Analyses de programmes

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve: 3 heures 30 minutes

Juin/Juillet 2018

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !
Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple: $O(n^2)$, $O(n \log n)$,...
Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***.
Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

1 Génération de programmes

Si $a \geqslant 0$ et $b > 0$ sont deux entiers, on note $a \bmod b$ le reste de la division euclidienne de a par b, autrement dit l'unique entier r avec $0 \leqslant r < b$ tel qu'il existe un entier q satisfaisant $a = bq + r$. Le quotient q est quant à lui noté $(a|b)$. On note $|x|$ la valeur absolue de l'entier relatif x. On utilise la notation $[x_1, x_2, \ldots, x_k]$ pour dénoter la liste d'éléments $x_1, x_2, \ldots, x_k$ (précisement dans cet ordre et possiblement avec des redondances) et la notation $\{x_1, x_2, \ldots, x_k\}$ pour dénoter l'ensemble d'éléments $x_1, x_2, \ldots, x_k$, qui sont deux-à-deux distincts et dont l'ordre d'écriture est sans importance.

On fixe pour tout le sujet

$$m = 2^{31} - 1 = 2\,147\,483\,647$$

et l'on définit les suites $(\tilde{u}(n))_{n \in [\![0, 2^{20}[\![}$ et $(u(n))_{n \in \mathbb{N}}$ par

$$\tilde{u}(0) = u_0, \qquad \tilde{u}(n+1) = \big(16\,807 \times \tilde{u}(n) + 17\big) \bmod m \quad u(n) = \tilde{u}(n \bmod 2^{20})$$

avec u_0 l'entier qui vous a été donné (à reporter sur votre fiche réponse) et $2^{20} = 1\,048\,576$.

Question 1 *Calculer* $u(n) \bmod 101$ *pour*
a) $n = 16$ ***b)*** $n = 1024$ ***c)*** $n = 1048592$

On définit $(v(n))_{n \in \mathbb{N}}$ par

$$v(n) = \sum_{k=0}^{7} (u(n+k) \bmod 2)$$

Question 2 *Calculer* $v(n)$ *pour*
a) $n = 32$ ***b)*** $n = 256$ ***c)*** $n = 65536$

Dans ce sujet, nous considérons un langage assembleur idéalisé. Ce langage dispose de 2048 variables, notées r_0 à r_{2047} et ordonnées selon leur indice ($r_0 < r_1 < \ldots < r_{2047}$).

Une instruction assembleur est de la forme

$$dst \overset{op}{\leftarrow} srcs \Rightarrow jmp$$

où *dst* spécifie la variable modifiée par l'instruction, *srcs* spécifie la liste des variables lues par l'instruction et *jmp* spécifie l'adresse à laquelle le programme peut sauter directement après exécution de cette instruction. Une instruction modifie au plus une variable (0 ou 1), lit au plus deux variables (0, 1 ou 2) et spécifie au plus une adresse de saut direct (0 ou 1). On appelle NOP (qui signifie "no operation") l'instruction $[] \overset{op}{\leftarrow} [] \Rightarrow []$ qui n'effectue ni lecture, ni écriture, ni saut direct. Étant donnée une instruction $i = dst \overset{op}{\leftarrow} srcs \Rightarrow jmp$, on note respectivement

$$\boldsymbol{définies}(i) = \begin{cases} \{r\} & \text{si } dst = [r] \\ \{\,\} & \text{si } dst = [\,] \end{cases} \qquad \boldsymbol{lues}(i) = \bigcup_{r \in srcs} \{r\}$$

les *ensembles* de variables respectivement définies et lues par l'instruction i. On note également

$$\boldsymbol{suivants}(i) = \begin{cases} \{j\} & \text{si } jmp = [j] \\ \{\,\} & \text{si } jmp = [\,] \end{cases}$$

l'ensemble (vide ou singleton) de l'adresse du saut direct potentiellement effectué par l'instruction.

$$
\begin{array}{l}
r_0 = 0 \\
\textsf{while } r_0 < 42 : \\
\quad r_1 = r_0 + 1 \\
\quad r_2 = r_2 + r_1 \\
\quad r_0 = r_1 \times 2 \\
\textsf{return } r_2
\end{array}
$$

(a) Programme source

$$
\begin{array}{ll}
 & r_0 := 0 \\
L_1 : & r_1 := r_0 + 1 \\
 & r_2 := r_2 + r_1 \\
 & r_0 := r_1 \times 2 \\
 & \textsf{if } r_0 < 42 \textsf{ goto } L_1 \\
 & \textsf{return } r_2
\end{array}
$$

(b) Programme assembleur

$$
\begin{array}{l}
i = 0 : [r_0] \overset{op}{\leftarrow} [\quad] \Rightarrow [] \\
i = 1 : [r_1] \overset{op}{\leftarrow} [r_0] \Rightarrow [] \\
i = 2 : [r_2] \overset{op}{\leftarrow} [r_2, r_1] \Rightarrow [] \\
i = 3 : [r_0] \overset{op}{\leftarrow} [r_1] \Rightarrow [] \\
i = 4 : [\ ] \overset{op}{\leftarrow} [r_0] \Rightarrow [1] \\
i = 5 : [\ ] \overset{op}{\leftarrow} [r_2] \Rightarrow []
\end{array}
$$

(c) Programme idéalisé

FIGURE 1 – Exemple de programme

Un programme *prog* est une liste d'instructions. L'adresse d'une instruction correspond à sa position dans le programme, comptée à partir de 0. En plus des sauts directs, spécifiés par *jmp*, chaque instruction peut potentiellement exécuter l'instruction qui la suit dans le programme, sauf la dernière instruction du programme qui clos l'exécution. On note ***vars***(*prog*) la liste, triée selon l'ordre croissant, des variables distinctes utilisées (en lecture ou écriture) par le programme *prog*. Par exemple, cette liste vaut $[r_0, r_1, r_2]$ pour le programme donné en Figure 1c.

On remarque que l'on fait abstraction de la sémantique exacte des opérations. Par exemple, le programme présenté en Figure 1a, qui correspond au programme assembleur donné en Figure 1b, est représenté en assembleur idéalisé par le programme en Figure 1c. On concentre ainsi notre attention sur les transferts mémoires *potentiellement* effectués par le programme.

Dans ce sujet, on va analyser de tels programmes afin de les optimiser en éliminant du code inutile. Cette optimisation portera uniquement sur l'observation des données manipulées, les opérations de calcul en elles-même ne nous intéresseront pas. La difficulté sera par contre de suivre le flot d'exécution, que l'on va modéliser à l'aide d'un graphe (section 2), pour pouvoir analyser le programme (section 3) avant d'effectuer des éliminations de code (sections 4 et 5).

À partir des suites $(u(n))_{n\in\mathbb{N}}$ et $(v(n))_{n\in\mathbb{N}}$ définies précédemment, nous allons construire un générateur pseudo-aléatoire de programmes de taille $p \in \mathbb{N}$ (noté $(\mathcal{G}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$). À cette fin, il nous faut définir un générateur de variables modifiées par le programme (noté $(\mathcal{D}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$), de variables lues par le programme (noté $(\mathcal{S}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$) et enfin d'adresses de sauts directs (noté $(\mathcal{J}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$). On définit ces diverses suites dans les paragraphes suivants.

On définit la suite $(\mathcal{J}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$ des adresses de saut direct d'un programme pseudo-aléatoirement généré de taille p par

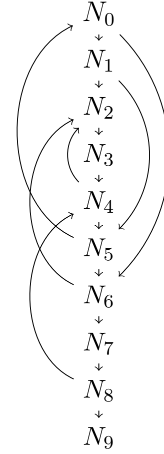
$$
\mathcal{J}_p(n,i) = \begin{cases} [\min(p-1, |i + v(n) - 8|)] & \text{si } u(n) \bmod 2 = 0 \text{ et } i < p-1 \\ [] & \text{sinon} \end{cases}
$$

On définit la suite $(\mathcal{D}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$ des variables modifiées par

$$
\mathcal{D}_p(n,i) = \begin{cases} [r_{((i|32)+v(n)) \bmod 2048}] & \text{si } u(n) \bmod 11 \neq 0 \text{ et } i < p-1 \\ [] & \text{sinon} \end{cases}
$$

$$
\begin{array}{ll}
i=0: & [r_2] \overset{op}{\leftarrow} [\quad\quad] \Rightarrow [6] \\
i=1: & [\ \ ] \overset{op}{\leftarrow} [r_2, r_2] \Rightarrow [5] \\
i=2: & [\ \ ] \overset{op}{\leftarrow} [r_2, r_2] \Rightarrow [3] \\
i=3: & [r_3] \overset{op}{\leftarrow} [r_2, r_2] \Rightarrow [] \\
i=4: & [r_2] \overset{op}{\leftarrow} [\quad\quad] \Rightarrow [2] \\
i=5: & [r_3] \overset{op}{\leftarrow} [\quad\quad] \Rightarrow [0] \\
i=6: & [r_4] \overset{op}{\leftarrow} [r_3\quad] \Rightarrow [2] \\
i=7: & [r_5] \overset{op}{\leftarrow} [r_3\quad] \Rightarrow [] \\
i=8: & [r_4] \overset{op}{\leftarrow} [r_3\quad] \Rightarrow [4] \\
i=9: & [\ \ ] \overset{op}{\leftarrow} [r_3, r_4] \Rightarrow []
\end{array}
$$

(a) $\mathcal{G}_{p=10}(n = 54, i)$



(b) $CFG_{p=10}(n = 54)$

FIGURE 2 – Programme généré avec $\widetilde{u_0} = 42$

Enfin, on définit la suite $(\mathcal{S}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$ des variables lues par

$$
\mathcal{S}_p(n,i) = \begin{cases} [] & \text{si } i = 0 \\ [x,y] & \text{si } i > 0, \mathcal{E}(k_0) = [x] \text{ et } \mathcal{E}(k_1) = [y] \\ [x] & \text{si } i > 0, \mathcal{E}(k_0) = [x] \text{ et } \mathcal{E}(k_1) = [] \\ [y] & \text{si } i > 0, \mathcal{E}(k_0) = [] \text{ et } \mathcal{E}(k_1) = [y] \\ [] & \text{si } i > 0, \mathcal{E}(k_0) = [] \text{ et } \mathcal{E}(k_1) = [] \end{cases}
$$

$$
\text{avec} \begin{cases} \mathcal{E}(k) = \mathcal{D}_p(n - i + k, k) \text{ pour tout } k \in [\![0,p[\![\\ k_0 = \max(0, i - 1 - (u(n) \bmod 8)) \\ k_1 = \max(0, i - 1 - (u(n+1) \bmod 8)) \end{cases}
$$

Question 3 *Calculer* $(\mathcal{D}_p(32+n,32), \mathcal{S}_p(32+n,32), \mathcal{J}_p(32+n,32))$ *pour*
a) $(p,n) = (64,10)$ ***b)*** $(p,n) = (128,18)$ ***c)*** $(p,n) = (256,22)$

On obtient ainsi un générateur pseudo-aléatoire de programmes idéalisés de taille p en définissant la suite $(\mathcal{G}_p(n,i))_{n\geqslant 10, i\in[\![0,p[\![}$ par

$$
\mathcal{G}_p(n,i) = \mathcal{D}_p(n+i,i) \overset{op}{\leftarrow} \mathcal{S}_p(n+i,i) \Rightarrow \mathcal{J}_p(n+i,i)
$$

Pour p et n fixés, la liste d'instructions $[\mathcal{G}_p(n,0), \mathcal{G}_p(n,1), \ldots, \mathcal{G}_p(n,p-1)]$ correspond alors à un programme pseudo-aléatoirement généré de taille p. Un programme de $p = 10$ instructions généré à partir de la graine $\widetilde{u_0} = 42$ pour $n = 54$ est ainsi présenté en Figure 2a.

On note $|\boldsymbol{vars}(prog)|$ le nombre de variables distinctes utilisées (en lecture ou écriture) par le programme *prog*. De façon similaire, on note $|\boldsymbol{sauts}(prog)|$ le nombre d'adresses de saut direct distinctes contenues dans ce programme.

Question 4 *Calculer* $(|\boldsymbol{vars}((\mathcal{G}_p(n,i))_{i\in[\![0,p[\![})| \bmod 101, |\boldsymbol{sauts}((\mathcal{G}_p(n,i))_{i\in[\![0,p[\![})| \bmod 101)$ *pour*
a) $(p,n) = (10,100)$ ***b)*** $(p,n) = (100,200)$ ***c)*** $(p,n) = (20421,300)$

2 Graphe de flot de contrôle

Le graphe de flot de contrôle d'un programme *prog* de taille p est un graphe orienté dont les nœuds $N_0, \ldots, N_{p-1}$ correspondent aux instructions du programme et pour lequel il existe une arête de N_i vers N_j si et seulement si $j = i + 1$ (exécution séquentielle) ou $j \in$ ***suivants***(i) (i peut effectuer un saut direct en j). Autrement dit, une arête de N_i vers N_j indique que la $j^{\text{ème}}$ instruction de *prog* peut s'exécuter immédiatement après la $i^{\text{ème}}$ instruction. En particulier, si une instruction i effectue un saut direct à l'adresse $i + 1$, le graphe de flot de contrôle contient une seule arête de N_i vers N_{i+1}. La dernière instruction du programme est donc le seul nœud sans successeur.

Indication : On précise également que l'on ne traitera que des programmes de moins de $2^{22} = 4\,194\,304$ instructions.

Question à développer pendant l'oral 1 *Eu égard à la forme générale d'un programme, quelle représentation machine est la plus adaptée à l'implémentation du graphe de flot de contrôle ?*

On note $\mathcal{N}(g)$ l'ensemble des nœuds du graphe de flot de contrôle g, $|$***arêtes***$(g)|$ le nombre d'arêtes distinctes de g et ***succ***(N) l'ensemble des successeurs d'un nœud $N \in \mathcal{N}(g)$. On définit $CFG_p(n)$ comme étant le graphe de flot de contrôle correspondant au programme $(\mathcal{G}_p(n,i))_{i \in [\![0,p[\![}$. On a par exemple $|$***arêtes***$(CFG_{p=10}(n = 54))| = 15$, d'après le graphe représenté en Figure 2b.

Question 5 *Calculer* $|$***arêtes***$(CFG_p(n))| \bmod 101$ *pour*
a) $(p, n) = (10, 50)$ ***b)*** $(p, n) = (1000, 75)$ ***c)*** $(p, n) = (10536, 100)$

3 Analyse de durée de vie

On définit la notion de durée de vie d'une variable de la façon suivante :

— On dit qu'une variable r est activée par une instruction i si i lit une valeur dans la variable r (c'est-à-dire, $r \in$ ***lues***(i)). Dans ce cas, on dira que la variable r est vivante immédiatement avant i.
Dans l'exemple présenté en Figure 1c, la variable r_0 est ainsi activée par l'instruction $i = 1$.

— De façon duale, une variable r est tuée par une instruction i si i écrit une valeur dans la variable r (c'est-à-dire, $r \in$ ***définies***(i)).
Dans l'exemple présenté en Figure 1c, la variable r_0 est ainsi tuée par l'instruction $i = 3$.

— Une variable r est vivante immédiatement après l'instruction i si et seulement si elle est vivante immédiatement avant l'une des instructions succédant (après exécution séquentielle ou saut direct) à i.
Dans l'exemple présenté en Figure 1c, la variable r_2 est ainsi vivante immédiatement après l'instruction $i = 4$ car elle est vivante immédiatement avant $i = 5$.

— Si une instruction i n'active ni ne tue la variable r, alors r est vivante immédiatement avant i si et seulement si elle est vivante immédiatement après i.
Dans l'exemple présenté en Figure 1c, la variable r_2 est ainsi vivante immédiatement avant $i = 4$ car elle est vivante immédiatement après $i = 4$ et qu'elle n'est ni activée ni tuée par celle-ci.

i	X		Y		Z	
	$\boldsymbol{vivant}_{\text{avant}}(i)$	$\boldsymbol{vivant}_{\text{après}}(i)$	$\boldsymbol{vivant}_{\text{avant}}(i)$	$\boldsymbol{vivant}_{\text{après}}(i)$	$\boldsymbol{vivant}_{\text{avant}}(i)$	$\boldsymbol{vivant}_{\text{après}}(i)$
0	$\{r_2\}$	$\{r_0, r_2\}$	$\{r_2, r_3\}$	$\{r_0, r_2, r_3\}$	$\{r_2\}$	$\{r_0, r_2\}$
1	$\{r_0, r_2\}$	$\{r_1, r_2\}$	$\{r_0, r_2, r_3\}$	$\{r_1, r_2, r_3\}$	$\{r_0, r_2\}$	$\{r_1\}$
2	$\{r_1, r_2\}$	$\{r_1, r_2\}$	$\{r_1, r_2, r_3\}$	$\{r_1, r_2, r_3\}$	$\{r_1\}$	$\{r_1\}$
3	$\{r_1, r_2\}$	$\{r_0, r_2\}$	$\{r_1, r_2, r_3\}$	$\{r_0, r_2, r_3\}$	$\{r_1\}$	$\{r_0, r_2\}$
4	$\{r_0, r_2\}$	$\{r_0, r_2\}$	$\{r_0, r_2, r_3\}$	$\{r_0, r_2, r_3\}$	$\{r_0, r_2\}$	$\{r_0, r_2\}$
5	$\{r_2\}$	$\{\}$	$\{r_2\}$	$\{\}$	$\{r_2\}$	$\{\}$

Table 1 – Exemples et contre-exemple de durées de vie

On note $\boldsymbol{vivant}_{\text{avant}}(i)$ (respectivement, $\boldsymbol{vivant}_{\text{après}}(i)$) l'ensemble des variables vivantes immédiatement avant (resp., après) l'instruction i. Les définitions précédentes se traduisent formellement par un système d'équations définit sur le graphe de flot de contrôle g du programme :

$$\forall N_i \in \mathcal{N}(g), \begin{cases} \boldsymbol{vivant}_{\text{avant}}(i) = \boldsymbol{lues}(i) \cup (\boldsymbol{vivant}_{\text{après}}(i) \setminus \boldsymbol{définies}(i)) \\ \boldsymbol{vivant}_{\text{après}}(i) = \bigcup_{N_j \in \boldsymbol{succ}(N_i)} \boldsymbol{vivant}_{\text{avant}}(j) \end{cases} \quad (1)$$

On s'intéresse à la plus petite solution de ce système d'équations. Par exemple, Tableau 1 présente deux solutions, X et Y, du système d'équations précédent appliqué au programme donné en Figure 1c ainsi qu'une non-solution Z. La solution X est plus petite que Y : pour tout N_i, on a $X.\boldsymbol{vivant}_{\text{avant}}(i) \subseteq Y.\boldsymbol{vivant}_{\text{avant}}(i)$ et $X.\boldsymbol{vivant}_{\text{après}}(i) \subseteq Y.\boldsymbol{vivant}_{\text{après}}(i)$.

Question à développer pendant l'oral 2 *Expliquer pourquoi Z n'est pas une solution valide du système d'équations correspondant au programme présenté en Figure 1c.*

Afin de calculer la plus petite solution, on propose l'algorithme suivant, opérant sur le graphe de flot de contrôle g correspondant au programme considéré :

L1. (*Initialisation*). Pour tout $N_i \in \mathcal{N}(g)$, initialiser $\boldsymbol{vivant}_{\text{avant}}(i) = \emptyset$. $\boldsymbol{vivant}_{\text{après}}(i) = \emptyset$

L2. *(Copie).* Pour tout $N_i \in \mathcal{N}(g)$, sauvegarder les valeurs précédemment calculées :
$\boldsymbol{vivant'}_{\text{avant}}(i) \leftarrow \boldsymbol{vivant}_{\text{avant}}(i)$
$\boldsymbol{vivant'}_{\text{après}}(i) \leftarrow \boldsymbol{vivant}_{\text{après}}(i)$

L3. *(Itération).* Pour chaque nœud N_i de $\mathcal{N}(g)$, mettre à jour l'ensemble des variables vivantes avant i en assignant

$$\boldsymbol{vivant}_{\text{avant}}(i) \leftarrow \boldsymbol{lues}(i) \cup (\boldsymbol{vivant}_{\text{après}}(i) \setminus \boldsymbol{définies}(i))$$

puis mettre à jour l'ensemble des variables vivantes après i en assignant

$$\boldsymbol{vivant}_{\text{après}}(i) \leftarrow \bigcup_{N_j \in \boldsymbol{succ}(N_i)} \boldsymbol{vivant}_{\text{avant}}(j)$$

L4. *(Terminaison ?).* Si $\boldsymbol{vivant'}_{\text{avant}}(i) = \boldsymbol{vivant}_{\text{avant}}(i)$ et $\boldsymbol{vivant'}_{\text{après}}(i) = \boldsymbol{vivant}_{\text{après}}(i)$ pour toutes les instructions i du programme, alors l'algorithme termine : $\boldsymbol{vivant}_{\text{avant}}(-)$ et $\boldsymbol{vivant}_{\text{après}}(-)$ contiennent la plus petite solution du système d'équations.

L5. *(Convergence).* Sinon, retourner en L2. ▮

Question à développer pendant l'oral 3 *Prouver que cet algorithme termine et qu'il retourne effectivement la plus petite solution du système d'équations de durée de vie.*

Expliquer comment on peut (simplement) démontrer que X est la plus petite solution du système d'équations pour le programme présenté en Figure 1c.

Question à développer pendant l'oral 4 *Donner la complexité, dans le pire cas et exprimée en fonction du nombre de lignes p du programme, de l'algorithme présenté ci-dessus. Proposer (sans en justifier la complexité) des modifications à cet algorithme afin d'accélérer sa convergence.*

On note $\textit{\textbf{somme}}_{\text{vivant-avant}}(p, n, k)$ la somme des indices i des lignes $i \in [\![0, p[\![$ du programme $\textit{prog} = (\mathcal{G}_p(n, i))_{i \in [\![0,p[\![}$ pour lesquelles la $(k \bmod |\textit{\textbf{vars}}(\textit{prog})|)^{\text{ème}}$ variable de la liste $\textit{\textbf{vars}}(\textit{prog})$ est vivante avant i.

Indication : Pour les programmes considérés dans ce sujet, la solution est calculable en moins de 5 minutes. On remarque également que l'algorithme ne spécifie par l'ordre dans lequel les nœuds sont explorés : une implémentation devra le préciser.

Question 6 *Calculer* $\textit{\textbf{somme}}_{\textit{vivant-avant}}(p, n, u(n)) \bmod 101$ *pour*
a) $(p, n) = (20, 55)$ ***b)*** $(p, n) = (5059, 80)$ ***c)*** $(p, n) = (11091, 110)$

Lors de la compilation de l'assembleur vers une architecture matérielle spécifique, les variables utilisées par le programme assembleur doivent être traduites vers les registres matériels du processeur. Le nombre de registres matériels étant limité, on cherche à en utiliser le moins possible. Pour un programme donné, on dit que deux variables r_a et r_b interfèrent s'il est impossible de traduire ces deux variables vers un même registre matériel sans risquer de modifier le comportement du programme.

Par exemple, les variables r_3 et r_4 interfèrent dans le programme présenté en Figure 2a car r_4 est définie en $i = 6$ tandis que r_3 sera utilisée en $i = 7$: les deux variables ne peuvent donc pas être affectées au même registre matériel sans quoi l'une écraserait l'autre. Il en est de même pour les couples de variables (r_2, r_3), (r_2, r_4), (r_3, r_4) et (r_3, r_5). Tous les autres couples de variables n'interfèrent pas, comme par exemple le couple (r_2, r_5) : on peut donc les affecter à un même registre matériel. Il est ainsi possible de compiler ce programme vers une architecture disposant de seulement 3 registres `RAX`, `RBX` et `RCX` grâce, par exemple, à l'affectation suivante

$$r_2 \mapsto \texttt{RAX} \qquad r_3 \mapsto \texttt{RBX} \qquad r_4 \mapsto \texttt{RCX} \qquad r_5 \mapsto \texttt{RAX}$$

Question à développer pendant l'oral 5 *Étant donné le résultat de l'analyse de durée de vie, formuler un critère (simple) permettant de déterminer si deux variables interfèrent. Donner sa complexité en fonction de la taille du programme.*

On note $\max_{\text{interf}}(p, n)$ l'indice de la variable interférant avec le plus grand nombre de variables pour le programme $(\mathcal{G}_p(n, i))_{i \in [\![0,p[\![}$. Si plusieurs variables satisfont cette condition, on choisira celle d'indice minimal. Il s'agit de r_2 pour le programme présenté en Figure 2a : on a donc $\max_{\text{interf}}(10, 54) = 2$.

Question 7 *Calculer* $\left(\sum_{k=n}^{k=n+l} \max_{\textit{interf}}(p, k)\right) \bmod 101$ *pour*
a) $(p, n, l) = (50, 55, 20)$ ***b)*** $(p, n, l) = (5023, 110, 10)$ ***c)*** $(p, n, l) = (11342, 240, 5)$

4 Élimination de code mort

Toutes les instructions de notre assembleur idéalisé sont dites pures à l'exception

— des instructions n'ayant pas de variable de destination, de la forme

$$i: \quad [] \overset{op}{\leftarrow} srcs \Rightarrow jmp$$

quels que soient *srcs* et *jmp* ;

— des instructions effectuant un saut direct non séquentiel, de la forme

$$i: \quad dst \overset{op}{\leftarrow} srcs \Rightarrow [j]$$

avec $j \neq i+1$ et quels que soient *dst* et *srcs*.

On considère qu'une instruction pure n'a pas d'autre effet que de modifier sa variable de destination. Inversement, une instruction impure peut, par exemple, interagir avec un périphérique.

Une instruction est du code mort si celle-ci est pure et que la variable qu'elle modifie n'a aucune influence sur les exécutions possibles du programme : elle peut dont être remplacée par l'instruction NOP. Par exemple, l'instruction $i = 3$ du programme présenté en Figure 2a est du code mort. En effet, l'assignation à r_3 est écrasée par l'assignation effectuée en $i = 5$ et aucune lecture de r_3 ne peut avoir lieu entre temps, quelque soit le flot d'exécution du programme.

Question à développer pendant l'oral 6 *Donner un algorithme qui, étant donnés un programme et une instruction de ce programme, permet de déterminer si cette instruction est du code mort. Préciser sa complexité, exprimée en fonction de la taille du programme.*

On note $\min_{\text{instr}}(p, n)$ l'indice de la première instruction de $(\mathcal{G}_p(n,i))_{i \in [\![0,p[\![}$ qui est du code mort.

Question 8 *Calculer* $\left(\sum_{k=n}^{n+l} \min_{\text{instr}}(p,k)\right) \bmod 101$ *pour*

a) $(p,n,l) = (20, 55, 10)$ ***b)*** $(p,n,l) = (2142, 80, 20)$ ***c)*** $(p,n,l) = (5163, 110, 10)$

5 Élimination de code mort, itérée

Ayant identifié une instruction comme étant du code mort, on peut alors remplacer celle-ci par l'opération NOP. Cette transformation du programme a potentiellement pour effet de révéler que d'autres instructions sont également du code mort. Par exemple, la suppression de l'unique instruction morte $i = 3$ du programme de gauche

$$\begin{array}{l} i=0:[r_0] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=1:[r_1] \overset{op}{\leftarrow} [r_0] \Rightarrow [] \\ i=2:[r_2] \overset{op}{\leftarrow} [r_1] \Rightarrow [] \\ i=3:[r_3] \overset{op}{\leftarrow} [r_2] \Rightarrow [] \\ i=4:[r_3] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=5:[\ \ ] \overset{op}{\leftarrow} [r_3] \Rightarrow [] \end{array} \Rightarrow \begin{array}{l} i=0:[r_0] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=1:[r_1] \overset{op}{\leftarrow} [r_0] \Rightarrow [] \\ i=2:[r_2] \overset{op}{\leftarrow} [r_1] \Rightarrow [] \\ i=3:[\ \ ] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=4:[r_3] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=5:[\ \ ] \overset{op}{\leftarrow} [r_3] \Rightarrow [] \end{array} \Rightarrow \begin{array}{l} i=0:[\ \ ] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=1:[\ \ ] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=2:[\ \ ] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=3:[\ \ ] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=4:[r_3] \overset{op}{\leftarrow} [\ \ ] \Rightarrow [] \\ i=5:[\ \ ] \overset{op}{\leftarrow} [r_3] \Rightarrow [] \end{array}$$

révèle que l'instruction $i = 2$ est également du code mort dans le programme optimisé (au centre) et ainsi de suite jusqu'à l'instruction $i = 0$ (à droite). Dans cette partie, on s'attache à supprimer *tout* le code mort d'un programme donné. On aborde ce problème en trois étapes : on étudie tout d'abord l'élimination d'écritures (section 5.1) puis l'élimination de lectures (section 5.2) et enfin l'algorithme d'élimination de code mort itéré (section 5.3). Dans chaque cas, on suppose donné le graphe de flot de contrôle du programme que l'on souhaite transformer.

5.1 Élimination d'écritures

On définit une suite de programmes $(\mathcal{G}_p^{\mathrm{W}}(n,k))_{n\geqslant 10, k\in\mathbb{N}}$ par

$$\mathcal{G}_p^{\mathrm{W}}(n,k) = \begin{cases} (\mathcal{G}_p(n,i))_{i\in[\![0,p[\![} & \text{si } k=0 \\ \boldsymbol{\mathit{supprime\text{-}ecrit}}(\mathcal{G}_p^{\mathrm{W}}(n,k-1), u(n+k-1) \bmod p) & \text{si } k>0 \end{cases}$$

où $\boldsymbol{\mathit{supprime\text{-}ecrit}}(\mathit{prog}, i)$ remplace la $i^{\text{ème}}$ instruction

$$\mathit{dst}_i \overset{\mathit{op}}{\leftarrow} \mathit{srcs}_i \Rightarrow \mathit{jmp}_i$$

de *prog* par l'instruction

$$[] \overset{\mathit{op}}{\leftarrow} \mathit{srcs}_i \Rightarrow \mathit{jmp}_i$$

On note $\boldsymbol{\mathit{somme}}^{W}_{\text{reg-vivant}}(p,n,k)$ la somme, pour toutes les instructions $i \in [\![0,p[\![$, des indices de toutes les variables vivantes avant i pour le programme $\mathcal{G}_p^{\mathrm{W}}(n,k)$.

Question 9 *Calculer* $\left(\sum_{k=1}^{l} \boldsymbol{\mathit{somme}}^{W}_{\text{reg-vivant}}(p,n,k)\right) \bmod 101$ *pour*
a) $(p,n,l) = (10342, 80, 1000)$ ***b)*** $(p,n,l) = (10342, 230, 1100)$ ***c)*** $(p,n,l) = (10342, 350, 1200)$

Question à développer pendant l'oral 7 *Décrire l'algorithme que vous avez mis en œuvre pour répondre à la question précédente.*

5.2 Élimination de lectures

On définit une suite de programmes $(\mathcal{G}_p^{\mathrm{R}}(n,k))_{n\geqslant 10, k\in\mathbb{N}}$ par

$$\mathcal{G}_p^{\mathrm{R}}(n,k) = \begin{cases} (\mathcal{G}_p(n,i))_{i\in[\![0,p[\![} & \text{si } k=0 \\ \boldsymbol{\mathit{supprime\text{-}lu}}(\mathcal{G}_p^{\mathrm{R}}(n,k-1), u(n+k-1) \bmod p) & \text{si } k>0 \end{cases}$$

où $\boldsymbol{\mathit{supprime\text{-}lu}}(\mathit{prog}, i)$ remplace la $i^{\text{ème}}$ instruction

$$\mathit{dst}_i \overset{\mathit{op}}{\leftarrow} \mathit{srcs}_i \Rightarrow \mathit{jmp}_i$$

de *prog* par l'instruction

$$\mathit{dst}_i \overset{\mathit{op}}{\leftarrow} [] \Rightarrow \mathit{jmp}_i$$

On note $\boldsymbol{\mathit{somme}}^{R}_{\text{reg-vivant}}(p,n,k)$ la somme, pour toutes les instructions $i \in [\![0,p[\![$, des indices de toutes les variables vivantes avant i pour le programme $\mathcal{G}_p^{\mathrm{R}}(n,k)$.

Question 10 *Calculer* $\left(\sum_{k=1}^{l} \boldsymbol{\mathit{somme}}^{R}_{\text{reg-vivant}}(p,n,k)\right) \bmod 101$ *pour*
a) $(p,n,l) = (5200, 80, 1000)$ ***b)*** $(p,n,l) = (5200, 230, 1500)$ ***c)*** $(p,n,l) = (5200, 350, 2000)$

Question à développer pendant l'oral 8 *Décrire l'algorithme que vous avez mis en œuvre pour répondre à la question précédente. Donner également sa complexité, exprimée en fonction de la taille du programme d'entrée p et du nombre de lectures supprimées l.*

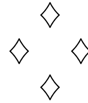
5.3 Élimination de code mort itérée

Étant donné un programme $(\mathcal{G}_p(n,i))_{i\in[\![0,p[\![}$, on obtient le programme $(\mathcal{G}_p^\dagger(n,i))_{i\in[\![0,p[\![}$ en remplaçant chacune des instructions mortes par un NOP. Il s'agit donc d'itérer l'élimination de code mort jusqu'à ce que *tout le code mort* ait été supprimé. On remarque que le nombre d'instructions reste inchangé lors de cette transformation : on laisse à une hypothétique phase d'optimisation ultérieure la tâche d'éliminer complètement les NOPs du programme.

On note $\boldsymbol{total}_{\mathrm{NOP}}(p,n)$ le nombre total de NOP dans le programme $(\mathcal{G}_p^\dagger(n,i))_{i\in[\![0,p[\![}$.

Question 11 *Calculer* $\boldsymbol{total}_{\mathrm{NOP}}(p,n) \bmod 101$ *pour*
a) $(p,n) = (7134, 160)$ ***b)*** $(p,n) = (8134, 260)$ ***c)*** $(p,n) = (9073, 360)$

Question à développer pendant l'oral 9 *Dans ce sujet, nous avons développé des analyses sur un langage assembleur idéalisé (comme en Figure 1c, par exemple) pour lequel nous n'avions pas connaissance des opérations effectivement exécutées par l'ordinateur (contrairement au programme considéré en Figure 1b, par exemple). Dans le cas général, est-il possible d'exploiter la connaissance des opérations effectivement exécutées pour améliorer la précision de nos analyses ?*



Fiche réponse type: Analyses de programmes

$\widetilde{u_0}$: 42

Question 1

a) 62

b) 0

c) 62

Question 2

a) 1

b) 3

c) 2

Question 3

a) $[r4], [r2, r3], [27]$

b) $[r4], [r3, r2], []$

c) $[r3], [r1, r1], [26]$

Question 4

a) (4, 3)

b) (7, 49)

c) (36, 24)

Question 5

a) 14

b) 83

c) 25

Question 6

a) 0

b) 46

c) 77

Question 7

a) 72

b) 62

c) 26

Question 8

a) 71

b) 72

c) 89

Question 9

a) 39

b) 72

c) 68

Question 10

a) 97

b) 2

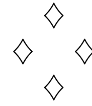
c) 53

Question 11

a) 23

b) 69

c) 34



Fiche réponse: Analyses de programmes

Nom, prénom, u_0: ....

Question 1

a)

b)

c)

Question 2

a)

b)

c)

Question 3

a)

b)

c)

Question 4

a)

b)

c)

Question 5

a)

b)

c)

Question 6

a)

b)

c)

Question 7

a)

b)

c)

Question 8

a)

b)

c)

Question 9

a)

b)

c)

Question 10

a)

b)

c)

Question 11

a)

b)

c)