $\mathcal{P}(x) \cap \mathcal{S}(u) = \{\varepsilon\}$ et $\max\limits_{w \in \mathcal{P}(x) \cap \mathcal{S}(u)} |w| = |\varepsilon| = 0$ donc H_0 est vraie.

• Soit $n \in \mathbb{N}$. On suppose H_n vraie et on cherche à montrer que H_{n+1} l'est aussi.
Soit $u \in \Sigma^{n+1}$. Il existe alors $\underline{u} \in \Sigma^n$ et $a \in \Sigma$ tels que $u = \underline{u} \cdot a$ On note alors $j = \delta^\star(\underline{u})$.
Comme $|\underline{u}| = n$, par H_n on a $j = \max_{w \in \mathcal{P}(x) \cap \mathcal{S}(\underline{u})} |w|$, et en particulier $x_1 \dots x_j \in \mathcal{S}(\underline{u})$. *(En effet il existe $w \in \mathcal{P}(x) \cap \mathcal{S}(\underline{u}) \cap \Sigma^j$ et le seul préfixe de x de taille j est $x_1 \dots x_j$, donc $w = x_1 \dots x_j \in \mathcal{S}(\underline{u})$.).*

On a aussi $\delta^\star(u) = \delta^\star(\underline{u}\cdot a) = \delta^1(\delta^\star(\underline{u}), a)$ par définition de $\delta^\star$, soit $\delta^\star(u) = \delta^1(j,a) = f(j,a)$ par définition des transitions de $\mathcal{A}_x$, soit $\delta^\star(u) = \max_{w\in\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j a)} |w|$ par définition de f. Il reste donc à montrer que $\max\limits_{w\in\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j a)} |w| = \max\limits_{w\in\mathcal{P}(x)\cap\mathcal{S}(u)} |w|$. En fait on va montrer que $\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j\cdot a) = \mathcal{P}(x)\cap\mathcal{S}(u)$ par double inclusion. D'après la question 2, et puisque $x_1\ldots x_j a\in\mathcal{S}(\underline{u}a) = \mathcal{S}(u)$, on a $\mathcal{S}(x_1\ldots x_j a) = \mathcal{S}(u)\cap\Sigma^{\leqslant j+1}$. En particulier $\mathcal{S}(x_1\ldots x_j a)\subseteq\mathcal{S}(u)$ donc $\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j\cdot a)\subseteq\mathcal{P}(x)\cap\mathcal{S}(u)$.
Réciproquement si $w\in\mathcal{P}(x)\cap\mathcal{S}(u)$, soit $w=\varepsilon$, auquel cas $w\in\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j\cdot a)$, soit $w\neq\varepsilon$, et on peut alors écrire $w = \underline{w}\cdot a$ pour un certain $\underline{w}\in\mathcal{S}(\underline{u})$. Dans ce cas, comme $\underline{w}\in\mathcal{P}(w)$ et $w\in\mathcal{P}(x)$, on a finalement $\underline{w}\in\mathcal{P}(x)\cap\mathcal{S}(\underline{u})$. Comme $|\underline{u}| = n$, on en déduit par H_n que $|\underline{w}|\leqslant\delta^\star(\underline{u}) = j$, donc $|w| = |\underline{w}|+1\leqslant j+1$, donc $w\in\mathcal{P}(x)\cap\mathcal{S}(u)\cap\Sigma^{\leqslant j+1} = \mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j a)$. D'où l'inclusion réciproque $\mathcal{P}(x)\cap\mathcal{S}(u)\subseteq\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j\cdot a)$.
On a donc bien

$$\delta^\star(u) = \max_{w\in\mathcal{P}(x)\cap\mathcal{S}(x_1\ldots x_j a)} |w| = \max_{w\in\mathcal{P}(x)\cap\mathcal{S}(u)} |w|$$

Ceci étant vrai pour u quelconque dans Σ^{n+1}, on a bien H_{n+1}. On conclut par principe de récurrence sur $\mathbb{N}$ que $\forall n\in\mathbb{N}, H_n$ est vraie.

Q. 8 Expliquer comment utiliser l'automate $\mathcal{A}_x$ pour repérer toutes les occurences de x dans un texte t de taille n ? Quelle serait la complexité temporelle de cette procédure (sans compter le pré-traitement consistant à créer $\mathcal{A}_x$) ? Quelle serait la complexité spatiale d'une telle méthode ?

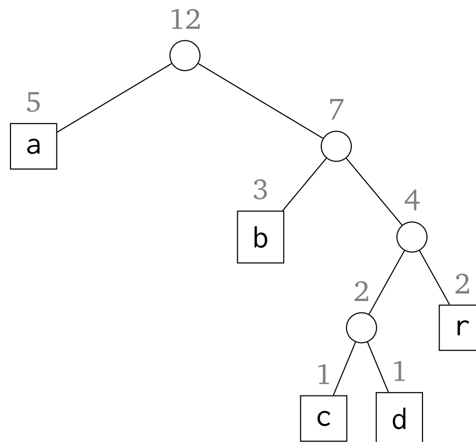
Solution

On simule la lecture du texte par l'automate et dès qu'on passe dans l'état final on a repéré la fin d'une occurence de x. L'automate $\mathcal{A}_x$ peut être représenté par sa table de transitions, qui est aussi la tables des valeurs de f, comme présenté à la question 5. Cette table étant déjà calculée, on peut accéder en $\mathcal{O}(1)$ à l'état suivant étant donné l'état courant et la lettre lue, ce qui resulte en une complexité en $\mathcal{O}(n)$ de la lecture du texte (à chaque étape, outre la transition, on teste si l'état courant est l'état final, et on incrémente un compteur, mais ces opérations sont aussi en $\mathcal{O}(1)$. La complexité spatiale est en $\mathcal{O}(m|\Sigma|)$ car c'est la taille de la table des transitions.

Exercice 3 : Algorithme de Huffman (Révisions)

Q. 1 Donner l'arbre de codage produit par l'algorithme de Huffman pour le texte `abracadabrab`. *On placera les lettres par ordre alphabétique de gauche à droite.*

Solution



Q. 2 À l'aide de l'arbre de codage précédent, compresser le texte babadada. Combien de bits compte le code obtenu ?

Solution

Le code obtenu est $10{\cdot}0{\cdot}10{\cdot}0{\cdot}1101{\cdot}0{\cdot}1101{\cdot}0$ qui est donc codé sur 16 bits.
*Remarque : Si on avait encodé ce texte avec des **char** il aurait fallut $8 \times 8 = 64$ bits. Si on avait codé chacun des 5 caractères possibles sur 3 bits (c'est le minimum si on veut un codage à taille fixe, car $2^2 < 5$), il aurait fallut $8 \times 3 = 24$ bits. Dans les deux cas le codage de Huffman s'avère plus compact.*

Exercice 4 : Algorithme de Rabin-Karp (Révisions)

Dans cet exercice on considère un alphabet $\Sigma = [\![0, B-1]\!]$ où $B \in \mathbb{N} \setminus \{0,1\}$, un entier $N \in \mathbb{N}^\star$ et une fonction de hachage $h : \Sigma^\star \to [\![0, N-1]\!]$.
Pour les exemples de mots de $\Sigma^\star$, on utilisera les lettres a, b, c, … à la place des entiers 0, 1, 2, …

Q. 1 Existe-t-il une telle fonction h qui soit injective ?

Solution

Non, $\Sigma^\star$ n'est pas fini et $[\![0, N-1]\!]$ l'est.

On se place dans le cas où $\forall w \in \Sigma^\star$,

$$h(w) = \left(\sum_{i=1}^{|w|} w_i\right) \mod N.$$

Q. 2 Donner un algorithme permettant, pour tout $a_{k-1}a_{k-2}\dots a_1a_0 \in \Sigma^*$ et $b \in \Sigma$, de calculer en $\mathcal{O}(1)$ la valeur de $h(a_{k-2}\dots a_1a_0{\cdot}b)$ connaissant celle de $h(a_{k-1}\dots a_1a_0)$.

Solution

$$h(a_{k-2}\dots a_0 \cdot b) = \left(\sum_{i=1}^{k-1} a_{k-1-i} + b\right) \mod N = \Big(h(a_{k-1}\dots a_0) - a_{k-1} + b\Big) \mod N.$$

Il suffit donc de soustraire la lettre sortante a_{k-1}, d'ajouter la lettre entrante b, puis de prendre le modulo : $\mathcal{O}(1)$.

Q. 3 Appliquer l'algorithme de Rabin-Karp pour la fonction de hachage ci-dessus et la valeur $N = 10$ sur le texte $b\,a\,c\,j$ et le motif $a\,b$.

Solution

On commence par calculer la valeur de la fonction de hachage pour le motif recherché.
$h(ab) = (0+1) \mod 10 = 1$.
Puisque le motif est de taille 2, on regarde ensuite toutes les fenêtres de taille 2 du texte à la recherche d'une fenêtre pour laquelle la fonction de hachage donne la même valeur.

Position	Fenêtre	Calcul	h	Conclusion
0	ba	$(b+a) \mod 10$	1	Faux positif : $ba \neq ab$
1	ac	$(1-b+c) \mod 10$	2	Pas candidat
2	cj	$(2-a+j) \mod 10$	1	Faux positif : $cj \neq ab$

Il n'y a aucune occurrence du motif ab dans le texte.

On se place dans le cas où $\forall w \in \Sigma^\star$,

$$h(w) = \left(\sum_{i=1}^{|w|} w_i B^{|w|-i}\right) \mod N.$$

Q. 4 Donner un algorithme permettant, pour tout $a_{k-1}a_{k-2}\dots a_1a_0 \in \Sigma^*$ et $b \in \Sigma$, de calculer en $\mathcal{O}(1)$ la valeur de $h(a_{k-2}\dots a_1a_0 \cdot b)$ connaissant celle de $h(a_{k-1}\dots a_1a_0)$.
Indication : On s'autorisera, au besoin, un pré-calcul.

Solution

$$h(a_{k-1}\dots a_0) = \sum_{i=1}^{k} a_{k-i}\, B^{k-i} = a_{k-1}B^{k-1} + a_{k-2}B^{k-2} + \dots + a_1B + a_0.$$

On observe que

$$h(a_{k-1}\dots a_0) - a_{k-1}B^{k-1} = a_{k-2}B^{k-2} + \dots + a_0 = h(a_{k-2}\dots a_0),$$

donc

$$h(a_{k-2}\dots a_0 \cdot b) = \big(h(a_{k-2}\dots a_0)\big)\cdot B + b = \Big(\big(h(a_{k-1}\dots a_0) - a_{k-1}\cdot B^{k-1}\big)\cdot B + b\Big) \mod N.$$

En pré-calulant $B^{k-1} \mod N$, le glissement s'effectue en $\mathcal{O}(1)$ puisqu'il suffit de :

1. soustraire $a_{k-1}\cdot B^{k-1} \mod N$ (lettre sortante) ;
2. multiplier par B, puis passer au modulo N ;
3. ajouter b (lettre entrante), puis passer au modulo N.

Q. 5 Appliquer l'algorithme de Rabin-Karp pour la fonction de hachage ci-dessus avec $B = 4$ et pour $N = 10$, sur le texte $c\,d\,a\,b$ et le motif $a\,b$.

Solution

On commence par calculer la valeur de la fonction de hachage pour le motif recherché.

$$h(ab) = 0\cdot B^1 + 1\cdot B^0 \mod 10 = 1.$$

On mémorise $B^{k-1} \mod N = B^1 \mod 10 = 4$.
Puisque le motif est de taille $k = 2$, on regarde ensuite toutes les fenêtres de taille 2 du texte à la

recherche d'une fenêtre pour laquelle la fonction de hachage donne la même valeur.

Position	Fenêtre	Calcul	h	Conclusion
0	cd $(2,3)$	$(2 \cdot 4 + 3) \mod 10$	1	Faux positif
1	da $(3,0)$	$((1 - c{\cdot}4){\cdot}4 + a) \mod 10$	2	—
2	ab $(0,1)$	$((2 - d{\cdot}4){\cdot}4 + b) \mod 10$	1	Vrai correspondance

Il y a donc une occurence du motif dans le texte, à la position 2.

Exercice 5 : Algorithme de Lempel-Ziv-Welsh (Révisions)

Q. 1 Rappeler quel est le principe de la compression selon LZW. Que peut-on dire des longueurs des motifs ajoutés au dictionnaire au cours de la compression ? Que peut-on dire des motifs ajoutés au dictionnaire lors de la décompression par rapport à ceux ajoutés lors de la compression ?

Q. 2 Exécuter à la main la compression de LZW sur le mot `abaaab`. *On pourra prendre comme dictionnaire de base celui qui numérote les lettres de l'alphabet minuscule de* 0 *à* 25.

Solution

```
à l'étape 1 :  [ab] -> 26
à l'étape 2 :  [ba] -> 27
à l'étape 3 :  [aa] -> 28
à l'étape 4 : on reconnaît [aa]
à l'étape 5 :  [aab] -> 29
int list = [0; 1; 0; 28; 1]
```

Q. 3 Exécuter à la main la décompression de LZW sur la chaîne obtenue à la question précédente, sans utiliser le dictionnaire construit précédemment.

Solution

```
le code 0 donne [a]
--motif [a] déjà codé
le code 1 donne [b]
--ajout de 26 -> [ab]
le code 0 donne [a]
--ajout de 27 -> [ba]
le code 28 est inconnu
on prolonge le motif courant par sa première lettre : a.a
--ajout de 28 -> [aa]
--motif [aa] déjà codé
le code 1 donne [b]
--ajout de 29 -> [aab]
- : string = "abaaab"
```

On cherche maintenant à implémenter la compression et la décompression selon LZW en OCaml. On utilise pour implémenter les dictionnaires (et les ensembles si besoin) le module `Hashtbl`. On fournit dans le fichier `compagnon_lzw.ml` plusieurs fonctions utiles, notamment de manipulation élémentaire des dictionnaires mis en jeu lors de la compression ou la décompression.

Q. 4 Coder en OCaml la compression et la décompression selon LZW. Tester ces procédures sur des textes et calculer les taux de compression observés.

Solution

```ocaml
let compression (t:string) (aff:bool) : int list =
  let res = ref [] in
  let dico = dico_minimal_motif_to_code () in
  let free = ref ((find_val_max dico)+1) in
  let motif = ref (string_of_char t.[0]) in
  for i=1 to (String.length t) -1 do
    (* if aff then Unix.sleepf 0.1; *)
    if aff then Format.printf "à l'étape %d : " i;
    let new_motif = (!motif)^(string_of_char t.[i]) in
    if Hashtbl.mem dico new_motif
    then begin
      if aff then Format.printf "on reconnaît [%s]\n" new_motif;
      motif := new_motif
    end
    else begin
      res := ( (Hashtbl.find dico (!motif))::!res );
      if aff then Format.printf " [%s] -> %d \n" (new_motif) (!free);
      Hashtbl.replace dico (new_motif) (!free);
      incr free;
      motif := (string_of_char t.[i])
      end
  done;
  res := ( (Hashtbl.find dico (!motif))::!res );
  List.rev !res
```

```ocaml
let decompression (l:int list) (aff:bool) : string  =
  let dico = dico_minimal_code_to_motif () in
  let ind_dico = Hashtbl.create (Hashtbl.length dico) in
  Hashtbl.iter (fun cle vlr -> Hashtbl.replace ind_dico vlr true) dico;
  let free = ref ((find_cle_max dico)+1) in

  let res = ref "" in
  let suffixe = ref "" in

  let rec aux (ll:int list) (k:int) : string =
    (* les k dernières lettres de !res sont à étudier              *)
    (* inv : !res contient la partie décompressée du texte codé ds l *)
    (*       et ll le reste encore compressé                       *)
    match k,ll with
    | 0,[] -> !res
    | 0, code :: q ->
      begin
        if Hashtbl.mem dico code
        then (
          let motif = Hashtbl.find dico code in
          if aff then Format.printf "le code %d donne [%s]\n" code motif;
          res := (!res) ^ motif;
```

```ocaml
        aux (q) (String.length motif)
      )
      else (
        if aff then Format.printf "le code %d est inconnu \non prolonge le
        ↪  motif courant par sa première lettre : %s.%c\n" code !suffixe
        ↪  !suffixe.[0];
        let motif = !suffixe ^ (string_of_char (!suffixe.[0])) in
        Hashtbl.replace dico code motif;
        res := (!res) ^ motif;
        aux (q) (String.length motif)
      )
    end
  | _, _ ->
    begin
      let new_letter : string = k_last (!res) k in
      let new_suffixe = (!suffixe) ^ new_letter in
      if Hashtbl.mem ind_dico new_suffixe
      then
        begin
          if aff then Format.printf "--motif [%s] déjà codé\n"
          ↪  new_suffixe;
          suffixe := new_suffixe;
        end
      else
        begin
          if aff then Format.printf "--ajout de %d -> [%s] \n" (!free)
          ↪  new_suffixe;
          Hashtbl.replace dico (!free) new_suffixe;
          incr free;
          Hashtbl.replace ind_dico new_suffixe true;
          suffixe := new_letter;
        end;
      aux ll (k-1)
    end

in aux l 0
```

Q. 5 Citer un autre algorithme de compression (au programme). En quoi ces deux méthodes de compression diffèrent-elles ?

Solution

On peut citer la compression à l'aide d'un arbre de codage, un arbre de codage offrant le meilleur taux de compression est fourni par l'algorithme de Huffmann.
Dans le cas d'une compression à l'aide d'un arbre de codage, l'arbre de codage doit être transmis avec le code en vue de la décompression, tandis qu'avec l'algorithme LZW on n'a pas besoin d'encoyer le dictionnaire qui associe aux motif leurs code, le code du texte suffit.

Éléments pour établir un algorithme de programmation dynamique

→ identifier une famille de "sous-problèmes"[a] qu'il est intéressant de résoudre ;
→ identifier quels paramètres permettent de caractériser ces sous-problèmes relativement à l'instance de départ ;
→ définir une quantité paramétrée qui représente la valeur optimale du sous-problème défini par ces paramètres ;
→ justifier que cette famille de quantités permet de résoudre le problème initial ;
→ dire comment calculer les cas de base, *i.e.* les quantités pour les paramètres minimaux ;
→ dire comment calculer l'une de ces quantités à partir des valeurs pour des paramètres plus petits ;
→ si on opte pour de l'impératif, fournir le pseudo-code qui explique quelle taille de tableau alouer pour stocker ces quantités, dans quel ordre on va remplir le tableau, comment on va ensuite récupérer la valeur optimale ;
→ dans le cas où seule la valeur optimale nous intéresse, il est souvent possible de modifier l'algorithme précédent en vue de réduire sa consommation mémoire ;
→ dans le cas où une solution optimale doit aussi être fournie, on complète l'algorithme précédent en ajoutant le plus souvent l'enregistrement d'information pertinente lors du remplissage du tableau, et en ajoutant une phase de reconstruction de la solution qui remonte dans le tableau à partir d'une case de valeur optimale.

a. On appelle ici, comme c'est l'usage, famille de sous-problèmes une famille d'instances, instances du problème initial ou parfois d'une variante de celui-ci.

Exercice 6 : Découpe de période

On suppose qu'on cherche à découper période de L jours en sessions de manière à maximiser l'utilité du découpage. La durée de chaque session est un nombre de jours entier choisi parmi un ensemble de durées possibles. À chaque durée de session possible est associée une utilité. L'utilité du découpage est la somme des utilités des sessions (peu importe l'ordre).

Q. 1 Formaliser ce problème concret comme un problème d'optimisation nommé DÉCOUPE.

Solution

AEF : à compléter

Q. 2 Donner une relation de récurence permettant de résoudre le problème DÉCOUPE.

Q. 3 Donner le pseudo-code d'un algorithme de programmation dynamique permettant de calculer la valeur optimale d'une instance de DÉCOUPE.

Q. 4 Donner le pseudo-code d'un algorithme de programmation dynamique permettant de calculer une solution optimale d'une instance de DÉCOUPE.

Q. 5 Quelles sont les complexités spatiales et temporelles des deux algorithmes précédents ?

Exercice 7 : Plus long sous-mot/facteur commun

Q. 1 Rappeler les définitions de **facteur** d'un mot et **sous-mot** d'un mot.
Donner un exemple illustrant la différence entre les deux.

Solution

On dit que y est un **sous-mot** de x ssi il existe φ une fonction strictement croissante de $[\![1, |y|]\!]$ dans $[\![1, |x|]\!]$ telle que $y = x_{\varphi(1)}x_{\varphi(2)}\dots x_{\varphi(m)}$.

On dit que y est un **facteur** de x ssi il existe deux mots u et v sur Σ tels que $x = u \cdot y \cdot v$.

jrd est un sous-mot de $jardin$ mais pas un facteur.

Q. 2 Définir le problème du plus long sous-mot commun (PLSMC).

Solution

$$\text{PLSMC}\begin{cases}\textbf{Entrée :}\ x \in \Sigma^\star,\ y \in \Sigma^\star \\ \textbf{Sortie :}\ \max\{\,|u| \mid u \in \Sigma^\star, u \text{ est sous-mot de } x \text{ et de } y\}\end{cases}$$

Q. 3 Définir le problème du plus long facteur commun (PLFC).

Solution

$$\text{PLFC}\begin{cases}\textbf{Entrée :}\ x \in \Sigma^\star,\ y \in \Sigma^\star \\ \textbf{Sortie :}\ \max\{\,|u| \mid u \in \Sigma^\star, u \text{ est facteur de } x \text{ et de } y\}\end{cases}$$

Q. 4 Expliquer comment résoudre le problème PLSMC par programmation dynamique.

Solution

Soit $(x,y)\in\Sigma^\star\times\Sigma^\star$ une instance de PLSMC. On note $n=|x|$ et $m=|x|$.

On pose, pour tout $(i,j)\in[\![0,n]\!]\times[\![0,m]\!]$:

$$\mathcal{E}_{i,j}\stackrel{\text{déf}}{=}\{\,u\,|\,u\in\Sigma^\star, u \text{ est sous-mot de } x_1\dots x_i \text{ et de } y_1\dots y_j\}$$
$$B_{i,j}\stackrel{\text{déf}}{=}\max\{|u|\,|\,u\in\mathcal{E}_{i,j}\}$$

La valeur optimale (*i.e.* la solution de PLSMC pour l'instance (x,y)) est alors donnée par $B_{n,m}$.

Le calcul des valeurs $B_{i,j}$ peut se faire grâce aux relations suivantes.

$$\forall i\in[\![0,n]\!], B_{i,0}=0 \qquad \text{car } y_1\dots y_0=\varepsilon \text{ n'a que } \varepsilon \text{ comme sous-mot, et } |\varepsilon|=0$$
$$\forall j\in[\![0,m]\!], B_{0,j}=0 \qquad \text{idem}$$
$$\forall (i,j)\in[\![1,n]\!]\times[\![1,m]\!], B_{i,j}=\begin{cases}1+B_{i-1,j-1} & \text{si } x_i=y_j\\ \max(B_{i-1,j},B_{i,j-1}) & \text{sinon}\end{cases} \qquad (\bigstar)$$

Justifions la relation de récurrence ($\bigstar$) par disjonction de cas. Soit (i,j) $\in[\![1,n]\!]\times[\![1,m]\!]$.

- Cas $x_i=y_j$.
  - Si $v\in\mathcal{E}_{i-1,j-1}$, alors $v\cdot x_i\in\mathcal{E}_{i,j}$, donc $B_{i,j}\geqslant|v\cdot x_i|=1+|v|$. Cette relation pour $v^*\in\mathcal{E}_{i-1,j-1}$ optimal, donne en particulier $B_{i,j}\geqslant 1+B_{i-1,j-1}$.
  - Si $u^*\in\mathcal{E}_{i,j}$ est optimal, nécéssairement $|u^*|\geqslant 1$ car $x_i\in\mathcal{E}_{i,j}$. On peut donc parler de la dernière lettre de u^* et décomposer $u^*=v\cdot z$ avec $v\in\Sigma^\star$ et $z\in\Sigma$. Si $z\neq x_i$, alors u^* serait sous-mot de $x_1\dots x_{i-1}$ et de $y_1\dots y_{j-1}$, et alors $u^*\cdot x_i=u^*\cdot y_j$ serait un sous-mot de $x_1\dots x_i$ et de $y_1\dots y_j$ de longueur $|u^*|+1$ soit $B_{i,j}+1$. ABSURDE. Ainsi $u^*=v\cdot x_i$ et $v\in\mathcal{E}_{i-1,j-1}$, donc $B_{i,j}=|u^*|=|v|+1\leqslant B_{i-1,j-1}+1$.

  Par double inégalité on a bien $B_{i,j}=1+B_{i-1,j-1}$ dans ce cas.
- Cas $x_i\neq y_j$.
  - Puisqu'un sous-mot d'un sous-mot est un sous-mot, $\mathcal{E}_{i,j-1}\subseteq\mathcal{E}_{i,j}$ donc $B_{i,j}\geqslant B_{i,j-1}$. De même $B_{i,j}\geqslant B_{i-1,j}$, d'où $B_{i,j}\geqslant\max(B_{i-1,j},B_{i,j-1})$.
  - Soit $u^*\in\mathcal{E}_{i,j}$ optimal. Si $u^*=\varepsilon$, $u^*\in\mathcal{E}_{i,j-1}$, donc $B_{i,j}=|u^*|\leqslant B_{i,j-1}\leqslant\max(B_{i-1,j},B_{i,j-1})$.
    Sinon, $u^*=v\cdot z$ avec $v\in\Sigma^\star$ et $z\in\Sigma$. Puisque $x_i\neq y_j$, on a $z\neq x_i$ ou $z\neq y_j$.
    Si $z\neq x_i$, alors $u^*\in\mathcal{E}_{i-1,j}$, et donc $B_{i,j}=|u^*|\leqslant B_{i-1,j}\leqslant\max(B_{i-1,j},B_{i,j-1})$.
    De même, si $z\neq y_j$ on a $B_{i,j}=|u^*|\leqslant B_{i,j-1}\leqslant\max(B_{i-1,j},B_{i,j-1})$.
    Ainsi on a bien $B_{i,j}\leqslant\max(B_{i-1,j},B_{i,j-1})$ dès lors que $x_i\neq y_j$.

  Par double inégalité on a bien $B_{i,j}=\max(B_{i-1,j},B_{i,j-1})$ dans ce cas.

On en déduit l'algorithme de programmation dynamique suivant.

Algorithme 2 : Algorithme pour PLSMC (valeur/solution optimale)

```
   Entrée : Deux mots de longueurs respectives n et m : x = x_1 ... x_n et y = y_1 ... y_m
   Sortie : La longueur d'un plus long sous-mot commun à x et y ou un tel sous-mot
 1 B ← tableau d'entiers indexé par ⟦0, n⟧ × ⟦0, m⟧ ;
 2 M ← tableau de couples d'entiers indexé par ⟦0, n⟧ × ⟦0, m⟧ ;
 3 pour i = 0 à n faire
 4 |   B[i][0] ← 0 ;
 5 |   M[i][0] ← (i − 1, 0) ;
 6 pour j = 0 à m faire
 7 |   B[0][j] ← 0 ;
 8 |   M[0][j] ← (0, j − 1) ;
 9 pour i = 1 à n faire
10 |   pour j = 1 à m faire
11 |   |   si x_i = y_j alors
12 |   |   |   B[i][j] ← 1 + B[i − 1][j − 1] ;
13 |   |   |   M[i][j] ← (i − 1, j − 1) ;
14 |   |   sinon
15 |   |   |   si B[i − 1][j] > B[i][j − 1] alors
16 |   |   |   |   B[i][j] ← B[i − 1][j] ;
17 |   |   |   |   M[i][j] ← (i − 1, j) ;
18 |   |   |   sinon
19 |   |   |   |   B[i][j] ← B[i][j − 1] ;
20 |   |   |   |   M[i][j] ← (i, j − 1) ;
21 retourner B[n][m] ;
22 (i, j) ← (n, m) ;
23 u ← ε ;
24 tant que i > 0 et j > 0 faire
25 |   si M[i][j] = (i − 1, j − 1) alors
26 |   |   u ← x_i · u
27 |   (i, j) ← M[i][j] ;
28 retourner u ;
```

Justifions la correction de l'algorithme 2 Les trois premières boucles remplissent le tableau B avec les valeurs de B grâce aux relations précédemment démontrées, autrement dit à la ligne 21 on a $\forall (i,j) \in [\![0,n]\!] \times [\![0,m]\!], \mathsf{B}[i][j] = B_{i,j}$. Ainsi la version de l'algorithme qui calcule la valeur est correct.

Pour montrer que l'algorithme qui calcule une solution est correct, on s'appuye sur l'invariant suivant pour la boucle ligne 24.

$$u \text{ est sous-mot de } x_{i+1} \dots x_n \text{ et de } y_{j+1} \dots y_m \text{ et } |u| = \mathsf{B}[n][m] - \mathsf{B}[i][j]$$

- Initialement $(i,j) = (n,m)$ donc $x_{i+1} \dots x_n = \varepsilon$ et de $y_{j+1} \dots y_m = \varepsilon$, ainsi $u = \varepsilon$ est bien un sous-mot de ces deux mots. De plus, on a alors $\mathsf{B}[n][m] - \mathsf{B}[i][j] = \mathsf{B}[n][m] - \mathsf{B}[n][m]$ et $|u| = |\varepsilon| = 0$. L'invariant proposé est donc vrai avant la boucle ligne 24.

- Supposons l'invariant vérifié à l'entrée d'un tour de boucle, montrons qu'il l'est toujours en sortie de ce tour. On note i^a (resp. j^a et u^a) la valeur de la variable i (resp. j et u) à l'entrée du tour de boucle. On note i^b (resp. j^b et u^b) les valeurs en sortie.
  - Si $\mathsf{M}[i^a][j^a] = (i^a - 1, j^a - 1)$, alors on a $i^b = i^a - 1$ et $j^b = j^a - 1$ d'après la l.27, et $u^b = x_{i^a} \cdot u^a$ d'après la l.**??**. De plus, vu comment est construit M entre les lignes 3 et 21, on a nécessairement $x_{i^a} = y_{j^a}$ et $\mathsf{B}[i^a][j^a] = 1 + \mathsf{B}[i^a - 1][j^a - 1]$.
  On a alors
  $$\begin{aligned} |u^b| &= 1 + |u^a| \\ &= 1 + (\mathsf{B}[n][m] - \mathsf{B}[i^a][j^a]) \qquad \text{par hypothèse} \\ &= \not{1} + \mathsf{B}[n][m] - \left(\not{1} + \mathsf{B}[i^a - 1][j^a - 1]\right) \\ &= \mathsf{B}[n][m] - \mathsf{B}[\underbrace{i^a - 1}_{i^b}][\underbrace{j^a - 1}_{j^b} \end{aligned}$$
  et comme u^a est sous-mot de $x_{i^a+1} \dots x_n$ par hypothèse, $u^b = x_{i^a} \cdot u^a$ est sous-mot de $x_{i^a} \dots x_n$ soit de $x_{i^b+1} \dots x_n$. De même $u^b = y_{j^a} \cdot u^a$ est sous-mot de $y_{j^a} \dots y_m$ soit de $y_{j^b+1} \dots y_m$. Ainsi les valeurs i^b,j^b et u^b vérifient bien l'invariant dans ce cas.
  - Sinon, on a $u^b = u^a$, or par hypothèse u^a est un sous-mot de $x_{i^a+1} \dots x_n$ et $y_{j^a+1} \dots y_m$, donc u^b aussi, et a fortiori u^b est sous-mot de $x_{i^a} \dots x_n$ et $y_{j^a} \dots y_m$.
  Vu comment est construit M entre les lignes 3 et 21 on a deux sous-cas possibles.
    - Si $\mathsf{M}[i^a][j^a] = (i^a - 1, j^a)$, on a aussi $\mathsf{B}[i^a][j^a] = \mathsf{B}[i^a - 1][j^a]$ par construction, et d'après la ligne 27, $i^b = i^a - 1$, $j^b = j^a$.
    - Si $\mathsf{M}[i^a][j^a] = (i^a, j^a - 1)$, on a aussi $\mathsf{B}[i^a][j^a] = \mathsf{B}[i^a][j^a - 1]$ par construction, et d'après la ligne 27, $i^b = i^a$, $j^b = j^a - 1$.

  Dans les deux sous-cas $\mathsf{B}[n][m] - \mathsf{B}[i^b][j^b] = \mathsf{B}[n][m] - \mathsf{B}[i^a][j^a]$, or par hypothèse $\mathsf{B}[n][m] - \mathsf{B}[i^a][j^a] = |u^a|$, et comme $u^b = u^a$ on a bien $\mathsf{B}[n][m] - \mathsf{B}[i^b][j^b] = |u^b|$.
  Ainsi les valeurs i^b,j^b et u^b vérifient bien l'invariant dans ce cas.

En sortant de la boucle ligne 24, on a $i = 0$ et $j = 0$, l'invariant nous assure donc que :
- u est sous-mot de $x_{0+1} \dots x_n$ et de $y_{0+1} \dots y_m$, soit sous mot de x et y ;,
- $|u| = \mathsf{B}[n][m] - \mathsf{B}[0][0]$, or on sait que $\mathsf{B}[n][m] = B_{n,m}$ depuis la ligne 21 et $\mathsf{B}[0][0] = 0$ (*Cf.* ligne 4), donc $|u| = B_{n,m}$.

Ces deux points assurent que u, le mot renvoyé, est bien un plus long sous-mot commun à x et y.

Q. 5 Expliquer comment résoudre le problème PLFC par programmation dynamique.

Solution

Soit $(x, y) \in \Sigma^\star \times \Sigma^\star$ une instance de PLFC. On note $n = |x|$ et $m = |x|$.

On pose, pour tout $(i, j) \in [\![0, n]\!] \times [\![0, m]\!]$:

$$\mathcal{S}_{i,j} \overset{\text{déf}}{=} \{\, u \mid u \in \Sigma^\star, u \text{ est suffixe de } x_1 \dots x_i \text{ et de } y_1 \dots y_j \}$$
$$A_{i,j} \overset{\text{déf}}{=} \max\{|u| \mid u \in \mathcal{S}_{i,j}\}$$

La valeur optimale (*i.e.* la solution de PLFC pour l'instance (x, y)) est alors donnée par $\max\{A_{i,j} \mid (i, j) \in [\![0, n]\!] \times [\![0, m]\!]\}$.
En effet, un facteur est toujours un suffixe d'un préfixe, ainsi un facteur de x est un suffixe de $x_1 \dots x_i$ pour un certain $i \in [\![0, n]\!]$, et de même un facteur de y est un suffixe de $y_1 \dots y_j$ pour

un certain $j \in [\![0, m]\!]$, donc on a

$$\begin{aligned}
\textsc{plfc}(x, y) &= \max \left\{ |u| \,\middle|\, u \in \Sigma^\star, u \text{ est facteur de } x \text{ et de } y \right\} \\
&= \max \left\{ |u| \,\middle|\, u \in \Sigma^\star, \exists (i,j) \in [\![0, n]\!] \times [\![0, m]\!], u \text{ est suffixe de } x_1 \dots x_i \text{ et de } y_1 \dots y_j \right\} \\
&= \max \left\{ |u| \,\middle|\, u \in \bigcup_{(i,j) \in [\![0,n]\!] \times [\![0,m]\!]} \mathcal{S}_{i,j} \right\} \\
&= \max_{(i,j) \in [\![0,n]\!] \times [\![0,m]\!]} \max \left\{ |u| \,\middle|\, u \in \mathcal{S}_{i,j} \right\} \\
&= \max_{(i,j) \in [\![0,n]\!] \times [\![0,m]\!]} A_{i,j}
\end{aligned}$$

Le calcul des valeurs $A_{i,j}$ peut se faire grâce aux relations suivantes.

$$\forall i \in [\![0, n]\!], A_{i,0} = 0 \qquad \text{car } y_1 \dots y_0 = \varepsilon \text{ n'a que } \varepsilon \text{ comme suffixe, et } |\varepsilon| = 0$$

$$\forall j \in [\![0, m]\!], A_{0,j} = 0 \qquad \text{idem}$$

$$\forall (i,j) \in [\![1, n]\!] \times [\![1, m]\!], A_{i,j} = \begin{cases} 1 + A_{i-1,j-1} & \text{si } x_i = y_j \\ 0 & \text{sinon} \end{cases} \qquad (\bigstar)$$

Justifions la relation de récurrence ($\bigstar$) par disjonction de cas. Soit (i,j) $\in [\![1, n]\!] \times [\![1, m]\!]$.

- Si $x_i \neq y_j$ le seul suffixe commun à $x_1 \dots x_i$ et $y_1 \dots y_j$ est ε, car un suffixe ayant au moins une lettre aurait pour dernière lettre x_i en tant que suffixe de $x_1 \dots x_i$ et pour dernière lettre y_j en tant que suffixe de $y_1 \dots y_j$, on aurait alors $x_i = y_j$. ABSURDE.
- Si $x_i = y_j$ on a alors :

$$\begin{aligned}
\mathcal{S}_{i,j} &= \{\varepsilon\} \cup \left\{ v \cdot x_i \,\middle|\, v \in \Sigma^\star, \begin{bmatrix} v \cdot x_i \text{ est suffixe de } x_1 \dots x_{i-1} x_i \\ v \cdot y_j \text{ est suffixe de } y_1 \dots y_{j-1} y_j \end{bmatrix} \right\} \\
&= \{\varepsilon\} \cup \left\{ v \cdot x_i \,\middle|\, v \in \Sigma^\star, \begin{bmatrix} v \text{ est suffixe de } x_1 \dots x_{i-1} \\ v \text{ est suffixe de } y_1 \dots y_{j-1} \end{bmatrix} \right\} \\
&= \{\varepsilon\} \cup \mathcal{S}_{i-1,j-1} \cdot \{x_i\}
\end{aligned}$$

$$\begin{aligned}
\text{donc } A_{i,j} &= \max \left(|\varepsilon|, \max \left\{ \underbrace{|v \cdot x_i|}_{|v|+1} \mid v \in \mathcal{S}_{i-1,j-1} \right\} \right) \\
&= \max \left(0, 1 + \max\{|v| \mid v \in \mathcal{S}_{i-1,j-1}\} \right) \\
&= \max \left(0, \underbrace{1 + A_{i-1,j-1}}_{\geqslant 1} \right) \\
&= 1 + A_{i-1,j-1}
\end{aligned}$$

On peut aussi justifier le cas $x_i = y_j$ par double inégalité comme suit.

Soit $u^* \in \mathcal{S}_{i,j}$ tel que $|u^*| = A_{i,j}$. On sait que $u^* \neq \varepsilon$ car le mot x_i est un suffixe de $x_1 \dots x_i$ et de $y_1 \dots y_j$, strictement plus long que ε. Ainsi u^* s'écrit $v \cdot z$ avec $v \in \Sigma^\star$ et $z \in \Sigma$. Puisque u^* est suffixe de $x_1 \dots x_i$ (resp. de $y_1 \dots y_j$), on a $z = x_i$ (resp. $z = y_j$) et v est un suffixe de $x_1 \dots x_{i-1}$ (resp. de $y_1 \dots y_{j-1}$). Ainsi $|v| \leqslant A_{i-1,j-1}$, or $|v| = |u^*| - 1 = A_{i,j} - 1$, donc $A_{i,j} \leqslant 1 + A_{i-1,j-1}$.

Réciproquement, si on considère un mot $v \in \mathcal{S}_{i-1,j-1}$ tel que $|v| = A_{i-1,j-1}$, le mot $v \cdot x_i$, qui s'écrit aussi $v \cdot y_j$, est un suffixe de $x_1 \dots x_{i-1} \cdot x_i$ et de $y_1 \dots y_{j-1} \cdot y_j$, de longueur $1 + A_{i-1,j-1}$. Cela assure que $1 + A_{i-1,j-1} \leqslant A_{i,j}$.

Par double inégalité, on a bien $A_{i,j} = 1 + A_{i-1,j-1}$ dans le cas où $x_i = y_j$.

On en déduit l'algorithme de programmation dynamique suivant.

Algorithme 3 : Algorithme pour PLFC (valeur/solution optimale)

Entrée : Deux mots de longueurs respectives n et m : $x = x_1 \dots x_n$ et $y = y_1 \dots y_m$
Sortie : La longueur d'un plus long facteur commun à x et y ou un tel facteur

```
1  A ← tableau d'entiers indexé par ⟦0, n⟧ × ⟦0, m⟧ ;
2  pour i = 0 à n faire
3  |  A[i][0] ← 0 ;
4  pour j = 0 à m faire
5  |  A[0][j] ← 0 ;
6  pour i = 1 à n faire
7  |  pour j = 1 à m faire
8  |  |  si x_i = y_j alors
9  |  |  |  A[i][j] ← 1 + A[i − 1][j − 1] ;
10 |  |  sinon
11 |  |  |  A[i][j] ← 0 ;
12 max ← 0 ;
13 i_max ← 0 ;
14 pour i = 1 à n faire
15 |  pour j = 1 à m faire
16 |  |  si A[i][j] > max alors
17 |  |  |  max ← A[i][j] ;
18 |  |  |  i_max ← i ;
19 retourner max ;
20 f ← i_max ; d ← (f − max) + 1 ;
21 retourner x_d ... x_f ;
```

Exercice 8 : Accessibilité dans un graphe non orienté

Soit $G = (S, A)$ un graphe non orienté. On note $n = |S|$ et on suppose que $S = [\![1, n]\!]$. De plus on note $m = |A|$. Pour $(u, v) \in S^2$, on dit que u et v sont **reliés** dans G, ce qu'on note $u \sim v$ s'il existe une chaîne entre u et v dans G. On cherche à calculer la relation $\sim$, c'est-à-dire à déterminer pour chaque paire de sommets s'ils sont reliés par une chaîne du graphe. On cherche à calculer cette relation sous la forme d'une matrice de booléens $(T[u][v])_{(u,v)\in S^2}$ indiquant si $u \sim v$.

Sommets intérieurs Si $(\gamma_k)_{k\in[\![0,l]\!]}$ est une chaîne élémentaire♣ de G, on note $I(\gamma)$ l'ensemble des sommets intérieurs de γ, *i.e.* les sommets de γ qui n'en sont pas les extrémités.

$$I(\gamma) \overset{\text{déf}}{=} \{\gamma_k \mid k \in [\![1, l-1]\!]\}$$

On remarque qu'une chaîne ayant seulement 0 ou 1 arête ne possède aucun sommet intérieur, autrement dit si $l \leqslant 1$, alors $I(\gamma) = \emptyset$ et la réciproque est vraie.

♣. *i.e.* ne passant pas deux fois par un même sommet

Chaînes élémentaires dans les sous-graphes. Pour tout $(u, v) \in S^2$, on note $\mathcal{C}_{u,v}$ l'ensemble des chaînes élémentaires de u à v. De plus, pour tout $k \in [\![0, n]\!]$, on définit $\mathcal{C}_{u,v}^{(k)}$ l'ensemble des chaînes élémentaires, de u à v, dont tous les sommets intérieurs sont dans $[\![1, k]\!]$.

$$\mathcal{C}_{u,v}^{(k)} \stackrel{\text{déf}}{=} \{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1, k]\!]\}.$$

Famille de relations $\sim_k$. On définit finalement, pour $k \in [\![0, n]\!]$, la relation binaire $\sim_k$ sur S comme suit.

$$\forall (u, v) \in S^2, u \sim_k v \stackrel{\text{déf}}{\Leftrightarrow} \mathcal{C}_{u,v}^{(k)} \neq \emptyset$$

Autrement dit, $u \sim_k v$ dès lors qu'il existe une chaîne de u à v n'utilisant, pour sommet intérieurs, que des sommets de $[\![1, k]\!]$.

Q. 1 Démontrer ou réfuter la proposition suivante : $\forall k \in [\![1, n]\!]$, la relation $\sim_k$ est une relation d'équivalence.

Solution

En général cette affirmation est fausse car la relation $\sim_k$ n'est pas toujours transitive. En effet si le graphe contient seulement les chaînes $1, 2, 6$ et $6, 3, 5$, on a à la fois $1 \sim_3 6$ et $6 \sim_3 5$, mais pas $1 \sim_3 5$ car la chaîne $1, 2, 6, 3, 5$ a pour sommet intérieur $6 > 3$.

Q. 2 Justifier en quoi le calcul des relations $(\sim_k)_{k \in [\![0,n]\!]}$ permet de répondre au problème du calcul de $\sim$.

Solution

$\sim_n = \sim$.

On cherche à établir des relations de récurrence permettant le calcul des $(\sim_k)_{k \in [\![0,n]\!]}$.

Q. 3 Que vaut $\sim_0$?

Solution

Pour tout $(u, v) \in S^2$,

$$\begin{aligned}
u \sim_0 v &\Leftrightarrow \text{il existe une chaîne élémentaire de } u \text{ à } v \text{ dont les sommets intérieurs sont dans } [\![1, 0]\!] \\
&\Leftrightarrow \text{il existe une chaîne élémentaire de } u \text{ à } v \text{ dont les sommets intérieurs sont dans } \emptyset \\
&\Leftrightarrow \text{il existe une chaîne élémentaire de } u \text{ à } v \text{ sans sommet intérieur} \\
&\Leftrightarrow \text{il existe une chaîne élémentaire de } u \text{ à } v \text{ de longueur 0 ou 1} \\
&\Leftrightarrow u = v \text{ ou } \{u, v\} \in A
\end{aligned}$$

Q. 4 Soient $u \in S$, $v \in S$, $k \in [\![1, n]\!]$. Proposer et démontrer une relation entre $u \sim_k v$ et $u \sim_{k-1} k$, $k \sim_{k-1} v$ et $u \sim_{k-1} v$.

Solution

$u \sim_k v = (u \sim_{k-1} v) \vee (u \sim_{k-1} k \wedge k \sim_{k-1} v)$.

En effet :

$$
\begin{aligned}
u \sim_k v &\Leftrightarrow \mathcal{C}_{u,v}^{(k)} \neq \emptyset \\
&\Leftrightarrow \{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1,k]\!]\} \neq \emptyset \\
&\Leftrightarrow (\{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1,k-1]\!]\} \\
&\quad \cup \{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1,k]\!] \wedge \exists i \in [\![1,|\gamma|-1]\!], \gamma_i = k\}) \neq \emptyset \\
&\Leftrightarrow \{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1,k-1]\!]\} \neq \emptyset \\
&\quad \vee \{\gamma \in \mathcal{C}_{u,v} \mid I(\gamma) \subseteq [\![1,k]\!] \wedge \exists!^{\clubsuit} i \in [\![1,|\gamma|-1]\!], \gamma_i = k\} \neq \emptyset \\
&\Leftrightarrow \mathcal{C}_{u,v}^{(k-1)} \neq \emptyset \\
&\quad \vee \{\gamma \in \mathcal{C}_{u,v} \mid \exists \gamma_1 \in \mathcal{C}_{u,k}^{(k-1)}, \exists \gamma_1 \in \mathcal{C}_{k,v}^{(k-1)}, \gamma = \gamma_1 \cdot \gamma_2\} \neq \emptyset \\
&\Leftrightarrow \mathcal{C}_{u,v}^{(k-1)} \neq \emptyset \\
&\quad \vee \mathcal{C}_{u,k}^{(k-1)} \cdot \mathcal{C}_{k,v}^{(k-1)} \neq \emptyset \\
&\Leftrightarrow \mathcal{C}_{u,v}^{(k-1)} \neq \emptyset \\
&\quad \vee (\mathcal{C}_{u,k}^{(k-1)} \neq \emptyset \wedge \mathcal{C}_{k,v}^{(k-1)} \neq \emptyset)
\end{aligned}
$$

Si le sommet k apparaît dans γ, il apparaît une seule fois parce que les chaines considérées sont élémentaires

Q. 5 Quel problème pose l'implémentation d'une telle relation de récurrence ?

Solution

Explosion combinatoire en $\mathcal{O}(3^n)$ alors qu'il suffit de calculer $\Theta(n^3)$ grandeurs que sont les $(u \sim_k v)$ pour $(u, v, k) \in S \times S \times [\![1, n]\!]$.

Dans les algorithmes qui suivent, la famille de relations $(\sim_k)_{k \in [\![0,n]\!]}$ sera représentée par un tableau tri-dimensionnel T indicé par $[\![0, n]\!] \times S^2$ contenant des booléens :

$$\forall k \in [\![0, n]\!], \forall (u, v) \in S^2, T[k][u][v] = \mathsf{V} \text{ si et seulement si } u \sim_k v$$

Q. 6 Proposer un algorithme permettant de calculer $\sim$, tout en évitant le problème soulevé en **Q. 5**.

Solution

Entrée : Un graphe non orienté $G = (S, A)$
Sortie : La relation d'accessibilité dans G, $\sim_G$, représentée par un tableau de booléens indicé par $S \times S$

```
1  On initialise R un tableau bi-dimensionnel indicé par S × S initialisé à F ;
2  On initialise Q un tableau bi-dimensionnel indicé par S × S initialisé à F ;
3  pour i = 1 à n faire
4      pour j = 1 à n faire
5          R[i][j] ← i = j ou {i, j} ∈ A ;
6  pour k = 1 à n faire
7      pour i = 1 à n faire
8          pour j = 1 à n faire
9              Q[i][j] ← R[i][j] ∨ (R[i][k] ∧ R[k][j]) ;
10     R ← Q ;
11 retourner R ;
```

Q. 7 Dans le cas où la complexité spatiale de l'algorithme proposé n'est pas en $\mathcal{O}(n^2)$, proposer une amélioration permettant d'atteindre une telle complexité spatiale.

Solution

Voir solution ci-dessus. On remarque que $\sim_k$ ne dépend que de $\sim_{k-1}$, aussi il n'est pas nécessaire de garder en mémoire tous les $\sim_i$ pour calculer $\sim_n$.

Q. 8 De quel algorithme au programme de MP2I l'algorithme 6 est-il proche ?

Solution

Floyd-Warshall. La seule différence est le fait qu'on s'intéresse ici à l'existence de chaîne et pas à leur longueur. Ainsi si on projette $\mathbb{R}^+$ sur V et $+\infty$ sur F l'exécution de Floyd-Warshall rejoint celle de cet algorithme.

Q. 9 Au moyen d'un parcours de graphe, proposer une solution algorithmique alternative au problème du calcul de $\sim$. Préciser la complexité temporelle de cette solution.

Solution

Le partitionnement associé au découpage sur les points de régénération d'un parcours de graphe fournit la décomposition en composante connexe du graphe, qui est exactement ce qu'on essaie de calculer ici. On obtient alors une complexité en $\mathcal{O}(n + m)$ pour le parcours, puis en $\mathcal{O}(n^2)$ pour l'allocation et le remplissage de la matrice résultat.
Ce qui est bien meilleur que la complexité obtenue dans cet exercice.

Exercice 9 : Suites récurrentes de complexité

Dans cet exercice il vous est demandé de donner, pour chacune des suites suivantes :

- un Θ décrivant son comportement asymptotique ;

- un programme OCaml **dont cette suite est la complexité**, pour des constantes a et b au choix[♣] ;
- un schéma de l'arbre des appels récursifs d'un appel au programme proposé.

a) $\begin{cases} u_0 &= 1 \\ u_n &= u_{n-1} + a \quad a \in \mathbb{R}^{+\star} \end{cases}$

b) $\begin{cases} u_0 &= 1 \\ u_n &= u_{n-1} + an \quad a \in \mathbb{R}^{+\star} \end{cases}$

c) $\begin{cases} u_0 &= 1 \\ u_n &= au_{n-1} + b \quad a \in [2, +\infty[, b \in \mathbb{R}^{+\star} \end{cases}$

d) $\begin{cases} u_0 &= 1 \\ u_n &= u_{n/2} + b \quad b \in \mathbb{R}^{+\star} \end{cases}$

e) $\begin{cases} u_0 &= 1 \\ u_n &= u_{n/2} + bn \quad b \in \mathbb{R}^{+\star} \end{cases}$

f) $\begin{cases} u_0 &= 1 \\ u_1 &= 1 \\ u_n &= u_{\lceil \frac{n}{2} \rceil} + u_{\lfloor \frac{n}{2} \rfloor} + b \quad b \in \mathbb{R}^{+\star} \end{cases}$

g) $\begin{cases} u_0 &= 1 \\ u_n &= au_{n/2} + b \quad a \in [2, +\infty[, b \in \mathbb{R}^{+\star} \end{cases}$

h) $\begin{cases} u_0 &= 1 \\ u_1 &= 1 \\ u_n &= u_{\lceil \frac{n}{2} \rceil} + u_{\lfloor \frac{n}{2} \rfloor} + bn \quad b \in \mathbb{R}^{+\star} \end{cases}$

Solution

e) Considérons la suite $(v_k)_{k\in\mathbb{N}}$ définie par : $\forall k \in \mathbb{N}, v_k = u_{2^k}$. $(v_k)_{k\in\mathbb{N}}$ vérifie la relation de récurrence ci-dessous.

$$\begin{cases} v_0 &= 1 + b \\ v_k &= v_{k-1} + b2^k \quad k \in \mathbb{N}^\star \end{cases}$$

Ainsi

$$\begin{aligned} v_k &= v_{k-1} + b2^k = v_{k-2} + 2^{k-1}b + 2^k b = v_{k-3} + 2^{k-2}b + 2^{k-1}b + 2^k b \\ &= v_0 + b\left(\sum_{i=0}^{k} 2^i\right) \\ &= (1+b) + b(2^{k+1} - 1) \end{aligned}$$

Par croissance de la suite $(u_n)_{n\in\mathbb{N}}$, pour tout $n \in \mathbb{N}^\star$:

$$\begin{aligned} u_{2^{\lfloor \log_2(n) \rfloor}} &\leqslant u_n \leqslant u_{2^{\lfloor \log_2(n) \rfloor + 1}} \\ v_{\lfloor \log_2(n) \rfloor} &\leqslant u_n \leqslant v_{\lfloor \log_2(n) \rfloor + 1} \\ (1+b) + b(2^{\lfloor \log_2(n) \rfloor + 1} - 1) &\leqslant u_n \leqslant (1+b) + b(2^{\lfloor \log_2(n) \rfloor + 2} - 1) \end{aligned}$$

Finalement $u_n = \Theta(2^{\log_2(n)}) = \Theta(n)$.

f) Considérons la suite $(v_k)_{k\in\mathbb{N}}$ définie par : $\forall k \in \mathbb{N}, v_k = u_{2^k}$. $(v_k)_{k\in\mathbb{N}}$ vérifie la relation de récurrence ci-dessous.

$$\begin{cases} v_0 &= 1 \\ v_k &= 2v_{k-1} + b \quad k \in \mathbb{N}^\star \end{cases}$$

Ainsi

$$\begin{aligned} v_k &= 2v_{k-1} + b = 2^2 v_{k-2} + 2^1 b + b = 2^3 v_{k-3} + 2^2 b + 2^1 b + b \\ &= 2^k v_0 + b\left(\sum_{i=0}^{k-1} 2^i\right) \\ &= 2^k(1+b) - b \end{aligned}$$

♣. Insistons : il s'agit écrire un programme OCaml ayant cette suite comme complexité et non un programme OCaml calculant cette suite.

Par croissance de la suite $(u_n)_{n\in\mathbb{N}}$, pour tout $n \in \mathbb{N}^\star$:

$$\begin{aligned} u_{2^{\lfloor \log_2(n) \rfloor}} &\leqslant u_n \leqslant u_{2^{\lfloor \log_2(n) \rfloor+1}} \\ v_{\lfloor \log_2(n) \rfloor} &\leqslant u_n \leqslant v_{\lfloor \log_2(n) \rfloor+1} \\ 2^{\lfloor \log_2(n) \rfloor}(1+b) - b &\leqslant u_n \leqslant 2^{\lfloor \log_2(n) \rfloor+1}(1+b) - b \end{aligned}$$

Finalement $u_n = \Theta(2^{\log_2(n)}) = \Theta(n)$.

$g)$ Considérons la suite $(v_k)_{k\in\mathbb{N}}$ définie par : $\forall k \in \mathbb{N}, v_k = u_{2^k}$. $(v_k)_{k\in\mathbb{N}}$ vérifie la relation de récurrence ci-dessous.

$$\begin{cases} v_0 &= a + b \\ v_k &= av_{k-1} + b \quad k \in \mathbb{N}^\star \end{cases}$$

Ainsi

$$\begin{aligned} v_k &= av_{k-1} + b = a^2 v_{k-2} + a^1 b + b = a^3 v_{k-3} + a^2 b + a^1 b + b \\ &= a^k v_0 + b \left(\sum_{i=0}^{k-1} a^i \right) \\ &= a^k (a + b + \frac{b}{a-1}) - \frac{b}{a-1} \end{aligned}$$

Par croissance de la suite $(u_n)_{n\in\mathbb{N}}$, pour tout $n \in \mathbb{N}^\star$:

$$\begin{aligned} u_{2^{\lfloor \log_2(n) \rfloor}} &\leqslant u_n \leqslant u_{2^{\lfloor \log_2(n) \rfloor+1}} \\ v_{\lfloor \log_2(n) \rfloor} &\leqslant u_n \leqslant v_{\lfloor \log_2(n) \rfloor+1} \\ a^{\lfloor \log_2(n) \rfloor}(a + b + \frac{b}{a-1}) - \frac{b}{a-1} &\leqslant u_n \leqslant a^{\lfloor \log_2(n) \rfloor+1}(a + b + \frac{b}{a-1}) - \frac{b}{a-1} \end{aligned}$$

Finalement $u_n = \Theta(a^{\log_2(n)}) = \Theta(n^{\log_2(a)})$. En effet $a^{\log_2(n)} = n^{\log_2(a)}$.

$h)$ Considérons la suite $(v_k)_{k\in\mathbb{N}}$ définie par : $\forall k \in \mathbb{N}, v_k = u_{2^k}$. $(v_k)_{k\in\mathbb{N}}$ vérifie la relation de récurrence ci-dessous.

$$\begin{cases} v_0 &= 1 \\ v_k &= 2v_{k-1} + b2^k \quad k \in \mathbb{N}^\star \end{cases}$$

Ainsi

$$\begin{aligned} v_k &= 2v_{k-1} + b2^k = 2^2 v_{k-2} + 2b2^k = 2^3 v_{k-3} + 3b2^k \\ &= 2^k v_0 + kb2^k \\ &= 2^k + kb2^k \end{aligned}$$

Par croissance de la suite $(u_n)_{n\in\mathbb{N}}$, pour tout $n \in \mathbb{N}^\star$:

$$\begin{aligned} u_{2^{\lfloor \log_2(n) \rfloor}} &\leqslant u_n \leqslant u_{2^{\lfloor \log_2(n) \rfloor+1}} \\ v_{\lfloor \log_2(n) \rfloor} &\leqslant u_n \leqslant v_{\lfloor \log_2(n) \rfloor+1} \\ 2^{\lfloor \log_2(n) \rfloor} + \lfloor \log_2(n) \rfloor b2^{\lfloor \log_2(n) \rfloor} &\leqslant u_n \leqslant 2^{\lfloor \log_2(n) \rfloor+1} + (\lfloor \log_2(n) \rfloor + 1)b2^{\lfloor \log_2(n) \rfloor+1} \end{aligned}$$

Finalement $u_n = \Theta(\log_2(n) 2^{\log_2(n)}) = \Theta(n \log_2(n))$.

Exercice 10 : Quelques équivalents classiques

Q. 1 Donner le comportement asymptotique des suites suivantes♣ au moyen de la notation Θ.

$a)$ $\lfloor \log_2(n) \rfloor$ $b)$ $2^{\lfloor \log_2(n) \rfloor}$

Q. 2 Donner le comportement asymptotique des sommes suivantes♡ au moyen de la notation Θ.

$a)$ $\sum_{k=1}^{n} k$ $c)$ $\sum_{k=0}^{n} 2^k$ $e)$ $\sum_{k=0}^{n} k2^k$ $g)$ $\sum_{k=1}^{n} \log k$

$b)$ $\sum_{k=1}^{n} k^2$ $d)$ $\sum_{k=0}^{n} \frac{1}{2^k}$ $f)$ $\sum_{k=1}^{n} \frac{1}{k}$ $h)$ $\sum_{k=0}^{n} k \log k$

Exercice 11 : Éléments majoritaires

On dit qu'un élément x est majoritaire dans un multi-ensemble E de cardinal n lorsque le nombre d'occurrences de x dans E est strictement supérieur à $\frac{n}{2}$. On suppose donc un multi-ensemble E représenté par un tableau de ses éléments et on cherche si oui ou non E admet un élément majoritaire (s'il existe, il est unique). Dans le cas d'une réponse affirmative on renverra l'élément en question. On s'intéresse dans cet exercice au nombre de tests d'égalité (ou inégalité évidemment) effectués entre des éléments du multi-ensemble E. On suppose définie une fonction nbOccur prenant en arguments un tableau T, un élément x, et deux indices i et j du tableau et retournant le nombre d'occurrences de x dans le tableau T entre les indices i et j (au sens large). On suppose de plus que l'appel nbOccur(T, x, i, j) fait $j - i + 1$ comparaisons.♠

Q. 1 Rappeler la définition d'un algorithme naïf majoritaire(T) retournant s'il existe un élément majoritaire du multi-ensemble E représenté par le tableau T. On précisera la complexité pire cas de l'algorithme.

Solution

Algorithme 4 : majoritaire

Entrée : Un multi-ensemble E représenté par un tableau T de taille n
Sortie : L'élément majoritaire de E s'il existe

```
1 pour i = 0 à n − 1 faire
2 │ m ← nbOccur(T, T[i], 0, n − 1) ;
3 │ si m > n/2 alors
4 └   │ retourner T[i]
```

La complexité pire cas est $\sum_{i=0}^{n-1} n = n^2$.

Q. 2 Supposant connu l'élément majoritaire du sous-tableau $T[0..n/2-1]$♢ et l'élément majoritaire du sous-tableau $T[n/2..n-1]$ où T est un tableau de taille n, donner l'élément majoritaire de T en faisant de l'ordre de n comparaisons.

Solution

Si un élément est majoritaire dans $T[0..n-1]$ alors il est majoritaire dans $T[0..n/2-1]$ ou dans $T[n/2..n-1]$, en effet par contraposée, si x apparaît au plus (sens large) $(n/2)/2$ fois dans $T[0..n/2]$ et au plus (sens large) $(n - n/2)/2$ fois dans $T[n/2+1..n-1]$ alors il apparaît

♣. *i.e.* implicitement indexées par n

♡. *i.e.* des suites, implicitement indexées par n, des sommes jusqu'au rang n

♠. Les suppositions des deux phrases précédentes sont là pour vous faire gagner du temps, il **faut** que vous sachiez implémenter un tel algorithme nbOccur.

♢. comprendre : on sait s'il existe un élément majoritaire et s'il existe, on sait qui il est

au plus $n/2$ fois dans $T[0..n-1]$.
Notons x_g l'élément majoritaire de $T[0..n/2]$ et x_d l'élément majoritaire de $T[n/2+1..n-1]$ (on dénote par $x_g = \text{None}$ l'inexistence d'un élément majoritaire, de même pour x_d).

- Si $x_g = x_d$ (cas où $x_g = x_d = \text{None}$ compris) alors l'élément majoritaire est x_g
- Sinon si $x_g \neq \text{None}$ on teste (en n comparaisons) si x_g est majoritaire
- Sinon si $x_d \neq \text{None}$ on teste (en n comparaisons) si x_d est majoritaire
- Sinon on retourne None

Q. 3 Donner alors un algorithme du paradigme diviser pour régner permettant la résolution du problème de la recherche d'un élément majoritaire.

Solution

Algorithme 5 : majoritaire

Entrée : Un tableau T de taille n, deux indices i et j
Sortie : L'élément majoritaire de $T[i..j]$ s'il existe

```
1  m ← (i + j)/2 ;                                        // ou plutôt i + (j − i)/2
2  x_g ← majoritaire(T, i, m) ;
3  x_d ← majoritaire(T, m + 1, j) ;
4  si x_g = x_d alors
5  |  retourner x_g
6  sinon si x_g ≠ None  et  nbOccur(T, x_g, i, j) > (j − i + 1)/2 alors
7  |  retourner x_g
8  sinon si x_d ≠ None  et  nbOccur(T, x_d, i, j) > (j − i + 1)/2 alors
9  |  retourner x_d
10 sinon
11 |  retourner None
```

Q. 4 Donner la complexité pire cas dans le cas où la taille du tableau donné en entrée est une puissance de 2.

Solution

En notant C_n cette complexité on a $C_n = C_{\lceil \frac{n}{2} \rceil} + C_{\lfloor \frac{n}{2} \rfloor} + 2n$. On en déduit une complexité en $\Theta(n \log_2(n))$.

Exercice 12 : Sous-tableau de somme maximale

Étant donné un tableau d'entiers, le problème du sous-tableau maximum consiste à trouver un ensemble d'indices consécutifs (*i.e.* un intervalle d'indices) non vide qui maximise la somme des éléments du tableau. Par exemple, le sous-tableau maximum du tableau `[-3, 4, 5, -1, 2, 3, -6, 4]` est le tableau`[4, 5, -1, 2, 3]` de valeur 13. Étant donné un tableau T on note donc $\text{stm}(T)$ la valeur du sous-tableau maximum.

Q. 1 Formaliser ce problème comme un problème d'optimisation.

Solution

entrée : A un tableau indicé par $[\![1, n]\!]$

sortie : $\underbrace{\max \left\{ \sum_{k=i}^{j} A[k] \;\middle|\; (i,j) \in [\![1,n]\!]^2,\ i<j \right\}}_{:=stm(A)}$

Q. 2 Dans un premier temps, on considère l'algorithme de résolution ci-dessous.
Quelle est la complexité de cet algorithme ?

Algorithme 6 : Sous tableau maximum

Entrée : A un tableau de taille n
Sortie : $\mathsf{stm}(A)$

```
1 A_max ← max A[1], ..., A[n] ;
2 pour i = 1 à n faire
3 |  pour j = i à n faire
4 |  |  sum ← 0 ;
5 |  |  pour k = i à j faire
6 |  |  |  sum ← sum + A[j] ;
7 |  |  |  si sum > A_max alors
8 |  |  |  |  A_max ← sum
9 retourner A_max
```

Solution

Cet algorithme est en $\Theta(n^3)$.

Q. 3 Proposer une variante en $\Theta(n^2)$ de cet algorithme.

Solution

Il suffit de ne pas recalculer la valeur d'une fenêtre de taille j en partant de 0, mais en partant de la valeur pour la fenêtre de taille $j-1$ calculée juste avant.

```
A_max = max A[1], ..., A[n]
pour i allant de 1 à n faire
    sum ← 0
    pour j allant de i à n faire
        sum ← sum + A[j]
        si sum > A_max
        alors A_max ← sum
retourner A_max
```

Dans la suite de l'exercice, on s'intéresse à des algorithmes de type diviser pour régner. Pour un tableau A indicé par $[1..n]$ avec $n > 1$, on note $A_1 = A[1..m]$ et $A_2 = A[m+1..n]$ les deux sous-tableaux définis par $m = \left\lfloor \frac{n+1}{2} \right\rfloor$. On note alors $\mathsf{stm}_1(A) = \mathsf{stm}(A_1)$ et $\mathsf{stm}_2(A) = \mathsf{stm}(A_2)$. On introduit aussi $\mathsf{stm}_3(A)$ la valeur maximum d'un sous-tableau de A qui couvre à la fois les indices m et $m+1$.

Q. 4 Comment calculer $\mathsf{stm}_3(A)$ efficacement ? Quelle est la complexité en fonction de n la taille de A ?

Solution

$\mathsf{stm}_3(A) = \max_{i\in[1..m]} \sum_{k=i}^{m} A[k] + \max_{j\in[m+1..n]} \sum_{k=m+1}^{j} A[k]$. Ces deux max sont indépendants et chacun se calcule en $\Theta(n)$. (On peut même dire qu'on fait $m-1$ étapes pour le premier et $n-m-2$ pour le deuxième).

Q. 5 Connaissant $\mathsf{stm}_1(A)$, $\mathsf{stm}_2(A)$ et $\mathsf{stm}_3(A)$, quelle est la valeur de $\mathsf{stm}(A)$? Dans le cadre d'une méthode diviser pour régner qui calcule $\mathsf{stm}(A)$ en calculant récursivement et $\mathsf{stm}_1(A)$ et $\mathsf{stm}_2(A)$ en quoi consisterait l'opération de fusion ? Quelle serait la complexité de cette opération ?

Solution

On peut partitionner l'ensemble des sous-tableaux de A en trois sous-ensembles : les sous-tableaux entièrement dans l'intervalle $[\![1, m]\!]$, ceux entièrement dans l'intervalle $[\![m+1, n]\!]$, et ceux à cheval. Notons-les respectivement $\mathcal{S}_1$, $\mathcal{S}_2$ et $\mathcal{S}_3$.

$$\begin{aligned}
\mathsf{stm}(A) &= \max\{\mathsf{sum}(t) \mid t \text{ un sous-tableau de } A\} \\
&= \max\{\mathsf{sum}(t) \mid t \in \mathcal{S}_1 \sqcup \mathcal{S}_2 \sqcup \mathcal{S}_3\} \\
&= \max\left(\max\{\mathsf{sum}(t) | t \in \mathcal{S}_1\}, \max\{\mathsf{sum}(t) | t \in \mathcal{S}_2\}, \max\{\mathsf{sum}(t) | t \in \mathcal{S}_3\}\right) \\
&= \max(\mathsf{stm}_1(A), \mathsf{stm}_2(A), \mathsf{stm}_3(A))
\end{aligned}$$

Le maximum entre trois valeurs se calculant en temps constant, la complexité de la fusion est donc liée au calcul de $\mathsf{stm}_3(A)$ citée plus haut.

Q. 6 Soit T_n le nombre d'opérations élémentaires (additions et comparaisons d'éléments du tableau) que réalise cette méthode pour un tableau de taille n. Donner l'équation de récurrence satisfaite par T_n. En déduire la complexité de la méthode.

Solution

$T_n = \underbrace{0}_{\text{diviser}} + \underbrace{T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil}}_{\text{régner}} + \underbrace{C\,n}_{\text{fusionner}}$ pour une certaine constante C.

On reconnaît l'équation de récurrence que vérifie la complexité du tri fusion, ainsi on sait que la complexité de cet algorithme est en $\Theta(n \log n)$.

On s'intéresse maintenant à une autre approche diviser pour régner afin de réduire encore la complexité. Pour ce faire, on définit, pour un tableau A indicé par $[1..n]$ les valeurs suivantes :

- $\mathsf{pref}(A) = \max\{\sum_{k=1}^{i} A[k] | i \in [1..n]\}$
- $\mathsf{suff}(A) = \max\{\sum_{k=i}^{n} A[k] | i \in [1..n]\}$
- $\mathsf{tota}(A) = \sum_{k=1}^{n} A[k]$

Autrement dit, $\mathsf{pref}(A)$ (resp. $\mathsf{suff}(A)$) est la valeur maximum d'un sous-tableau préfixe (resp. suffixe) de A, et $\mathsf{tota}(A)$ est la somme des valeurs de A.

Q. 7 Expliquer comment calculer ces valeurs pour un tableau A.

Solution

On calcule ces valeurs pour A_1 et A_2 récursivement à l'aide des formules suivantes (que l'on peut justifier comme précédemment par disjonction de cas et associativité du max.
$\mathsf{stm}(A) = \max\{\mathsf{stm}_1, \mathsf{stm}_2, \mathsf{suff}_1 + \mathsf{pref}_2\}$
$\mathsf{pref}(A) = \max\{\mathsf{pref}_1, \mathsf{tota}_1 + \mathsf{pref}_2\}$
$\mathsf{suff}(A) = \max\{\mathsf{suff}_2, \mathsf{suff}_1 + \mathsf{tota}_2\}$
$\mathsf{tota}(A) = \mathsf{tota}_1 + \mathsf{tota}_2$

La complexité de cette opération de fusion est donc $\Theta(1)$.

Q. 8 Quelle est la complexité de l'algorithme diviser pour régner associé ? Justifier.

Solution

On note T_n le nombre d'opérations de ce dernier algorithme pour un tableau de taille n.
$T_n = \underbrace{0}_{\text{diviser}} + \underbrace{T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil}}_{\text{régner}} + \underbrace{1}_{\text{fusionner}}$.
On peut alors montrer que $T_n \in \Theta(n)$.

Exercice 13 : Multiplication d'entiers selon Karatsuba

Dans cet exercice on s'intéresse au problème de la multiplication de grands entiers machines. On suppose les entiers représentés par la séquence indicée des bits de leur décomposition en base 2. Par exemple, l'entier 14 est représenté par la séquence $(1, 1, 1, 0)$. On appellera alors taille d'un entier la longueur d'une telle séquence. On suppose de telles séquences stockées dans des tableaux permettant l'indexation en temps constant. On remarque que les multiplications et divisions entières par des puissances de 2 correspondent à des décalages de bits. En effet, si $(u_{n-1}, u_{n-2}, \ldots, u_0)$ est la représentation d'un entier u et $p \in \mathbb{N}$ alors $u/2^p$ est représenté par $(u_{n-p-1}, u_{n-p-2}, \ldots, u_p)$ et $u \times 2^p$ par $(u_{n-1}, u_{n-2}, \ldots, u_0, \underbrace{0, 0, \ldots 0}_{p})$. Dans tout l'exercice on s'intéresse uniquement au problème de la multiplication de deux entiers de même taille et on mesure la complexité au regard du nombre de multiplications de nombres à 1 bit. Ainsi l'opération 1×0 est considérée comme une opération atomique de coût 1 alors que le coût de calcul de 1110110×101110 dépend de l'algorithme choisi pour faire le calcul de $\times$.

Q. 1 Expliquer en quoi l'hypothèse d'une taille commune des deux opérandes n'est pas trop simplificatrice.

Q. 2 Rappeler l'algorithme de multiplications naïf d'entiers (celui appris en primaire), décliné pour des nombres écrits en base 2. Quelle est sa complexité ?

Solution

Algorithme 7 : Produit naïf d'entiers

Entrée : Deux suites $u = (u_{n-1-i})_{i\in[\![0,n-1]\!]}$ et $v = (v_{n-1-i})_{i\in[\![0,n-1]\!]}$ de bits
Sortie : Une suite de bits représentant l'entier $u \times v$

```
1 R ← ();
2 pour i = 0 à n − 1 faire
3 |   w = (0)_{i∈[[0,n−1]]};
4 |   pour j = 0 à n − 1 faire
5 |   |   w_j ← u_i × v_j
6 |   R ← R + w × 2^i
7 retourner R
```

On a une complexité en $\Theta(n^2)$.

On remarque que si $u = u_a + 2^p u_b$ avec $u_a < 2^p$ et $v = v_a + 2^p v_b$ avec $v_a < 2^p$ alors $u \times v = u_a \times v_a + 2^p(u_b \times v_a + v_b \times u_a) + 2^{2p} u_b \times v_b$.

Q. 3 Proposer un algorithme de multiplication d'entiers utilisant le paradigme diviser-pour-régner et utilisant la remarque précédente. Faire une rapide étude de complexité dans le cas où les entrées sont de taille 2^k. Cet algorithme diviser-pour-régner présente-t-il un avantage par rapport à l'algorithme naïf.

Solution

$$\begin{aligned}
\mathsf{pdt}(u,v) &= u_0 \times v_0 && \text{si } u = (u_0) \text{ et } v = (v_0)\\
\mathsf{pdt}(u,v) &= 0 && \text{sinon si } u = () \text{ ou } v = ()\\
\mathsf{pdt}(u,v) &= \mathsf{pdt}(u_a,v_a) + 2^p(\mathsf{pdt}(u_b,v_a) + \mathsf{pdt}(u_a,v_b)) && \text{sinon}\\
&\quad + 2^{2p}\mathsf{pdt}(u_b,v_b)
\end{aligned}$$

Avec les mêmes notations que ci-avant. Notons C_p la complexité de $\mathsf{pdt}(u,v)$ sur deux entrées de taille 2^p. On a $C_0 = 1$ et $C_{p+1} = 4C_p$, finalement $C_p = 4^p$ et donc pour des entrées de taille n qui sont des puissances de 2 on a une complexité de l'ordre de $4^{\log_2(n)} = n^2$. Donc pas avantageux.

Q. 4 En s'intéressant à la valeur de $u_a v_a + u_b v_b - (u_a - u_b)(v_a - v_b)$ proposer un algorithme faisant trois appels récursifs sur des entrées dont la taille est divisée par 2.

Solution

Algorithme 8 : Karatsuba

Entrée : Deux suites $u = (u_{n-1-i})_{i \in [\![0,n-1]\!]}$ et $v = (v_{m-1-i})_{i \in [\![0,m-1]\!]}$ de bits

Sortie : Une suite de bits représentant l'entier $u \times v$

```
1  si n ⩽ 1 ou m ⩽ 1 alors
2  |  retourner u × v
3  sinon
4  |  p = min(n, m)/2 ;
5  |  u_a = u/10^p ;
6  |  u_b = u mod 10^p ;
7  |  v_a = v/10^p ;
8  |  v_b = v mod 10^p ;
9  |  x = Karatsuba(u_a, v_a) ;
10 |  y = Karatsuba(u_b, v_b) ;
11 |  z = Karatsuba((u_a − u_b), (v_a − v_b)) ;
12 |  retourner x10^{2p} + (x + y − z)10^p + y
```

Remarque : Les appels à $(u_a - u_b)$ ou $(v_a - v_b)$ peuvent être négatifs, auquel cas on effectue les appels récursifs avec les valeurs opposées en se rappelant qu'on doit multiplier le résultat par ± 1.

Q. 5 Faire une étude rapide de complexité dans le cas où les entiers en arguments sont de taille $n = 2^p$.

Solution

Notons C_p la complexité de Karatsuba(u, v) sur deux entrées de taille 2^p. On a $C_0 = 1$ et $C_{p+1} = 3C_{p-1}$, finalement $C_p = 3^p$ et donc pour des entrées de taille n qui sont des puissances de 2 on a une complexité de l'ordre de $3^{\log_2(n)} = n^{\log_2(3)}$.

Q. 6 Donner la complexité pire cas de l'algorithme de la question 4 au moyen d'un Θ. Au vu de la question précédente, on pourra intuiter puis démontrer par récurrence un encadrement sur le nombre de multiplications élémentaires effectuées par l'algorithme de la question 4 sur deux entrées de taille respectivement n et m.

Solution

Montrons donc par récurrence que le nombre de multiplications élémentaires effectué par Karatsuba(u, v) sur deux entrées de taille respectivement n et m, dénoté par $c_{u,v}$, est tel que :

$$3^{\lfloor \log_2(\min(n,m)) \rfloor} \leqslant c_{u,v} \leqslant 3^{\lceil \log_2(\min(n,m)) \rceil}$$

- Initialisation : Si $n = 1$ ou $m = 1$ alors $c_{u,v} = 1 = 3^0$.
- Hérédité : Sinon, en notant $u_a = u/10^p$, $u_b = u \text{ mod } 10^p$, $v_a = v/10^p$ $v_b = v \text{ mod } 10^p$, on a $c_{u,v} = c_{u_a,v_a} + c_{u_b,v_b} + c_{u_a+u_b,v_a+v_b}$ de plus en supposant, sans perdre en généralité que $n \leqslant m$ on a que $p = n/2$ et $|u_b| = p = |v_b|$, $|u_a| = n - p$ et $|v_a| = m - p$, de plus

$|u_a - u_b| \leqslant n-p$ et $|v_a - v_b| \leqslant m-p$. On en conclut par hypothèse de récurrence que :

$$\begin{aligned}
3^{\lfloor \log_2(p) \rfloor} + 3^{\lfloor \log_2(n-p) \rfloor} + 3^{\lfloor \log_2(n-p) \rfloor} \leqslant & c_{u,v} \leqslant 3^{\lceil \log_2(p) \rceil} + 3^{\lceil \log_2(n-p) \rceil} + 3^{\lceil \log_2(n-p) \rceil} \\
3 \times 3^{\lfloor \log_2(p) \rfloor} \leqslant & c_{u,v} \leqslant 3 \times 3^{\lceil \log_2(n-p) \rceil} \\
3 \times 3^{\lfloor \log_2(p) \rfloor} \leqslant & c_{u,v} \leqslant 3 \times 3^{\lceil \log_2(n-p) \rceil} \\
3^{\lfloor \log_2(2p) \rfloor} \leqslant & c_{u,v} \leqslant 3^{\lceil \log_2(2(n-p)) \rceil}
\end{aligned}$$

- Si n est pair, on a $n = 2p$ et donc $2(n-p) = n$
- Si n est impair, alors $n = 2p+1$ donc $2(n-p) = n+1$, mais $\lceil \log_2(n) \rceil = \lceil \log_2(n+1) \rceil$ par parité, mais aussi (par parité toujours) $\lfloor \log_2(n) \rfloor = \lfloor \log_2(n-1) \rfloor = \lfloor \log_2(2p) \rfloor$, finalement

$$3^{\lfloor \log_2(n) \rfloor} \leqslant c_{u,v} \leqslant 3^{\lceil \log_2(n) \rceil}$$

Aussi on donc bien le résultat par récurrence.
Nous avons donc que la complexité pire cas est un $\Theta(n^{\log_2(3)})$.

Q. 7 Dans les questions précédentes nous avons ignoré le coût des additions et soustractions sur les grands entiers. On ne suppose plus que c'est le cas : une addition, soustraction sur deux entiers de tailles n et m coûte dorénavant $\max(n, m)$. Faire une rapide étude de complexité de l'algorithme de la question 4 (on se contentera des entiers de la forme 2^p). Conclure.

Solution

Notons alors C_p la complexité de Karatsuba(u, v) sur deux entrées de taille 2^p. On remarque alors qu'il y a 4 additions/soustractions sur des entiers de taille $2^p/2$ et deux additions sur des entiers de taille 2^p. On a $C_0 = 1$ et $C_{p+1} = 3C_{p-1} + 4\frac{2^p}{2} + 2 \times 2^p = 3C_{p-1} + 4 \times 2^p = 3^2 C_{p-2} + 4(2^p + 2^{p-1}) = \cdots = 3^p C_0 + 4(\sum_{i=1}^{p} 2^i) = 3^p + 2^{p+1} - 1$ et donc pour des entrées de taille n qui sont des puissances de 2 on a une complexité de l'ordre de $3^{\log_2(n)} + 2n - 1 = \Theta(n^{\log_2(3)})$. Les opérations sur les entiers ont une contribution négligeable.

Exercice 14 : Algorithmes en logique propositionnelle (révisions)

On fixe dans cet exercice la formule $H = \Big((y \vee x) \wedge (\neg y \vee z)\Big) \leftrightarrow \Big(z \rightarrow (x \wedge y)\Big)$

Q. 1 Appliquer l'algorithme du cours pour fournir H' une FND équivalente à H.

Solution

On commence par établir la table de vérité de H.

x	y	z	$\overbrace{y \vee x}^{a}$	$\overbrace{\neg y \vee z}^{b}$	$z \rightarrow (x \wedge y)$	$a \wedge b$	H
F	F	F	F	V	V	F	F
F	F	V	F	V	F	F	V
F	V	F	V	F	V	F	F
F	V	V	V	V	F	V	F
V	F	V	V	V	V	V	V
V	F	V	V	V	F	V	F
V	V	V	F	F	V	F	F
V	V	V	V	V	V	V	V

On utilise alors les lignes où H est à V pour construire une FND équivalente à H.
$H \equiv (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z)$

Q. 2 Appliquer l'algorithme du cours pour fournir H'' une FNC équivalente à H.

Solution

On utilise cette fois les lignes de la table de vérité où H est à F pour construire une FNC équivalente à H.

$$\begin{aligned} H &\equiv \neg(\neg x \wedge \neg y \wedge \neg z) \wedge \neg(\neg x \wedge y) \wedge \neg(x \wedge \neg y \wedge z) \wedge \neg(x \wedge y \wedge \neg z) \\ &\equiv (x \vee y \vee z) \wedge (x \vee \neg y) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \end{aligned}$$

Q. 3 Appliquer à H' un algorithme polynomial permettant de résoudre le problème FND-Sat. Que peut-on en déduire pour H ?

Solution

Rappels : Une FND est satisfiable ssi une de ses clauses conjonctives l'est, et une clause conjonctive est satisfiable ssi elle ne contient pas un littéral et son opposé, en effet dans ce cas il suffit de considérer un environnement qui met à vrai chaque littéral pour avoir un modèle. On décide donc si une FND est satisfiable en temps polynomial, en examinant chaque clause à l'aide d'un tableau indexé par les variables indiquant si la variable apparaît positivement, négativement ou pas du tout dans la clause. La première clause H', à savoir $\neg x \wedge \neg y \wedge z$ ne contient aucun littéral et son opposé, ainsi elle nous donne un environnement propositionnel qui satisfait H' : $x \mapsto \mathsf{F}, y \mapsto \mathsf{F}, z \mapsto \mathsf{V}$.

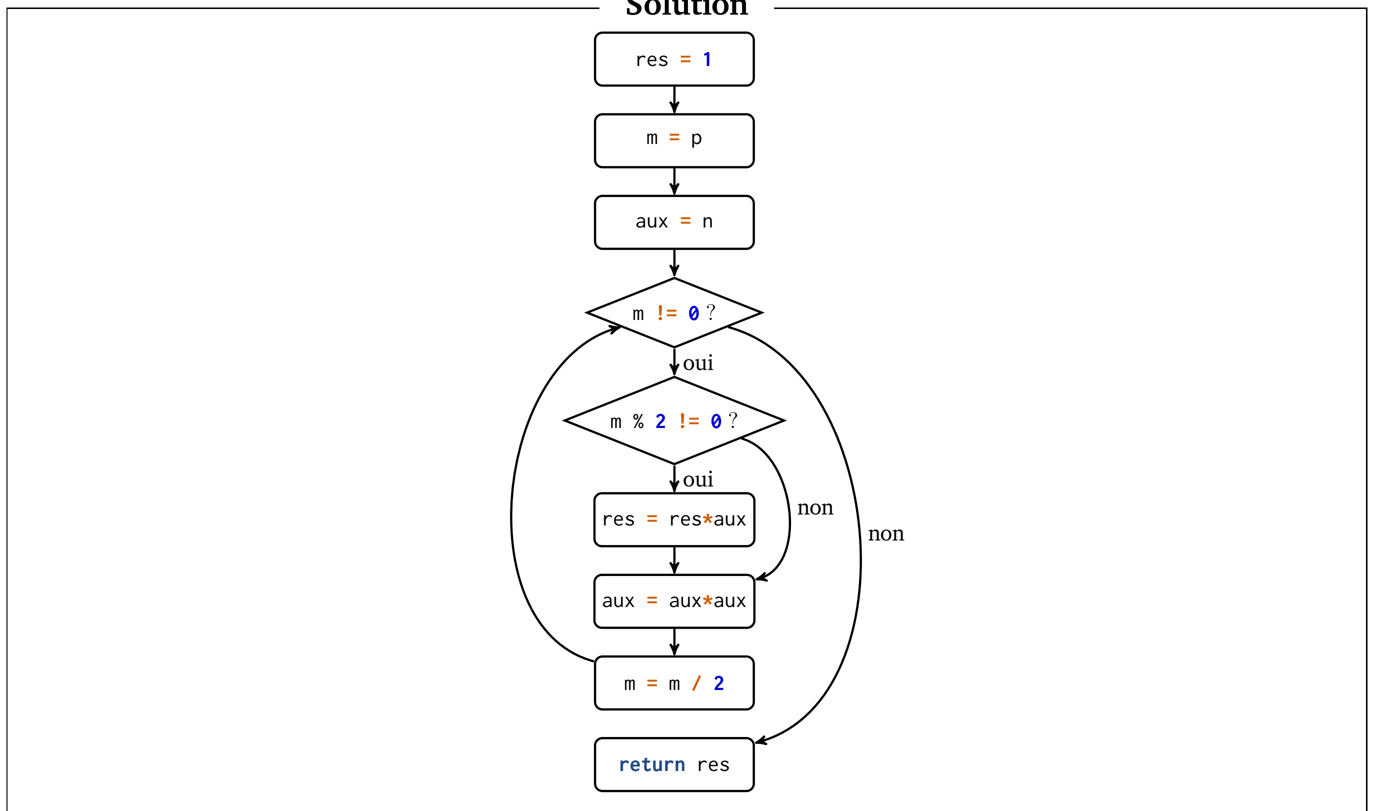
Q. 4 Appliquer l'algorithme de Quine pour déterminer si H'' est satisfiable, et dans ce cas donner un environnement propositionnel qui la satisfait.
On traitera les variables dans l'ordre alphabétique et on commencera par appliquer une substitution par $\top$.

Exercice 15 : Graphe de flot de contrôle

Q. 1 Donner le graphe de flot de contrôle de la fonction `expo_rapide` dont le code C est donné ci-dessous.

```
int expo_rapide(int n, int p) {
  /* hyp : p ⩾ 0
     retourne n^p
   */
  int res;
  int m;
  int aux;

  res = 1;
  m = p;
  aux = n;
  while (m != 0) {
    if (m % 2 != 0) {
      res = res * aux ;
    }
    aux = aux * aux;
    m = m / 2;
  }
  return res;
}
```



Q. 2 Proposer un couple (n,p) de valeurs assurant que chaque arc du graphe de flot de contrôle de la question précédente est emprunté lors de l'exécution de la fonction `expo_rapide` sur ces valeurs n et p.

Solution

Il suffit de choisir $\mathtt{n} = 1$ et $\mathtt{m} = 2$.

Q. 3 *Question subsidiaire* Proposer un invariant de boucle liant les variables `res`, `aux`, `n` et `m` qui permettrait de montrer la correction de la fonction `expo_rapide`.

Solution

$\mathtt{res} \times \mathtt{aux}^{\mathtt{m}} = \mathtt{n}^{\mathtt{p}}$