Guillaume BOURGOIN
Rithy NGUON
Hassan TAOULALLOU

28/02/11

M2 Info

Algo Répartie
Devoir n°2

Exercice 1

Question 1.a

D'après la ligne 7 de l'algorithme, un processus ne retire sa candidature que s'il a tiré Pile et reçu Face. Donc pour qu'un processus candidat soit éliminé au cours d'une ronde r il faut qu'un autre candidat ait tiré Face lors de cette ronde r, et ce dernier ne sera pas éliminé, puisqu'il a tiré Face.
Donc si un processus est éliminé au cours d'une ronde, c'est qu'il en existe un autre qui lui ne l'est pas, donc s'il y a au moins un candidat au début de la ronde, il en reste toujours au moins un à la fin de la ronde.

Question 1.b

S'il y a au moins 2 candidats au début de la ronde r, alors il existe un candidat q qui se trouve "juste avant" p (c'est-à-dire le premier candidat qu'on trouve en visitant le prédécesseur de p, puis le prédécesseur du prédécesseur de p, et ainsi de suite).
Puisqu'on a au moins 2 candidats, p est évidemment différent de q (si en remontant les prédécesseurs depuis p on ne trouvait aucun candidat avant p, cela voudrait dire que p est le seul candidat, puisqu'on aurait visité tous les autres), les tirages de p et de q sont donc indépendants.
Au cours d'une ronde on a donc 4 tirages possibles pour p et q :

	p	q
1	Pile	Pile
2	Pile	Face
3	Face	Pile
4	Face	Face

Seul dans le cas du second tirage, où p tire Pile et reçoit Face, p est éliminé.
Ces 4 tirages sont équiprobables, p a donc une chance du 4 de le plus être candidat à la fin de la ronde.

Question 1.c

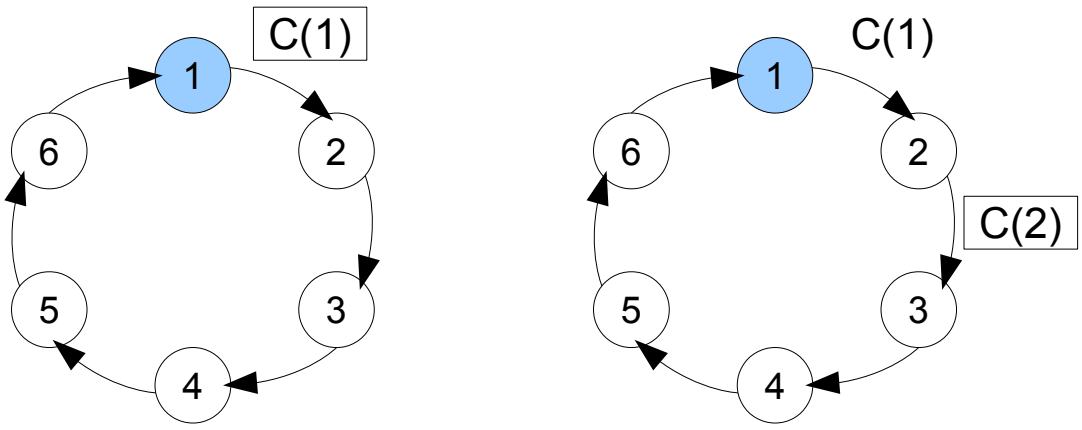
La probabilité qu'un processus ne soit pas éliminé après 1 ronde étant de 3/4, celle qu'il ne soit pas éliminé au bout de k rondes est de $(3/4)^k$ s'il reste encore plus d'un candidat.
Or $(3/4)^k$ tend vers 0 quand k tend vers l'infini.
Donc pour chaque processus la probabilité de rester candidat tend vers 0 tant qu'il reste plus d'un candidat. Donc la probabilité qu'il reste plus d'un candidat tend vers 0 quand le nombre de ronde tend vers l'infini.

Question 1.d

Comme un processus n'a pas d'identifiant et que tous ont le même code, un candidat ne peut pas simplement faire passer un jeton spécial et attendre de le récupérer pour prouver qu'il est le seul, car lorsqu'un jeton lui arrive il n'a aucun moyen de savoir si c'est son propre jeton qui lui revient (il serait alors le seul candidat) ou le jeton d'un autre candidat qui fait exactement la même chose (il ne serait donc pas seul).
En revanche, comme tous les processus connaissent la taille de l'anneau, ils peuvent utiliser cette information pour connaître l'origine du jeton qu'il reçoivent : pour savoir s'il est le seul candidat un processus envoie "C(1)", et lorsqu'un non-candidats reçoit C(i) il renvoie C(i+1). Ainsi, si le processus candidat reçoit C(n) il sait qu'il est le seul candidat, mais s'il reçoit C(j), avec j<n, alors il sait qu'il n'est pas le seul.

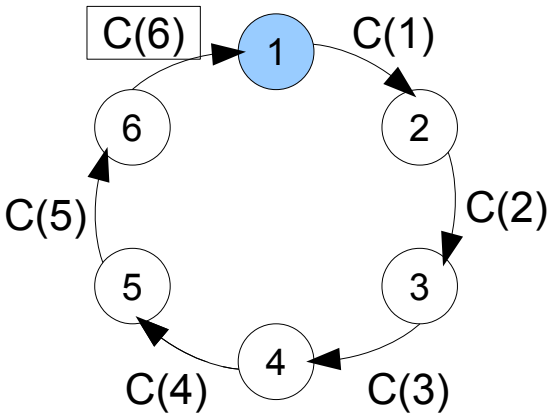
Par exemple, avec n=6 :

- Cas où 1 est le seul candidat :



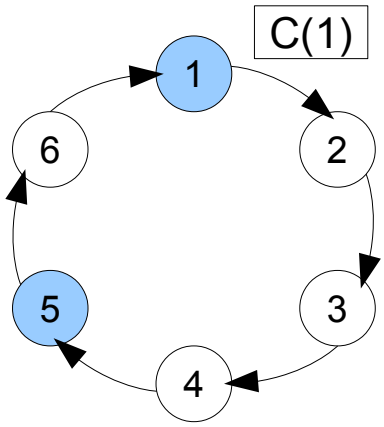
Remarque : Les processus n'ont pas d'identifiants, les numéros écrits sur les schémas se permettent uniquement de les désigner dans les explications.

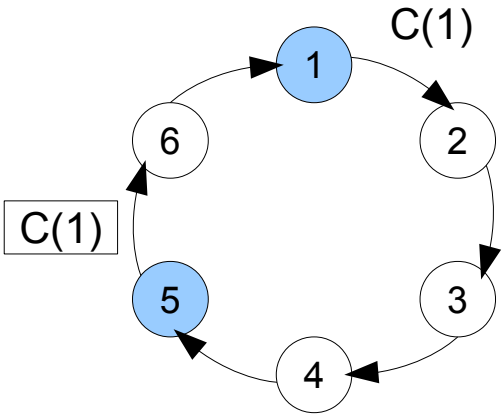
...quelques étapes plus tard :



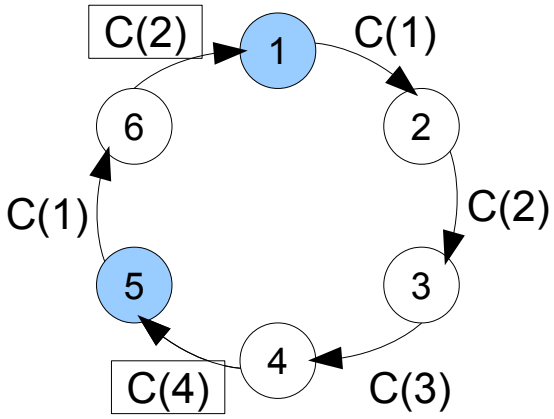
1 reçoit C(6) et 6=n : il sait qu'il est le seul candidat.

- Cas où 1 et 5 sont candidats :





...quelques étapes plus tard :



1 reçoit C(2) et 2<n : il sait qu'il n'est pas le seul candidat.
5 reçoit C(4) et 4<n : il sait qu'il n'est pas le seul candidat.

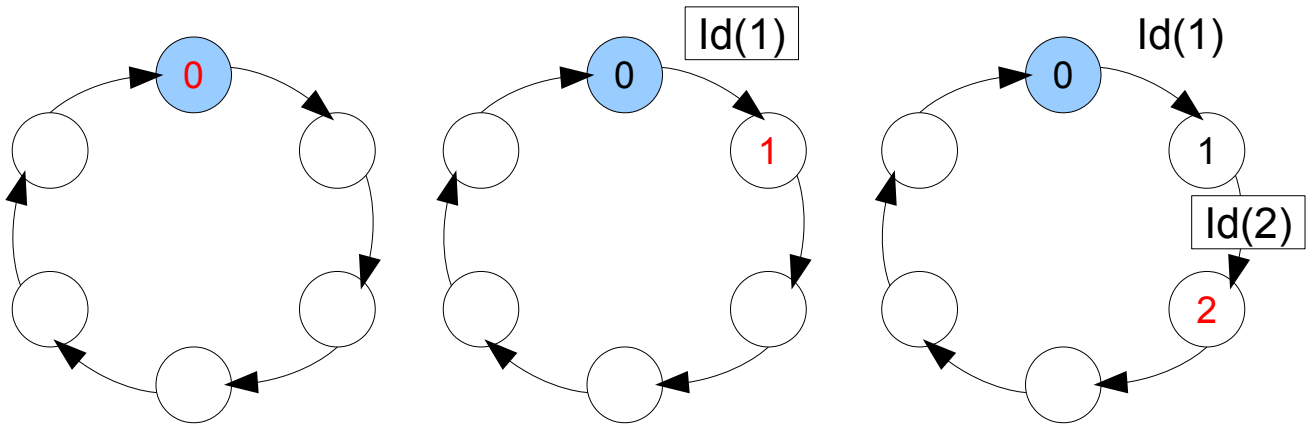
Pour écrire l'algo, utilisons une variable *Elu* qui prend la valeur True si le processus est le seul candidat et False sinon.

Algorithme :

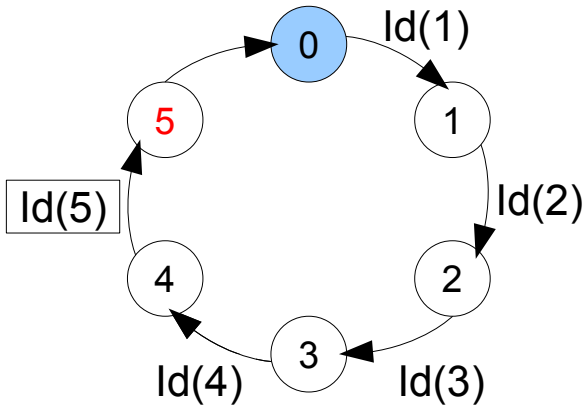
if $Candidat_p$
then
    send $C(1)$ **to** $Successeur_p$
    receive C(i) **from** $Predecesseur_p$
    if i = n **then** Elu_p := True **else** Elu_p := False
else
    receive C(i) **from** $Predecesseur_p$
    send $C(i+1)$ **to** $Successeur_p$

Question 1.e

On peut reprendre un algorithme assez similaire au précédent : le candidat unique s'attribue l'identifiant 0 puis fait passer un jeton de la forme Id(1), et chaque non candidat qui reçoit Id(i) prend l'identifiant i et envoie à son successeur Id(i+1).
De cette façon, les processus auront des identités allant de 0 à n-1.



...quelques étapes plus tard :



Remarque : Avec cet algorithme, 5 enverra un message Id(6) à 0 qui ne sera jamais reçu. On pourrait modifier l'algo pour que 0 reçoive ce message et le prenne comme une confirmation que tous les processus ont bien reçus leur identifiant.

Algorithme :

if $Candidat_p$
then
    $Identifiant_p$:= 0
    send *Id*(1) **to** $Successeur_p$
else
    receive *Id*(i) **from** $Predecesseur_p$
    $Identifiant_p$:= i
    send *Id*(i+1) **to** $Successeur_p$

Question 1.f

Les candidats pourraient par exemple tirer à Pile ou Face lors des rondes impaires et vérifier s'ils sont les seuls processus encore en activité lors des rondes paires.
Une fois qu'un processus se sait élu, il stoppe les autres de la même façon qu'il distribue les identifiant dans la question précédente (ça n'est pas demandé dans cette question mais on va en profiter pour leur donner des identités).

Ici, il serait assez long de donner un exemple d'exécution détaillé, on se contentera donc de donner l'algorithme avec quelques annotations.

(cf algo page suivante)

Dans cet algorithme les candidats jouent à Pile ou Face une ronde sur 2 (parties A.3 et B.3), donc on finira statistiquement par avoir un unique candidat (un "élu") comme montré dans les questions précédentes.
S'il ne reste plus qu'un processus candidat, celui-ci le saura assez vite, puisque tout candidat teste s'il est le seul une ronde sur 2 (parties A.2 et B.2).
Lorsqu'un processus sait qu'il est l'élu il s'arrête après avoir transmis un signal qui stoppe également les autres processus non candidats (parties A.1 et B.1).
Donc cet algorithme respecte bien les propriétés (1) et (2) : il termine s'il ne reste plus qu'un processus candidat, et la probabilité que cela se produise tend vers 1 quand le nombre *r* de rondes tend vers l'infini.

Algorithme :

```
Continue_p := True
Elu_p := False
r := 0

while Continue_p

    if Candidat_p                          //A) candidat
    then
        if Elu_p                           //A.1) candidat élu (arrêt)
        then
            Identifiant_p := 0
            send Id(1) to Successeur_p
            Continue_p := False
        else
            if (r modulo 2) = 1            //A.2) candidat qui teste s'il est le seul
            then
                send C(1) to Successeur_p
                receive C(i) from Predecesseur_p
                if i = n then Elu_p := True
            else                           //A.3) candidat qui joue à Pile ou Face
                V_p := tirer( )
                send V_p to Successeur_p
                receive W from Predecesseur_p
                if (V_p = Pile and W = Face) then Candidat_p := False

    else                                   //B) non-candidat
        receive W from Predecesseur_p
        if W = Id(i)                       //B.1) non-candidat qui transmet un identifiant (arrêt)
        then
            Identifiant_p := i
            send Id(i+1) to Successeur_p
            Continue_p := False
        else
            if W = C(i)                    //B.2) non-candidat qui transmet un compteur
            then
                send C(i+1) to Successeur_p
            else                           //B.3) du non-candidat qui transmet une pièce
                send W to Successeur_p

    r := r+1
```

Exercice 2

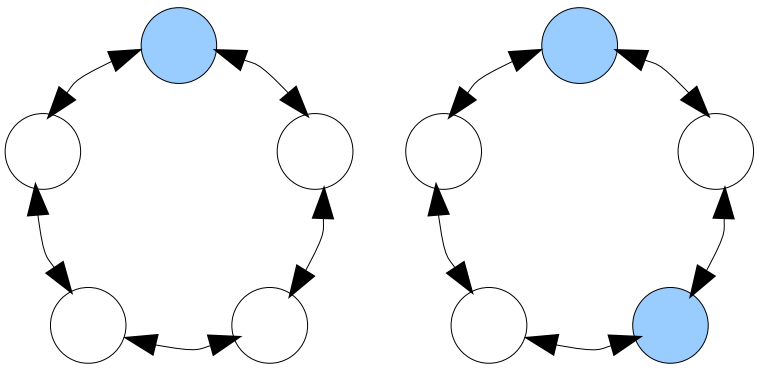
Question 2.a

Dans le cas d'un anneau non orienté avec 3 processus, chacun communique avec tout le monde, il est donc facile de désigner un candidat unique : le premier candidat qui parle décide qu'il est "élu", le ou les autres candidats le verront en s'exécutant et ne seront plus candidat.
Insérons une variable *élu* dans l'algo qui est initialisée à False. Quand un processus s'exécute, s'il est candidat, et si aucun de ses 2 voisins n'est élu, alors il passe sa variable *élu* à True, sinon, il passe sa variable *Candidat* à False.

Algorithme :

if $Candidat_p$
then
    if $Successeur_p.Elu$ = True **or** $Predecesseur_p.Elu$ = True
    then
        $Candidat_p$ = False
    else
        Elu_p = True

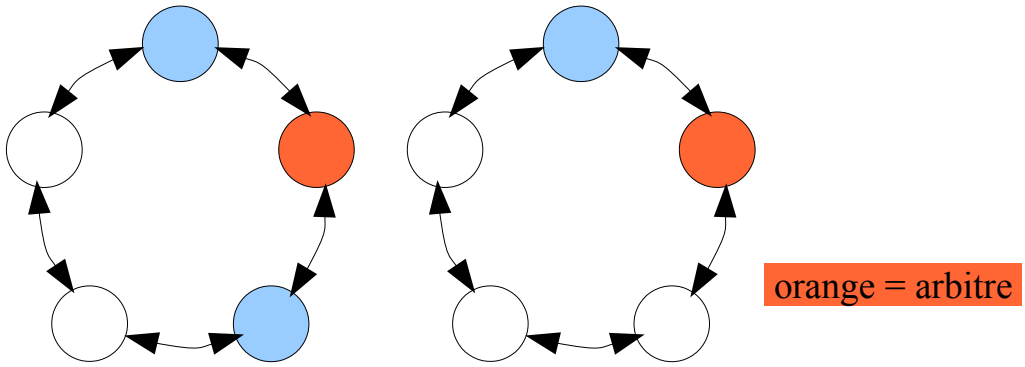
Le cas n=5 est un peu plus compliqué, car ici aucun des processus ne peut voir tous les autres. Essayons tout de même d'appliquer notre algorithme précédent :
Comme chaque élu interdit à ses voisin de rester candidat, quand tout le monde s'est exécuté une fois, on se trouve forcement dans une de ces 2 situations :



bleu = candidat

Ici, il peut donc y avoir 2 processus dont élu=True, par souci de respect de la sémantique nous allons donc renommer cette variable "primaire" (il ne sont pas encore élus mais ils ont gagné les élections primaires).
Dans le premier cas, on a réussi : il n'y a plus qu'un seul processus qui soit candidat.

Dans le second cas, il y a un processus qui se trouve dans une position unique : celui qui est entre les 2 pré-élus. On pourrait lui demander de tirer au sort l'un des 2 autres, ou, encore plus simple, on pourrait choisir de toujours élire son prédécesseur. Il suffirait donc de prendre une variable *arbitre* qui est initialisée à False pour tous, lorsqu'un processus se trouve entre 2 pré-élus il passe *arbitre* à True, et lorsqu'un pré-élue se trouve après un arbitre il n'est plus candidat.



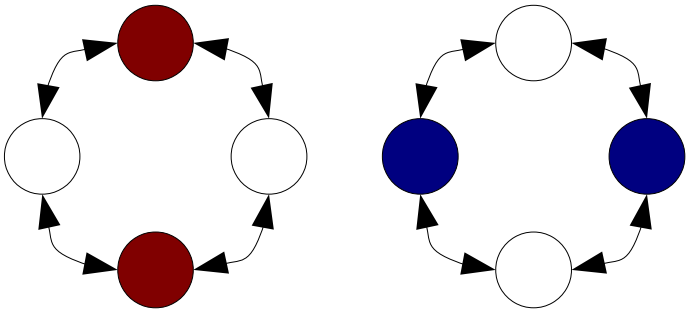
Algorithme :

for r := 0 **to** ∞ **do**
    if $Candidat_p$
    then
        if $Successeur_p.Primaire$ = True **or** $Predecesseur_p.Primaire$ = True
        then
            $Candidat_p$ = False
        else
            $Primaire_p$ = True
        if $Predecesseur_p.Arbitre$ = True
        then
            $Candidat_p$ = False
    else
        if $Successeur_p.Primaire$ = True **and** $Predecesseur_p.Primaire$ = True
        then
            $Arbitre_p$ = True

Remarque : le second algo n'enlève rien par rapport au premier, il marche aussi pour n=3.

Question 2.b.i

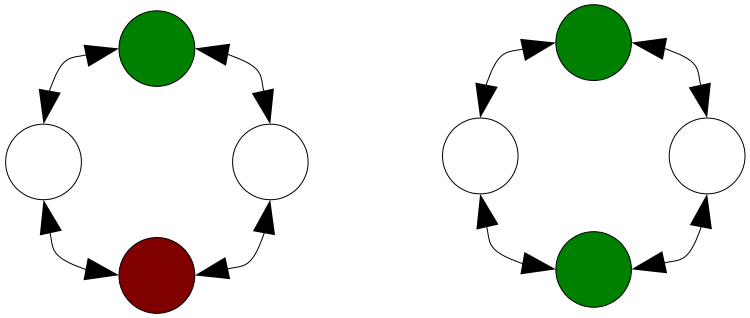
Sur un anneau de 4 processus, on peut trouver 2 ensembles de 2 états qui respectent les conditions de symétries :



À chaque fois il faut que les processus blancs et les colorés soient dans le même état au départ, c'est-à-dire candidat ou pas, donc ces 2 exemples sont en fait identiques.

Considérons le premier schéma comme notre configuration γ : les 2 rouges sont donc au départ dans le même état (disons par exemple qu'ils sont candidats) et les 2 blancs sont aussi dans le même état, puisque le successeur du rouge du haut doit être dans le même état que le successeur du rouge du bas, etc (disons par exemple qu'ils ne sont pas candidat).
Donc pour atteindre une configuration γ' telle que (1) les rouges soient toujours symétrique, (2) les blancs ne changent pas d'états et (3) les rouges s'exécutent au moins une fois, c'est très simple, il suffit de considérer le cas d'exécution où seuls les rouges ont effectué leur algorithme (qu'on ne connaît pas).
On se retrouve alors dans cette situation là :

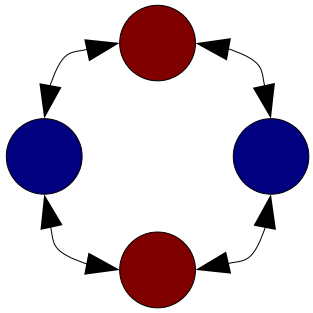
celui du haut s'exécute puis celui du bas:



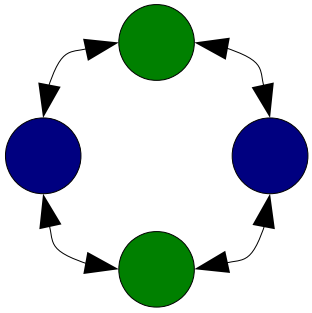
On change la couleur des 2 processus qui se sont exécutés pour mettre en évidence le fait qu'ils ne sont pas forcement dans le même état qu'avant de s'exécuter, leur code disait peut-être "*ne soit plus candidat*" ou bien "*si tu es entouré par 2 non-candidats change l'une de tes variables*" mais quoi qu'il en soit ils sont forcement à nouveau tous les 2 dans le même état puisqu'il ont exécuté le même code avec les mêmes données (leurs variables étaient les mêmes ainsi que les variables de leur prédécesseurs et successeurs respectifs). Et comme on sait qu'il ne sont pas voisins (une des conditions d'un ensemble symétrique) l'exécution du 1er ne peut pas avoir d'impact sur l'exécution du second.

Question 2.b.ii

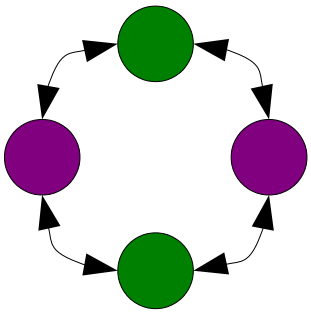
Prenons un anneau qu'on peut diviser en plusieurs sous-ensembles symétriques, par exemple l'anneau n=4 qu'on peut diviser en 2 sous-ensembles :



Si les 2 rouges s'exécutent, comme on l'a vu précédemment, les processus ne restent pas forcement dans le même état, mais cet ensemble reste symétrique :



À présent, si on exécute les 2 bleus, leur ensemble reste symétrique, toujours pour la même raison :



Et on peut continuer ainsi à l'infini : tant qu'on exécute chaque ensemble symétrique entièrement avant de passer à un autre, tous ces ensembles resteront symétrique.

Remarque : on peut même trouver une infinité d'ordre d'exécutions qui ne brisent pas la symétrie, on peut prendre n'importe quelle suite de ce type : Haut, Bas, Droite, Gauche, Bas, Haut, Bas, Haut, Gauche, Droite, Haut, Bas, Droite, Gauche, ...
À chaque fois on en exécute un au hasard puis on exécute les autres qui sont de la même couleur (ils peuvent être plusieurs quand *n* est plus grand que 4).

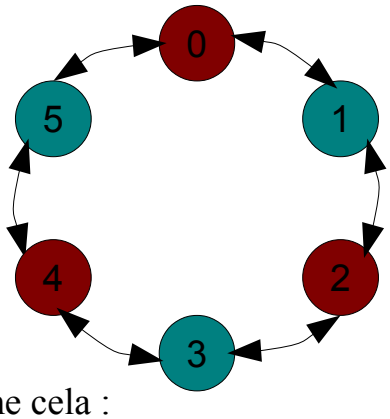
Question 2.b.iii

On peut facilement en déduire qu'il n'y a pas d'algorithme d'élection réellement fiable pour un anneau tel que n=4 si on ne sait rien de la répartition de départ des candidats ni de l'ordre dans lequel les processus vont s'exécuter : par exemple, si au départ les 2 rouges étaient candidats et le 2 bleus ne l'étaient pas, on aurait bien 2 ensembles symétriques, et comme on a montré que dans une telle situation il était possible que les 2 rouges restent toujours identiques en tout points, il n'y aura jamais un candidat unique durable (les 2 rouges seront tous les 2 candidats, ou tous les 2 non-candidats, infiniment souvent).

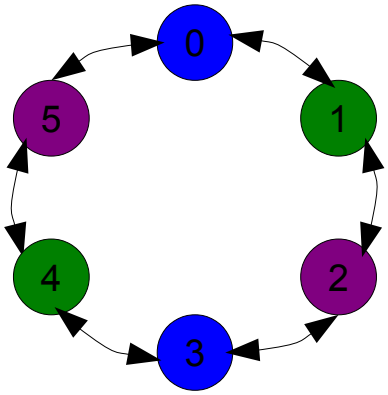
Question 2.b.iv

Si *n* n'est pas premier, on pourra toujours le diviser en sous-ensemble $L_1,...,L_k$ de la façon suivante (pour plus de clarté dans les explication, imaginons à nouveau que les processus sont numérotés, de 0 à n-1) :
On choisit un diviseur *d* de *n* (autre que *1* ou *n*) et on prend les processus divisibles par *d* pour former L_1, puis leurs successeurs respectifs (ceux qui congruent à 1 modulo *d*) pour former L_2, les successeurs de ces derniers pour former L_3, etc

Par exemple, pour n=6, on peut choisir d=2 et le diviser comme ceci :



Ou d=3 et le diviser comme cela :



Si au départ tous les processus partageant une même couleur (sur l'un de ces 2 schémas) sont dans le même état on risque de ne jamais pouvoir les départager.

THE END