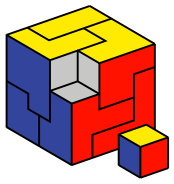
Prologin 2025



Concours national d'informatique

Épreuve écrite d'algorithmique
Polytechnique, Lausanne, Nice, Bordeaux, Lyon

23 février 2025

Prolotigre

Préambule

Bienvenue à **Prologin**. Ce sujet est l'épreuve écrite d'algorithmique et constitue la première des trois parties de votre épreuve régionale, sa durée est de 3 heures. Pendant cette épreuve, vous passerez un entretien (10 minutes) avec un organisateur. Ensuite, vous aurez une épreuve de programmation sur machine (3 heures et 30 minutes) l'après-midi.

Conseils

— Lisez bien tout le sujet avant de commencer.
— **Soignez la présentation** de votre copie.
— N'hésitez pas à poser des questions.
— Si vous avez fini en avance, relisez bien.
— N'oubliez pas de passer une bonne journée.

Remarques

— Le barème est donné à titre indicatif uniquement.
— Indiquez lisiblement vos nom et prénom, la ville où vous passez l'épreuve et la date en haut de votre copie.
— Lorsqu'un algorithme est demandé, vous pouvez le décrire avec suffisamment de précision, le pseudo-coder ou l'implémenter dans le langage de votre choix. Dans le dernier cas, veuillez néanmoins préciser le langage que vous utilisez.
— Ce sont des humains qui lisent vos copies : laissez une marge, aérez votre code, ajoutez des commentaires (**seulement** lorsqu'ils sont nécessaires) et évitez au maximum les fautes d'orthographe.
— Le barème récompense les algorithmes les plus efficaces : écrivez des fonctions qui trouvent la solution le plus rapidement possible.
— Si vous trouvez le sujet trop simple, relisez-le, réfléchissez bien, puis dites-le-nous, nous pouvons ajouter des questions plus difficiles.

À propos du sujet

Ce sujet est composé de 2 parties indépendantes :
— Un tigre utile : Graphes de flot de contrôle et analyse de vivacité (50 points)
— Prolotigre_1 : Assignation simple statique (10 points)
— Problèmes de mémoire : Assignation des registres (20 points)

Si vous vous retrouvez bloqué sur une question, n'hésitez pas à passer à la suivante ou à demander de l'aide. **Les questions ne sont pas triées par ordre de difficulté**. Des questions simples se cachent, éparpillées dans tout le sujet.

Introduction

Joseph Codant est bien embêté. Il aime beaucoup le langage de programmation *Prolotigre*[1], mais il a tendance à écrire beaucoup *beaucoup* trop de code inutile.

Aidez le pauvre Joseph en déterminant quelles lignes de son programme sont inutiles et doivent être retirées.

1. Plus d'informations sur la syntaxe du *Prolotigre* se trouvent en annexe

1 Un tigre utile

Pour commencer, il nous faut définir ce qu'est une ligne inutile.
En Prolotigre, il existe 2 types de lignes :
— Les lignes à effet de bord [2] ;
— Les autres lignes.
Par exemple, dans le programme ci-dessous :

```
1 a := 2;
2 afficher 42;
3 b := a;
4 afficher b;
```

Les lignes 2 et 4 affichent quelque chose à l'écran. Elles ont donc un effet visible à l'extérieur du programme, ce sont des lignes à effet de bord.

1.1 Des variables définies

Dans cette section, on s'intéressera uniquement aux programmes sans structures conditionnelles. Les programmes contiendront alors uniquement des assignations de variables, et des lignes à effet de bord.

On va à présent donner une première définition de *ligne inutile* et *variable inutile*, que l'on raffinera par la suite du sujet :
— Une ligne avec effet de bord est toujours dite *utile*. Toutes les variables impliquées dans cette ligne sont alors également dites *utiles*.
— Une variable qui est déclarée une fois, mais qui n'apparaît plus jamais par la suite dans le programme est appelée *inutile*. Au contraire, une variable qui apparait au moins une seconde fois dans le programme est dite *utile*.
— Une ligne n'ayant aucun effet de bord est alors dite *inutile* si et seulement si elle ne contient que des variables inutiles.

Voyons un petit exemple :

```
1 a := 2;
2 b := a;
3 c := 3;
4 d := 5;
5 afficher c;
```

Ici, la variable `a` est utilisée pour définir la variable `b`, elle est donc utile. C'est aussi le cas de la variable `c` qui est utilisée à la ligne 5. La variable `b`, en revanche, est inutile car elle n'apparaît nulle part après sa définition. De même, la variable `d` n'apparaît jamais après sa définition, elle est également inutile. Seule la ligne 4 est considérée comme inutile, car seule la variable `d` apparaît dans la ligne, qui est une variable inutile.

La première technique pour déterminer l'utilité des variables est de calculer le nombre d'occurrence de chaque variable, c'est-à-dire le nombre de fois où chaque variable apparaît.

Par exemple, dans le programme suivant :

```
1 a := 1;
2 b := 2;
3 c := 1 + 1;
4 c := c + 1;
5 afficher b;
```

— la variable `a` apparaît à la ligne 1 ;
— la variable `b` apparaît aux lignes 2 et 5 ;
— la variable `c` apparaît aux lignes 3, et deux fois à la ligne 4.

2. Une ligne est dite à effet de bord (traduction littérale de l'anglais « *side effect* », dont le sens est plus proche d'*effet secondaire*) si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur. Par exemple, **`afficher`** cause un effet de bord, là où modifier une variable n'en cause pas.

Question 1 (1 point)

> Listez le nombre d'occurrences de chaque variable dans le programme ci-dessous :
>
> ```
> 1 a := 4;
> 2 a := a + 1;
> 3 b := 4;
> 4 c := a + b;
> ```

Question 2 (1 point)

> Indiquez alors dans ce même programme quelles variables et quelles lignes sont utiles.

Question 3 (2 points)

> Justifiez que notre première définition de variable inutile ne reflète pas toujours la réelle utilité d'une variable.

1.2 Des variables définies pour être utilisées

On va donc revoir notre définition de variable et de ligne utiles. Pour cela, il est nécessaire de faire la différence entre *définition* et *utilisation*.

Dans cette section, on s'intéresse toujours uniquement aux programmes sans structures conditionnelles. Les programmes contiendront alors uniquement des assignations de variables, et des lignes à effet de bord.

Introduisons un peu de terminologie. Chaque apparition d'une variable dans un programme est soit une *définition*, soit une *utilisation*.

— Lorsqu'on attribue une valeur à une variable, il s'agit d'une *définition* ;
— Dans tous les autres cas, il s'agit d'une *utilisation*.

Autrement dit, lors d'une déclaration de variable, la variable à gauche du `:=` est *définie*, et celles à droite sont *utilisées*. Dans une ligne à effet de bord, il n'y a que des variables utilisées.

Notez qu'il est possible qu'une même variable soit *définie* **et** *utilisée* sur une même ligne. Par exemple, la ligne `a := a + b;` définit `a` et utilise `a` et `b`.

Dans cette section, on considère maintenant qu'une variable est inutile seulement si elle n'est jamais *utilisée*. On considère alors maintenant qu'une ligne est *utile* si :

— la ligne possède un effet de bord, ou
— la ligne *définit* une variable utile.

Question 4 (1 point)

> Pour chaque ligne du programme ci-dessous, indiquez les variables définies et les variables utilisées :
>
> ```
> 1 a := 3;
> 2 b := a;
> 3 c := a + 1;
> 4 b := 3;
> 5 afficher c;
> ```

Question 5 (1 point)

> Dans le code précédent, quelles sont alors les variables utiles, et quelles sont les lignes utiles ?

Question 6 (2 points)

Trouvez un programme contenant une variable inutile selon notre nouvelle définition, mais utile selon la définition précédente.

Question 7 (2 points)

Proposez un algorithme pour lister les variables utiles d'un programme, selon la définition actuelle. Vous pouvez si vous le souhaitez compléter le pseudo-code suivant :

Algorithme 1 : Algorithme calculant les variables utiles

Entrées :
- N, le nombre de lignes dans le programme ;
- `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
- `util`, avec `util`$[i]$ la liste des variables utilisées à la ligne i.

Résultat : la liste des variables utiles

```
// Complétez le code
```

Question 8 (3 points)

Proposez à présent un algorithme qui renvoie la liste des lignes utiles du programme [3]. Vous pouvez vous aider du pseudo-code suivant :

Algorithme 2 : Algorithme calculant les lignes utiles

Entrées :
- N, le nombre de lignes dans le programme ;
- `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
- `util`, avec `util`$[i]$ la liste des variables utilisées à la ligne i ;

Résultat : la liste des lignes définissant des variables utiles

```
// Complétez le code
```

Question 9 (2 points)

Justifiez que notre nouvelle définition de ligne utile ne reflète pas toujours la réelle utilité d'une ligne de code.

1.3 Des variables définies pour être utilisées intelligemment

Pour un programme linéaire, c'est-à-dire un programme sans structure conditionnelle, on peut raffiner une nouvelle fois notre définition de variable utile :
- Une variable utilisée dans une ligne à effet de bord est toujours utile.
- Une variable utilisée dans la définition d'une variable utile est utile.
- Toute autre variable est inutile.

On conserve la définition de ligne utile de la section précédente, c'est-à-dire, une ligne utile est une ligne contenant un effet de bord, ou une ligne définissant une variable utile.

Question 10 (1 point)

Selon notre nouvelle définition de ligne et variable utiles, indiquez les variables et les lignes utiles dans le programme suivant :

```
1 a := 1;
2 b := a;
3 c := a;
4 d := c + 1;
5 afficher d;
```

3. On peut distinguer les lignes à effet de bord comme étant celles qui ne définissent aucune variable.

Question 11 (2 points)

> Trouvez un programme contenant une variable inutile selon notre nouvelle définition, qui aurait été considérée comme utile selon la définition précédente.

Question 12 (2 points)

> Trouvez, pour la troisième fois, une situation où notre définition de l'utilité d'une ligne ne réflète pas la réelle utilité de la ligne.

Question 13 (3 points)

> Pour un programme linéaire (c'est-à-dire, sans structures conditionnelles), proposez une meilleure définition de variable utile et ligne utile qui reflète réellement l'utilité d'une ligne.

Question 14 (4 points)

> Proposez une nouvelle fois un algorithme permettant de trouver toutes les lignes utiles, selon la nouvelle définition de ligne utile. Vous pouvez une nouvelle fois vous aider du pseudo-code suivant :
>
> **Algorithme 3 :** Algorithme calculant les lignes utiles
>
> **Entrées :**
> — N, le nombre de lignes dans le programme ;
> — `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
> — `util`, avec `util`$[i]$ la liste des variables utilisées à la ligne i ;
> **Résultat :** la liste des lignes définissant des variables utiles
> `// Complétez le code`

Question 15 (3 points)

> Proposez finalement un algorithme permettant de trouver toutes les variables utiles, selon la nouvelle définition de variable utile. Vous pouvez une nouvelle fois vous aider du pseudo-code suivant :
>
> **Algorithme 4 :** Algorithme calculant les variables utiles
>
> **Entrées :**
> — N, le nombre de lignes dans le programme ;
> — `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
> — `util`, avec `util`$[i]$ la liste des variables utilisées à la ligne i ;
> **Résultat :** la liste des variables utiles
> `// Complétez le code`

1.4 Des variables définies pour être possiblement utilisées intelligemment

Dans cette section, on s'intéresse enfin aux structures conditionnelles et aux boucles.[4] Voici un autre morceau du code de Joseph :

```
1 i := 42;
2 si i = 12 alors
3     afficher 1;
4 afficher 2;
```

La variable `i` n'étant jamais utilisée dans une ligne à effet de bord, elle serait considérée comme inutile dans notre définition précédente. Cependant, si on considère que toutes les variables utilisées dans les conditions sont utiles, on se retrouve avec le problème inverse :

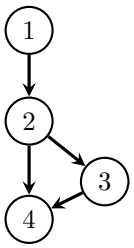
4. Pour la liste exacte des structures qui existent en Prolotigre, regardez l'annexe.

```
1 i := 42;
2 si i = 12 alors
3     i := i + 1;
4 afficher 2;
```

Dans ce nouveau code, la variable `i` devrait être considérée comme inutile, malgré le fait qu'elle soit utilisée dans une condition.

On va donc utiliser une information supplémentaire pour calculer nos variables inutiles en présence de structures conditionnelles : *le chemin du code*.

« Chemin du code » réfère aux chemins que le code peut prendre lors de l'exécution [5]. Cela est représenté à l'aide d'un *graphe de flot de contrôle* (ou GFC). Voici par exemple le GFC du code de Joseph ci-dessus :



Sur ce schéma, chaque cercle (plus communément appelé *nœud*) correspond à une ligne du code, avec comme valeur le numéro de ligne associé. Chaque flèche (aussi appelée *arc*) pointe vers la ligne qui va être exécutée après.

Dans le cas de la condition, il y a un arc dans le cas d'une condition vraie ($2 \to 3$) et un dans le cas d'une condition fausse ($2 \to 4$). Il est également nécessaire de dessiner les arcs de tous les cas de figure pour les boucles :

```
1 i := 40;
2 répète
3     i := i + 1;
4 tant_que i < 42;
5 afficher i;
```

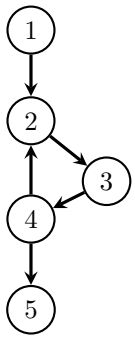


FIGURE 1 – GFC pour une boucle

Question 16 (2 points)

Dessinez le GFC pour le code *Prolotigre* suivant :

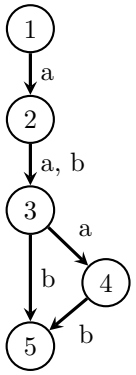
```
1 var := 4;
2 si var > 0 alors
3     répète
4         var := var * 2;
5     tant_que var < 1000;
6     var := var + 5;
7 afficher var;
```

Une fois ce graphe dessiné, il nous faut définir pour chacun des nœuds la vivacité des variables. On dit qu'une variable est vivante sur un arc s'il existe un chemin depuis cet arc jusqu'à une utilisation de la variable, en faisant attention à ne pas passer par une définition de cette même variable.

5. Cela ne prend pas en code la valeur des conditions. Même si une condition est toujours vraie ou toujours fausse, on considèrera toujours que le code peut rentrer dans les deux cas.

Dans l'exemple ci-dessous, la variable `a` est vivante sur l'arc $3 \to 4$ car il existe un chemin depuis une définition de `a` (ligne 1) allant vers une utilisation de `a` (ligne 4) passant par l'arc $3 \to 4$, sans jamais passer par une autre définition de `a` : $(1 \to 2 \to 3 \to 4)$. En revanche, il n'existe pas de chemin allant d'une définition à une utilisation de `b` passant par l'arc $3 \to 4$, et ne passant pas par une autre définition, la variable n'est donc pas vivante sur cet arc.

```
1  a := 2;
2  b := 3;
3  si a = 2 alors
4      b := a + 1;
5  afficher b;
```

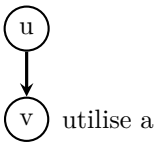


ligne	def	util
1	a	
2	b	
3		a
4	b	a
5		b

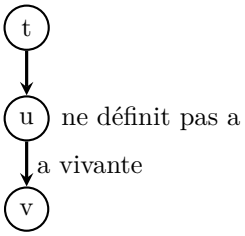
FIGURE 2 – Exemple de vivacité

De manière un peu plus formelle, voici les 2 propriétés permettant de déterminer si une variable est vivante à un nœud donné :

__ Soit 2 nœuds u et v adjacents. Une variable est vivante sur l'arc $(u \to v)$ si elle est utilisée dans v.



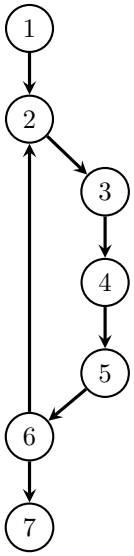
__ Soit un nœud t, tel qu'il existe un arc $(t \to u)$. Une variable est vivante sur l'arc $(t \to u)$ si elle est vivante sur l'arc $(u \to v)$ **et qu'elle n'est pas définie en u.**



Question 17 (2 points)

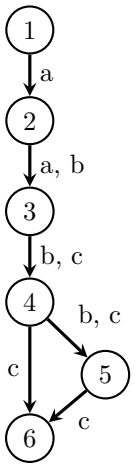
Listez les variables vivantes pour chaque arc du GFC ci-dessous. Nous vous conseillons de recopier le graphe sur votre copie.

```
1  a := 2;
2  répète
3      b := a - 1;
4      c := b * 2;
5      a := c + 1;
6  tant_que a < 5;
7  afficher b;
```



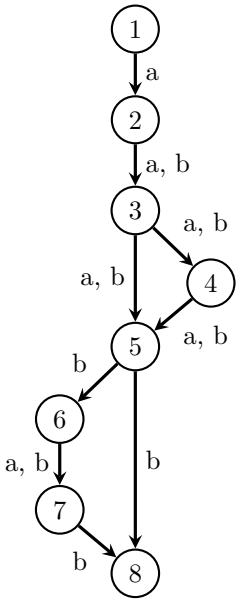
Question 18 (2 points)

Donnez un programme *Prolotigre* associé au GFC suivant :



Question 19 (2 points)

À l'aide du GFC ci-dessous, donnez les nœuds correspondants à une définition de la variable `a`. Listez également les nœuds utilisant obligatoirement `a`.



Question 20 (1 point)

Justifiez le fait qu'il est en général plus rapide de calculer la vivacité en partant de la dernière ligne du programme vers le début de celui-ci plutôt que dans l'ordre du programme.

Question 21 (5 points)

> Proposez un algorithme qui calcule la liste des variables vivantes pour chaque arc d'un GFC.
> On pourra également utiliser les fonctions suivantes :
> — Soit u, nœud d'un GFC. `succ(u)` renvoie tous les nœuds v tel qu'il existe un arc $u \to v$;
> — Soit u, nœud d'un GFC. `pred(u)` renvoie tous les nœuds v tel qu'il existe un arc $v \to u$;

Algorithme 5 : Algorithme calculant les variables vivantes de chaque arc

Entrées :
- N, le nombre de lignes du programme ;
- `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
- `util`, avec `util`$[i]$ la liste des variables définies à la ligne i ;

Résultat : Pour chaque arc $u \to v$, la liste des variables vivantes sur l'arc

```
// Complétez le code
```

Une fois la vivacité calculée, il nous est possible de déterminer les lignes définissant une variable inutile.

Une variable est dite utile si elle *participe* à un effet de bord : elle est utilisée sur une ligne à effet de bord [6], ou dans la définition d'une variable utile.

Par exemple :

```
1 a := 1;
2 b := a;
3 afficher b;
```

Dans le code ci-dessus, la variable `b` est utile car utilisée par une ligne à effet de bord. La variable `a` est également utile car permet de définir la variable `b`, elle-même utile.

Il nous est possible de déterminer cette notion à l'aide de la vivacité.

Voici un algorithme permettant de définir les définitions utiles :

Algorithme 6 : Algorithme listant les définitions utiles d'un programme

```
Fonction VarDefUtiles()
    P est une pile, contenant initialement toutes les lignes à effet de bord ;
    tant que P non vide faire
        ligne_2 ← depile(P) ;
        pour chaque variable var utilisée dans ligne_2 faire
            pour chaque ligne ligne_1 qui définit var et qui n'a pas encore été affiché faire
                si il existe un chemin de ligne_1 à ligne_2 où var est vivante sur tous les arcs du chemin
                 alors
                    afficher(ligne_1) ;
                    empiler ligne_1 dans P ;
                    // La ligne ligne_1 définissant une variable utile, il faut vérifier où sont
                    // définies les variables utilisées dans cette ligne.
                fin
            fin
        fin
    fin
fin
```

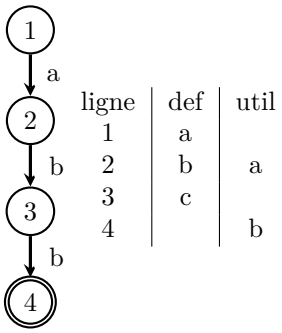
Question 22 (2 points)

> Donnez la complexité temporelle de la fonction `VarDefUtiles()` ci-dessus.

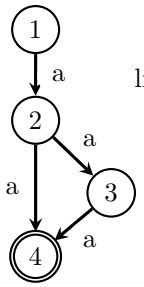
6. `afficher`

Question 23 (2 points)

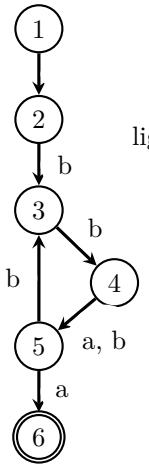
Pour chacun des GFC suivants, déterminez la ligne des définitions utiles. On considère que seulement la dernière ligne de chaque graphe est à effet de bord, notée avec un contour double.



ligne	def	util
1	a	
2	b	a
3	c	
4		b



ligne	def	util
1	a	
2		a
3	b	
4		a



ligne	def	util
1	a	
2	b	
3		
4	a	b
5		b
6		a

Question 24 (2 points)

Trouvez un code où l'algorithme 6 considère qu'une définition de variable est inutile, là où en réalité il n'est pas possible de supprimer la variable en question sans altérer le comportement du programme.

2 Prolotigre_1

Joseph est très heureux que vous l'ayez aidé, mais il cherche à obtenir une méthode plus efficace pour calculer les variables inutiles. Joseph a bien compris que la *redéfinition* des variables cause beaucoup de soucis pour déterminer les lignes inutiles, alors pour se débarasser de toutes les redéfinition, il veut utiliser « l'*assignation simple statique* ».

L'*assignation simple statique*[7] est une manière de structurer son programme afin de ne jamais redéfinir de variable.

Pour passer un programme en assignation simple statique, il nous faut renommer chacune des variables présentes dans le programme. Chaque variable définie est suivie du numéro de sa définition, et chaque variable utilisée prend le nom de sa définition. Par exemple, voici un petit programme avant et après renommage :

```
1 a := 2;
2 b := a + 1;
3 a := a + 4;
4 afficher b;
```

Listing 1 – Avant renommage

```
1 a_1 := 2;
2 b_1 := a_1 + 1;
3 a_2 := a_1 + 4;
4 afficher b_1;
```

Listing 2 – Après renommage

Question 25 (2 points)

Effectuez l'assignation simple statique pour le programme suivant :

```
1 a := 1 + 1;
2 b := a + 2;
3 b := a + 3;
4 b := b + 1;
5 c := a + b + b;
6 b := c + a;
7 a := 3;
8 afficher b;
```

Question 26 (3 points)

En considérant un programme linéaire[8], proposez un algorithme qui effectue l'assignation simple. Vous pouvez compléter l'algorithme suivant :

Algorithme 7 : Algorithme calculant le nouveau nom de chaque variable

Entrées :
- `def`, avec `def`$[i]$ la liste des variables définies à la ligne i ;
- `util`, avec `util`$[i]$ la liste des variables utilisées à la ligne i ;
- N, le nombre de lignes du programme

Résultat : Les listes `defs` et `util` modifiées avec les variables renommées

```
// Complétez le code
```

7. Static Single Assignment (ou SSA) en anglais.
8. un programme qui ne contient aucun **`si ... alors`** ni aucun **`répète ... tant_que`**.

2.1 Nœud-phi

Renommer les variables pose problème lorsqu'on modifie leur valeur dans une boucle ou une condition :

```
1 i_1 := 0;
2 si i_1 = 0 alors
3     i_2 := 1;
4 afficher i_?
```

Pour indiquer qu'il faut choisir une variable en fonction de l'exécution, on ajoute une nouvelle variable ainsi qu'un *nœud-phi* :

```
1 i_1 := 0;
2 si i_1 = 0 alors
3     i_2 := 1;
4 i_3 := Φ(i_1, i_2); // on prend soit i_1 soit i_2
5 afficher i_3;
```

La fonction Φ renvoie la variable correspondant à la version de la variable depuis laquelle on provient, selon si la condition du `si` était vraie ou non. De même pour les boucles :

```
1 i := 0;
2 répète
3     i := i + 1;
4 tant_que i = 0;
5 afficher i;
```

```
1 i_1 := 0;
2 répète
3     i_2 = Φ(i_1, i_3);
4     // Si i_3 n'est pas défini, phi renvoie i_1.
5     i_3 := i_2 + 1;
6 tant_que i_3 = 0;
7 afficher i_3;
```

Question 27 (2 points)

> Renommez et ajoutez les nœud-phis dans le programme ci-dessous :
>
> ```
> 1  a := 1;
> 2  b := 1;
> 3  c := 0;
> 4  répète
> 5      si b < 20 alors
> 6          b := a;
> 7          c := c + 1;
> 8      si b >= 20 alors
> 9          b := c;
> 10         c := c + 2;
> 11 tant_que c < 100;
> 12 afficher b;
> ```

Question 28 (3 points)

> Donnez un algorithme pour retirer les lignes inutiles sur un programme avec variables renommées, et indiquez sa complexité.

3 Problèmes de mémoire

Joseph a fini d'écrire son programme, mais a un nombre d'emplacements mémoire limités pour chacune de ses variables.

Voici les 8 variables utilisées par son programme :

- `N` ;
- `tigre` ;
- `i` ;
- `Prologin` ;
- `liste` ;
- `a` ;
- `vivacité` ;
- `ligne`.

Malheureusement, il n'y a que 3 emplacements en mémoire pour les variables du programme de Joseph, et certaines d'entre elles sont utilisées en même temps, et doivent être stockées au même moment.

Voici les variables qui doivent être stockées au même moment :

- `N` et `tigre` ;
- `N` et `liste` ;
- `N` et `a` ;
- `N` et `ligne` ;
- `tigre` et `i` ;
- `tigre` et `ligne` ;
- `i` et `Prologin` ;
- `i` et `a` ;
- `i` et `vivacité` ;
- `Prologin` et `liste` ;
- `Prologin` et `ligne` ;
- `liste` et `a` ;
- `a` et `vivacité`.

Question 29 (1 point)

> En considérant les 3 emplacements mémoire $e1$, $e2$, et $e3$, assignez chaque variable à l'un des trois emplacements, de telle sorte à ce qu'aucune des variables stockées au même moment se retrouvent dans le même emplacement.

Question 30 (1 point)

> Représentez les contraintes des variables sous forme de graphe, ainsi que votre solution à la question précédente.

Joseph a oublié de prendre en compte quelque chose en calculant les emplacements : la copie.

Dans le programme ci-dessous, on dit que `c` est une copie de `b` :

```
1 a := 2;
2 b := a + 1;
3 c := b;
4 a := c - 1;
```

Dans ce types de programmes, la copie peut être remplacée par l'original :

```
1 a := 2;
2 b := a + 1;
3 a := b - 1;
```

Question 31 (1 point)

> Considérons les copies suivantes :
>
> 1. `a` est une copie de `N` ;
> 2. `liste` est une copie de `a` ;
> 3. `ligne` est une copie de `tigre` ;
> 4. `Prologin` est une copie de `i` ;
>
> Simplifiez votre réponse à la question précédente en fusionnant les copies dans le graphe.

Question 32 (2 points)

> En partant de notre graphe simplifié de la question précédente, supposons que les variables `tigre` et `vivacité` doivent être stockées en même temps, est-il possible de répartir les variables dans les trois registres ?

Question 33 (2 points)

> Si maintenant il s'avère que `tigre` et `vivacité` sont en fait des copies, est-il maintenant possible de répartir les variables dans les trois registres ?

Considérons un autre cas où le programme de Joseph contient 5 variables :
— `a`
— `b`
— `c`
— `d`
— `Jean_Eude_destructeur_de_mondes`

Pendant l'exécution du programme, on remarque que les paires de variables suivantes sont vivantes en même temps :
— `a` et `b`
— `a` et `c`
— `b` et `c`
— `d` et `a`
— `d` et `b`
— `Jean_Eude_destructeur_de_mondes` et `c`

Question 34 (1 point)

> Proposez une répartition des variables sur trois registres $e1$, $e2$ et $e3$.

Question 35 (2 points)

> Supposons maintenant que Joseph se rende compte que la variable d est utilisée en même temps que `Jean_Eude_destructeur_de_mondes`. Peut-il encore répartir les variables sur les trois registres ?

Question 36 (2 points)

> En fait, Joseph se rend compte que d est une copie de `Jean_Eude_destructeur_de_mondes`. S'il fusionne les deux variables en une seule, peut-il à présent répartir les variables sur les trois registres ?

Question 37 (2 points)

> Donnez une condition suffisante (la plus précise possible) sur les degrés des sommets qui indique qu'il est possible d'effectuer une fusion de deux nœuds.

Certains des programmes de Joseph n'ont toujours pas assez d'emplacements. Pour remédier au problème, il ajoute sur son ordinateur une pile de variables.

Cette pile peut contenir autant de variables que l'on souhaite, même si ces variables ne peuvent pas être dans le même emplacement.

Le problème, c'est que cette pile est assez lente, il faut donc stocker le moins de variables possibles dedans.

Voici la nouvelle liste de variables utilisée par Joseph :

- `a` ;
- `b` ;
- `c` ;
- `d` ;
- `e` ;
- `f` ;
- `g` ;
- `h` ;
- `i` ;
- `j`.

Et voici les variables ne pouvant pas être stockées dans le même emplacement :

- `a` et `b` ;
- `a` et `d` ;
- `a` et `e` ;
- `a` et `f` ;
- `a` et `g` ;
- `a` et `i` ;
- `b` et `c` ;
- `b` et `d` ;
- `b` et `f` ;
- `b` et `g` ;
- `b` et `i` ;
- `b` et `j` ;
- `c` et `d` ;
- `c` et `j` ;
- `d` et `e` ;
- `d` et `f` ;
- `d` et `i` ;
- `e` et `j` ;
- `f` et `g` ;
- `f` et `i` ;
- `g` et `i` ;
- `h` et `i`.

Question 38 (3 points)

> Donnez une répartition des variables sur 3 emplacements, ou la pile si nécessaire, en utilisant la pile le moins possible. Indiquez sur votre représentation les variables qui sont sur la pile.

Question 39 (3 points)

> En supposant que vous pouvez choisir quelles variables sont des copies, trouvez une répartition valide sans utiliser la pile et en utilisant le moins de fusion possible.

4 Questions Bonus

Question bonus 40 (1 point)

> Supprimez toutes les lignes inutiles de ce sujet.

Question bonus 41 (1 point)

> Donnez un programme *Prolotigre* dont le GFC est également un cercle d'incantation (toute représentation visuelle du cercle est la bienvenue).
> Détaillez précisément les effets de votre cercle d'incantation.

Question bonus 42 (1 point)

> Donnez le nom de la variable `a` ici présente après renommage SSA de toutes les variables du sujet.

Question bonus 43 (1 point)

> En considérant que 2 variables ne peuvent pas être dans le même emplacement si elles ont au moins 1 lettre en commun[9] donnez le nombre minimum d'emplacement mémoires requis pour stocker toutes variables du sujet, ainsi qu'une répartition pour chacun de ces emplacements.

9. sans prendre les majuscules en compte. Les variables `N` et `n` ne peuvent donc pas être dans le même emplacement.

A Syntaxe du Prolotigre

A.1 Syntaxe générale

Toute déclaration doit être suivie d'un point virgule « ; ».
Lors d'un appel de fonction, le nom de la méthode[10] est suivi des arguments de l'appel comme ceci :

```
afficher 12;
```

Dans le cas où il y a plusieurs arguments, ils sont séparés par des espaces.
Pour passer des arguments composés, on les entoure de parenthèses :

```
afficher (1 + 1) (2 + 2);
```

A.1.1 `afficher`

La méthode **`afficher`** affiche sur la sortie standard tous ses arguments, chacun séparés par des espaces.
Le code suivant :

```
afficher 1 2 3
```

affiche sur la sortie standard :

```
1 2 3
```

A.2 Définition de variable

On assigne une valeur à une variable comme suit :

```
a := 1
```

Assigner une valeur à une variable non définie revient à faire une définition de celle-ci.

A.3 Blocs conditionnels et comparaisons

Les blocs conditionnels suivent la syntaxe suivante :

```
si condition alors
    // contenu;
```

Il n'existe pas de bloc **`sinon`** en *Prolotigre*.
Le contenu du bloc se doit d'être indenté. L'indentation détermine la fin du bloc :

```
si condition alors
    // dans le bloc;
// en dehors du bloc;
```

Voici un tableau listant les différents opérateurs pour les conditions :

opérateur	action
=	égalité
!=	différence
<	strictement inférieur
>	strictement supérieur
<=	inférieur ou égal
>=	supérieur ou égal

10. la seule méthode qui existe en Prolotigre est **`afficher`**

A.4 Boucles

Le *Prolotigre* ne contient que des boucles **répète ... tant_que** :

```
répète
    // contenu;
tant_que condition;
```

Lorsqu'on arrive sur la ligne avec le **tant_que**, le programme reprend à partir du **répète** si la condition est vraie.

Tout comme les conditions, le contenu de la boucle est indenté.