Rappels et exercices[1]

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

[1] Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Informations pratiques

Informations pratiques

- Cours (Jeudi matin) et TD (Jeudi après-midi cette semaine, puis les mercredi après-midi) pendant 12 semaines.
- J'assurerai les cours.
- M. Benoît Hœssen assurera les TD.
- Eviter d'utiliser la $13^{\text{ème}}$ semaine. On les rattrapera avant
- 1 examen et 1 CC
- CPP (commission pédagogique et paritaire)
- benferhat@cril.fr

Pseudo langage et Langage C

- On utilisera le langage C (ou le pseudo langage) pour écrie les algorithmes.
- Il ne s'agit pas d'un cours sur le langage C.

Pseudo langage

Eléments de base

- **Déclaration des variable**

  Var Nom_de_la_variable : type

  Le type peut être :
  - Booléen
  - Entier
  - Caractère ou bien Chaîne de caractères
  - Réel
  - etc

Pseudo langage

Eléments de base

- **Déclaration des variable**

  Var Nom_de_la_variable : type

  Le type peut être :
  - Booléen
  - Entier
  - Caractère ou bien Chaîne de caractères
  - Réel
  - etc
- **Affection**

  ”←” ou tout simplement ”=”

Pseudo langage

Eléments de base

- **Déclaration des variable**

  Var Nom_de_la_variable : type

  Le type peut être :
  - Booléen
  - Entier
  - Caractère ou bien Chaîne de caractères
  - Réel
  - etc
- **Affection**

  ”←” ou tout simplement ”=”
- **Entrée-Sortie**
  - Lire,
  - Ecrire

Pseudo langage

Eléments de base

- **Déclaration des variable**

  Var Nom_de_la_variable : type

  Le type peut être :
  - Booléen
  - Entier
  - Caractère ou bien Chaîne de caractères
  - Réel
  - etc
- **Affection**

  ”←” ou tout simplement ”=”
- **Entrée-Sortie**
  - Lire,
  - Ecrire
- **Opérations arithmitiques**
  - +,-,/ (division réelle),*,
  - Mod (reste de la division), Div (résultat de la division)

Pseudo langage

Eléments de base

Pseudo langage

Eléments de base

- Conditionnel :

```
Si      condition
alors   { Instructions1 }
sinon   { Instructions2 }
```

Pseudo langage

Eléments de base

- Conditionnel :

```
Si        condition
alors     { Instructions1 }
sinon     { Instructions2 }
```

- Boucles :

```
Tantque   condition
{
          Instructions
}
```

Pseudo langage

Eléments de base

- Conditionnel :

```
Si      condition
alors   { Instructions1 }
sinon   { Instructions2 }
```

- Boucles :

```
Tantque   condition
{
          Instructions
}
```

- Boucles Pour :

```
Pour   i← valeur_initiale   ”à”   valeur_finale   (”pas=valeur”)
{
          Instructions
}
```

Quelques exercices simples

Exercices

La somme des chiffres d'un nombre

- Ecrire une fonction itérative
  - qui prend en paramètre un entier positif n et
  - qui retourne la somme des chiffres qui le composent.

Par exemple, si $n = 2095$ la fonction retournera le nombre 16.

Exercices

La somme des chiffres d'un nombre

- Ecrire une fonction itérative
  - qui prend en paramètre un entier positif n et
  - qui retourne la somme des chiffres qui le composent.

  Par exemple, si $n = 2095$ la fonction retournera le nombre 16.

Rappels

- $a\%b$: donne le reste de la division de a par b.
- a / b : donne le résultat de la division entière de a par b lorsque a et b sont des variables de type entier.

La somme des chiffres d'un nombre

```
int sommechiffres(int n)
 {
  int Res=0;
  while  (n!=0)
  {
  Res = Res + (n % 10);
  n = n / 10;
  }
  return Res;
 }
```

Exercices

Enigme

Considérons l'énigme suivant : Le 24 février 2014, Vincent a fêté son anniversaire. En présence de ses amis réunis pour l'occasion, il a fait la remarque suivante :

Cette année, mon âge est égal à la somme des chiffres de mon année de naissance.

- Ecrire une fonction itérative qui calcule l'âge (ainsi que l'année de naissance) possible de Vincent en 2014. Il est demandé d'afficher toutes les solutions possibles de l'énigme.

Enigme

```
void Enigme_anniversaire ()
 {
  int i;
  for (i=0; i<2015; i++) {
   if ((2014-i)==sommechiffres(i))
   printf("Annee=%d et Age=%d \n", i, (2014-i));}
 }
```

Remarques

- L'énigme admet deux solutions :
  - Annee=1988 (Age=26) et
  - Annee=2006 (Age=8).
- Dans la boucle for, on peut raisonnablement initialiser par exemple *i* à 1880.

Enigme

Une autre variante de la fonction

La valeur maximale de la sommes des chiffres composant une année est de 28 (qui correspond à l'année 1999). Une variante de la fonction *Enigme_anniversaire* est d'utiliser uniquement 29 itérations :

```
void Enigme_anniversaire ()
{
 int i;
 for (i=0; i<29; i++)
 {
 if ((2014-i)==sommechiffres(2014-i))
 printf("Solution trouvee : Annee=%d et Age=%d \n", i, (2014-i));
 }
}
```

Nombre premier

Exercices (simples)

- Ecrire une fonction qui vérifie si un nombre est premier.
- On rappelle qu'un nombre est premier s'il admet uniquement deux diviseurs différents : 1 et lui-même.

Nombre premier : algorithme super naïf

Principe

Compter le nombre de diviseurs de n.

Nombre premier : algorithme super naïf

Principe

Compter le nombre de diviseurs de n.

```
int Estpremiersupernaif(const int n)
{
 int i;
 int cpt_diviseurs=2;
 for(i = 2; i <n; i++)
  if(n % i == 0) cpt_diviseurs++;
 if (cpt_diviseurs==2)
  return 1;          // n est premier
  else return 0;   // n n'est pas premier
}
```

Nombre premier : algorithme juste naïf

Principe

Parcourir tous les entiers compris entre 2 et n et vérifier s'il existe un diviseur de n.

Nombre premier : algorithme juste naïf

Principe

Parcourir tous les entiers compris entre 2 et n et vérifier s'il existe un diviseur de n.

```
int Estpremiernaif(const int n)
{
 int i;
 if( n <= 1) return 0;
 for(i = 2; i <n; i++)
  if(n % i == 0) return 0;
 return 1;
}
```

Nombre premier : s'arrêter à $\frac{n}{2}$

Principe

Parcourir tous les entiers compris entre 2 et $\frac{n}{2}$ et vérifier s'il existe un diviseur de n.

Nombre premier : s'arrêter à $\frac{n}{2}$

Principe

Parcourir tous les entiers compris entre 2 et $\frac{n}{2}$ et vérifier s'il existe un diviseur de n.

```
int Estpremierpar2(const int n)
{
 int i, j=n/2+1;
 if( n <= 1) return 0;
 for(i = 2; i < j; i++)
  if(n % i == 0) return 0;
 return 1;
}
```

Nombre premier : s'arrêter à la racine carrée

Principe

Parcourir tous les entiers compris entre 2 et $\sqrt{n}$ et vérifier s'il existe un diviseur de n.

n

Intuitivement

Supposons qu'il existe un diviseur $a > \sqrt{n}$ de n. Cela signifie qu'il existe un b tel que :

$$n = a * b.$$

Par conséquent, b est également diviseur de n et est nécessairement inférieur $\sqrt{n}$. C'est-à-dire si $a > \sqrt{n}$ est diviseur de n alors il existe nécessairement un autre diviseur $b < \sqrt{n}$

Nombre premier : s'arrêter à la racine carrée

Principe

Parcourir tous les entiers compris entre 2 et $\sqrt{n}$ et vérifier s'il existe un diviseur de n.

```
int Estpremierracine2(const int n)
{
 int i, j=(int) (sqrt(n)+1);
 if( n <= 1) return 0;
 else if (n==2) return 1;
 for(i = 2; i <= j;i ++)
  if(n % i == 0) return 0;
 return 1;
}
```

Algorithmes approximatifs

- Les algorithmes précédents ne fonctionnent pas avec des grands nombres (200 chiffres par exemples)
- Utiliser des propriétés des nombres premiers

Propriétés des nombres premiers

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.
  Prenons :

$$d = a * b * c + 1.$$

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.
  Prenons :

$$d = a * b * c + 1.$$

  Clairement,

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.
  Prenons :

$$d = a * b * c + 1.$$

  Clairement,

$$d \bmod a = d \bmod b = d \bmod c = 1.$$

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.
  Prenons :

$$d = a * b * c + 1.$$

  Clairement,

$$d \bmod a = d \bmod b = d \bmod c = 1.$$

  De ce fait,
  - soit d est premier,

Propriétés des nombres premiers

- Tout entier positif p peut s'écrire comme le produit de nombres premiers (appelés facteurs premiers de p).
- Il existe un nombre infini de nombres premiers.
  Supposons qu'il y en que 3 (on peut ensuite généraliser): a, b, c.
  Prenons :

$$d = a * b * c + 1.$$

  Clairement,

$$d \bmod a = d \bmod b = d \bmod c = 1.$$

  De ce fait,
  - soit d est premier,
  - soit un de ces facteurs est premier. Ce facteur est différent de a (resp. de b, c), sinon $d \bmod a = 0$ (ce qui est impossible).

Propriétés des nombres premiers

- si p est premier alors pour tout entier $1 < a < p$ est tel que :

$$a^{p-1} \bmod p = 1,$$

Propriétés des nombres premiers

- si p est premier alors pour tout entier $1 < a < p$ est tel que :

$$a^{p-1} \bmod p = 1,$$

- On peut prendre a plus grand que p pourvu que $pgcd(a, p) = 1$.

Un algorithme simple mais partiellement exacte :

Pour déterminer si p est un nombre premier, on choisit un entier $1 < a < p$ quelconque, et on calcule :

Un algorithme simple mais partiellement exacte :

Pour déterminer si p est un nombre premier, on choisit un entier $1 < a < p$ quelconque, et on calcule :

$$r = a^{p-1} \bmod p$$

Un algorithme simple mais partiellement exacte :

Pour déterminer si p est un nombre premier, on choisit un entier $1 < a < p$ quelconque, et on calcule :

$$r = a^{p-1} \bmod p$$

- Si $r \neq 1$, on conclut que p n'est pas un nombre premier

Un algorithme simple mais partiellement exacte :

Pour déterminer si p est un nombre premier, on choisit un entier $1 < a < p$ quelconque, et on calcule :

$$r = a^{p-1} \bmod p$$

- Si $r \neq 1$, on conclut que p n'est pas un nombre premier
- Si $r = 1$, on conclut que p est peut-être un nombre premier.

Test de primalité = calcul de puissance

Le test de primalité de p revient à calculer :

- Une puissance (modulaire) $a^{p-1} \bmod p$

Test de primalité = calcul de puissance

Le test de primalité de p revient à calculer :

- Une puissance (modulaire) a^{p-1} mod p
- L'opération *mod p* permet d'avoir le résultat entre 0 et $p-1$.

Test de primalité = calcul de puissance

Le test de primalité de p revient à calculer :

- Une puissance (modulaire) a^{p-1} mod p
- L'opération *mod p* permet d'avoir le résultat entre 0 et $p-1$.
- Sans la présence de ce modulo, il peut-être difficile de représenter a^{p-1}.

Test de primalité = calcul de puissance

Le test de primalité de p revient à calculer :

- Une puissance (modulaire) a^{p-1} mod p
- L'opération *mod p* permet d'avoir le résultat entre 0 et $p-1$.
- Sans la présence de ce modulo, il peut-être difficile de représenter a^{p-1}.
- Cet algorithme est particulièrement adapté pour tester la primalité de grands nombres premiers.

Version probabiliste du test de la primalité

```
int Estpremierprobabiliste( long int p)
{
int i;
i=(rand()%(p-1))+2;
if( p <= 1) return 0;
if ((puissance_modulaire_rapide(i, p-1,p) % p) != 1)
 return (0);
 else return (1);
}
```

Une probabilité d'échec proche de 0 :

L'algorithme précédent n'est pas exacte lorsque $r = a^{p-1} \bmod p = 1$

- Au fait, pour $r = 1$ la probabilité que p soit premier est proche de $\frac{1}{2}$.

Une probabilité d'échec proche de 0 :

L'algorithme précédent n'est pas exacte lorsque $r = a^{p-1} \bmod p = 1$

- Au fait, pour $r = 1$ la probabilité que p soit premier est proche de $\frac{1}{2}$.
- Si on choisit m entiers $1 < a_i < p$, et que pour chaque a_i nous avons
$$r_i = a_i^{p-1} \bmod p = 1,$$
alors la probabilité que p soit premier est proche de $1 - (\frac{1}{2})^m$.

Une probabilité d'échec proche de 0 :

L'algorithme précédent n'est pas exacte lorsque $r = a^{p-1} \bmod p = 1$

- Au fait, pour $r = 1$ la probabilité que p soit premier est proche de $\frac{1}{2}$.
- Si on choisit m entiers $1 < a_i < p$, et que pour chaque a_i nous avons
$$r_i = a_i^{p-1} \bmod p = 1,$$
alors la probabilité que p soit premier est proche de $1 - (\frac{1}{2})^m$.
- Plus m est grand, plus on est certain que p est premier.

Exercices : nombre de nombres premiers

Nombre de nombres premiers

- Ecrire une fonction qui calcule le nombre de nombres premiers compris entre deux entiers *a* et *b*.

Nombre de nombres premiers

```
2    int Nbredenbrespremiersnaif(const  int a, const int b)
3   {
4       int i, nbrepremiers=0;
5          for(i = a; i <= b; i++){
6           if (Estpremiernaif(i) == 1) nbrepremiers++;
7       }
8    return nbrepremiers;
9   }
```

Nombres parfaits, déficients, abondants

Exercices (toujours simples)

- Un nombre est dit parfait s'il est égal à la somme de tous ses diviseurs (sauf lui même). Les quatre premiers nombres parfaits sont : 6, 28, 496 et 8128.

- Un nombre est dit abondant s'il est strictement supérieur à la somme de tous ses diviseurs (sauf lui même). Il est dit déficient s'il est strictement inférieur à la somme de tous ses diviseurs (sauf lui même).

Nombres parfaits, déficients, abondants

Exercices (toujours simples)

- Un nombre est dit parfait s'il est égal à la somme de tous ses diviseurs (sauf lui même). Les quatre premiers nombres parfaits sont : 6, 28, 496 et 8128.
- Un nombre est dit abondant s'il est strictement supérieur à la somme de tous ses diviseurs (sauf lui même). Il est dit déficient s'il est strictement inférieur à la somme de tous ses diviseurs (sauf lui même).
- Ecrire une fonction qui prend en paramètre un entier n et retourne 0 si n est parfait, 1 s'il est abondant et -1 s'il est déficient.

Nombres parfaits, déficients, abondants

Exercices (toujours simples)

- Un nombre est dit parfait s'il est égal à la somme de tous ses diviseurs (sauf lui même). Les quatre premiers nombres parfaits sont : 6, 28, 496 et 8128.
- Un nombre est dit abondant s'il est strictement supérieur à la somme de tous ses diviseurs (sauf lui même). Il est dit déficient s'il est strictement inférieur à la somme de tous ses diviseurs (sauf lui même).
- Ecrire une fonction qui prend en paramètre un entier n et retourne 0 si n est parfait, 1 s'il est abondant et -1 s'il est déficient.
- Dérouler à la main votre algorithme avec les nombres 33550336 et 33550338

Les nombres parfaits, abondants et déficients

```
int Estparfait(const int n)
{
 int i, j=(n/2+1), somme_diviseurs=0;
 if( n <= 1) return -2;
 for(i = 1; i <= j; ++i)
 {
  if(n % i == 0)
  {
   somme_diviseurs=somme_diviseurs+i;
   if (somme_diviseurs>n) return 1 /*abondant*/;
  }
 }
 if (somme_diviseurs==n) return 0 /*parfait*/;
 else return -1 /*Deficient*/;
 }
```

Inconvénient de l'algorithme

Cet fonction ne permet pas pas traiter des grands nombres.

Exemple

Il est impossible avec cette fonction de vérifier si ce nombre est parfait ou non :

2143017214372534641896850098120003621122809
6234110672148875007767407021022498722449863
9675763139171625518934583510629365037429057
1384628087196915514939714960786913554964846
1970842149210124742283755908364306092949967
1638825347975351183310878921541258291423929
5537308433532085966330524877367441133613875 2

Nombres parfaits : un autre algorithme

Exercices (toujours simples)

Nombres parfaits : un autre algorithme

Exercices (toujours simples)

- On s'intéresse aux nombres parfaits pairs (on ne sait pas si les nombres parfaits impairs existent). Il se trouve qu'un nombre pair est parfait si et seulement si
  - il est de la forme $2^{n-1}*(2^n\text{-}1)$ et
  - $(2^n\text{-}1)$ est un nombre premier.

- On supposons que nous disposons d'une fonction qui vérifie si un nombre $(2^n\text{-}1)$ est premier. Proposer un algorithme qui vérifie si un nombre pair est parfait et qui tient compte de cette remarque.

Nombres parfaits : un autre algorithme

Exercices (toujours simples)

- On s'intéresse aux nombres parfaits pairs (on ne sait pas si les nombres parfaits impairs existent). Il se trouve qu'un nombre pair est parfait si et seulement si
  - il est de la forme $2^{n-1}*(2^n\text{-}1)$ et
  - $(2^n\text{-}1)$ est un nombre premier.

- On supposons que nous disposons d'une fonction qui vérifie si un nombre $(2^n\text{-}1)$ est premier. Proposer un algorithme qui vérifie si un nombre pair est parfait et qui tient compte de cette remarque.

- Le deuxième algorithme est-il plus efficace?

Les nombres parfaits : version efficace

```
int Estparfaitpower(const int p)
{
int pn=1, pnmoinsun=1, produit=1;
// pn = 2^n
// pnmoinsun=2^{n-1}
// produit = 2^{n-1} * (2^n -1)
while (produit < p)
{
  pnmoinsun=pn;
  pn=pn*2;
  produit=pnmoinsun*(pn-1);
  if ((produit==p)
   if (Estpremierprobabiliste (pn-1))) return 1;
 }
 return 0;
}
```

Liste des nombres parfaits : un algorithme très efficace

- La propriété des nombres parfaits permet de fournir un algorithme extrêmement efficace pour lister les premiers nombres parfaits.

Liste des nombres parfaits : un algorithme très efficace

```
void Listeparfaitpower(const int n)
{
    double long pn=2, pnmoinsun=1;
    int i;
    // pn = 2^n
    // pnmoinsun=2^{n-1}
    // produit = 2^{n-1} * (2^n -1)

    for (i=1; i<=n; i++)
    {
    pnmoinsun=pn;
    pn=pn*2;
   if (Estpremierprobabiliste (pn-1))
   printf ("Le nombre %.0Lf est parfait \n", pnmoinsun*(pn-1));
    }
}
```

Nombres parfaits : un autre algorithme

Voici les premiers nombres parfaits :

6
28
496
8128
33550336
137438691328
2305843008139952128

Gestion statique de la mémoire:

Tableaux

Tableaux

- Un tableau est une structure de donnée permettant de mémoriser des valeurs du même type qui peut être :
  - un type simple
  - un type complexe (enregistrement)
- Chaque élément d'un tableau est représenté par un indice indiquant sa position
- La taille d'un tableau est le nombre de ses éléments
- La taille d'un tableau est **fixe**.
- L'espace mémoire est réservé au niveau de la compilation.

Tableaux

- Déclaration :
  Constante : C1;
  type Nom_du_tableau [C1];
- Les indices varient entre une valeur minimale (en général 0 ou 1) et une valeur maximale (une constante)
- Chaque élément du tableau est une variable indicée, identifiée par le nom du tableau suivi de l'indice entre crochets.

Exercice: Tableau

Exercice facile

Ecrire une fonction booléenne qui vérifie si un tableau d'entiers est trié.

Vérification si un tableau est trié

```
2  int est_trie (int A[], int N)
3  {
4   int i=0;
5   while (i<(N-1))
6   {
7    if (A[i]<A[i+1]) return 0;
8    i++;
9   }
0   return 1;
   }
```

Pomme, pêche, poire, abricot

Pomme, pêche, poire, abricot
Y'en a une, y'en a une
Pomme, pêche, poire, abricot
Y'en a une qui est en trop.

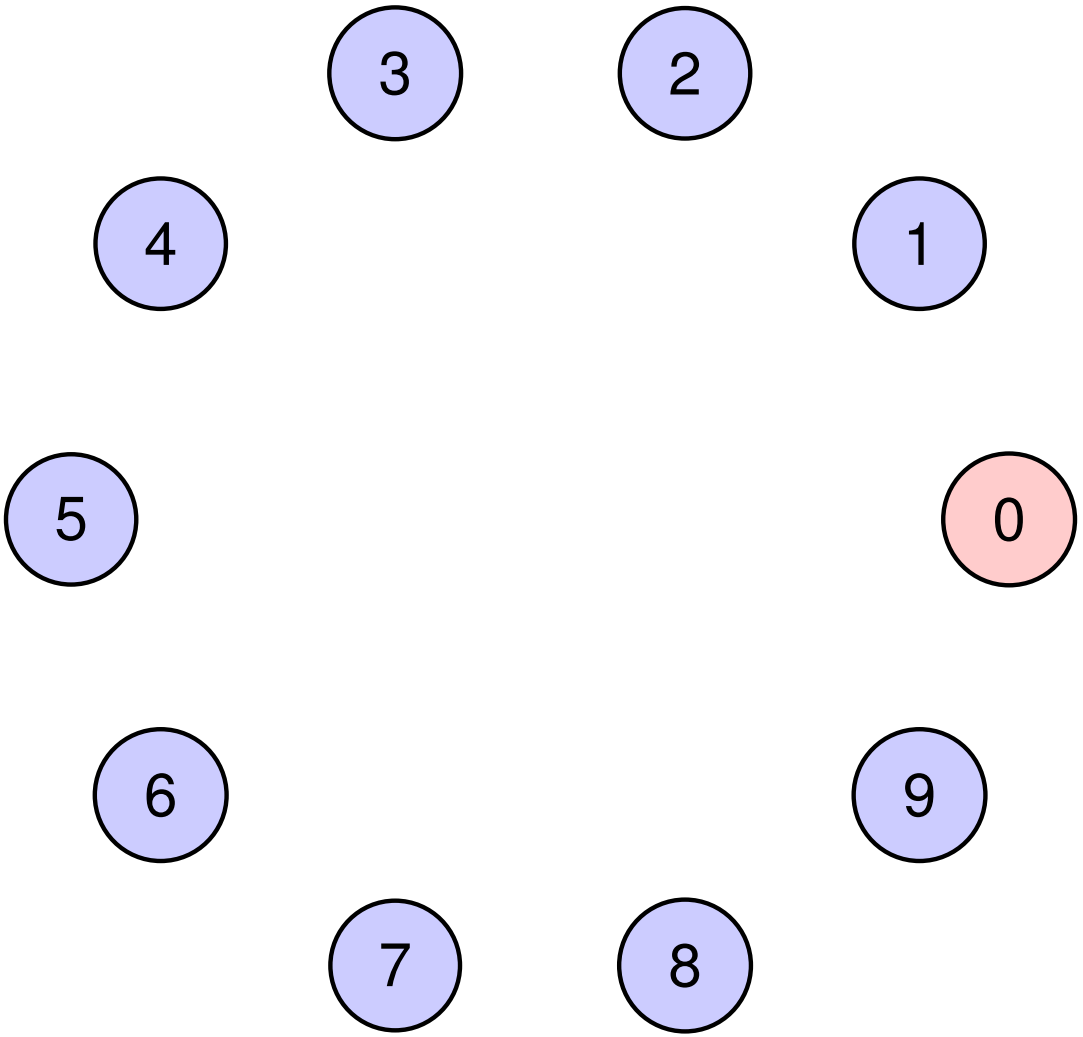
Exercice un peu plus difficile

Jeu

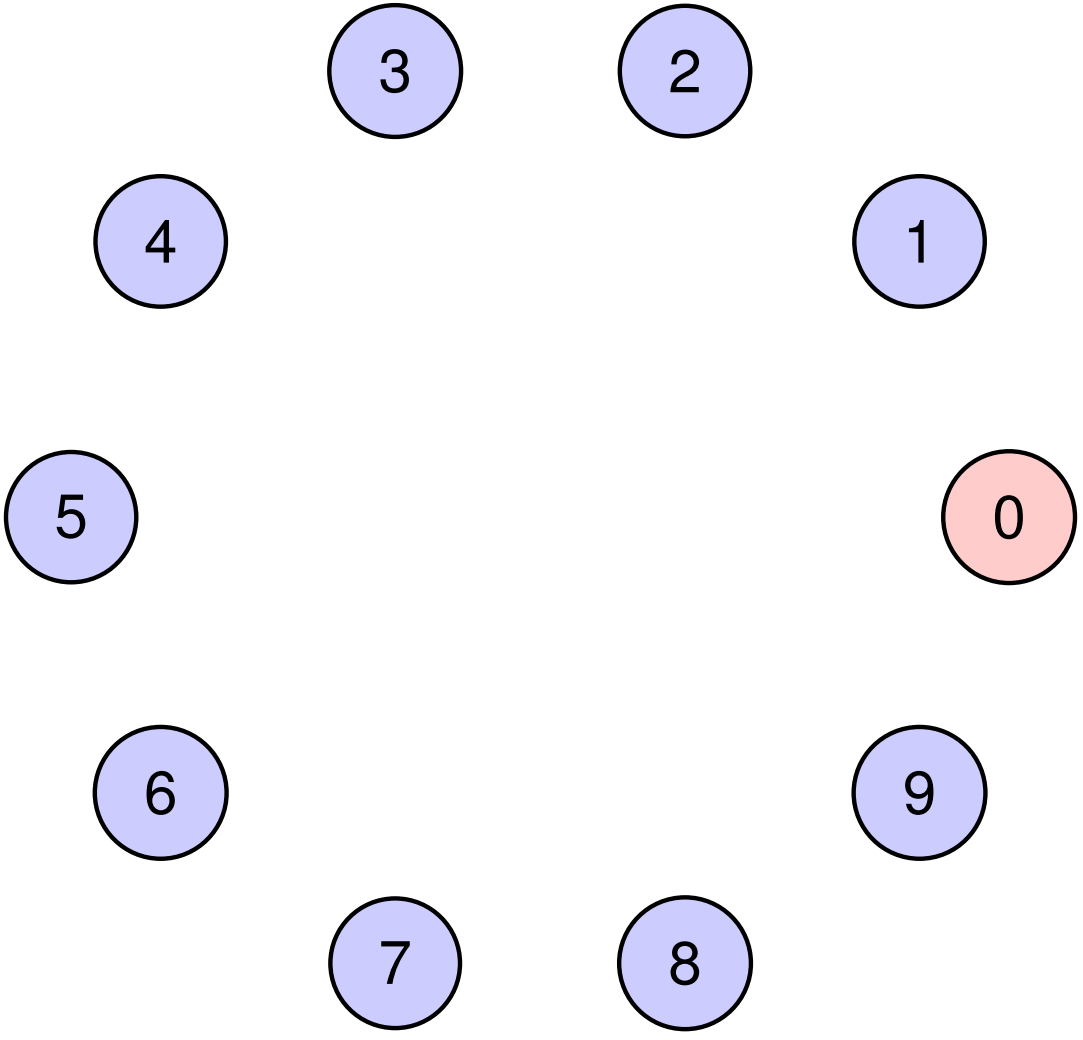
Ecrire un algorithme qui code le jeu suivant :

- N joueurs (numérotés de 1 à N) forment un cercle. Les numéros des *N* joueurs sont stockés dans un tableau.
- On se donne (on lit) un nombre M
- On compte de 1 à M de manière répétitive.
- Le $M^{ième}$ joueur est éliminé
- Le jeu s'arrête lorsqu'il ne reste qu'un seul joueur (le gagnant)

Déroulement du Jeu : Il reste 10 joueurs

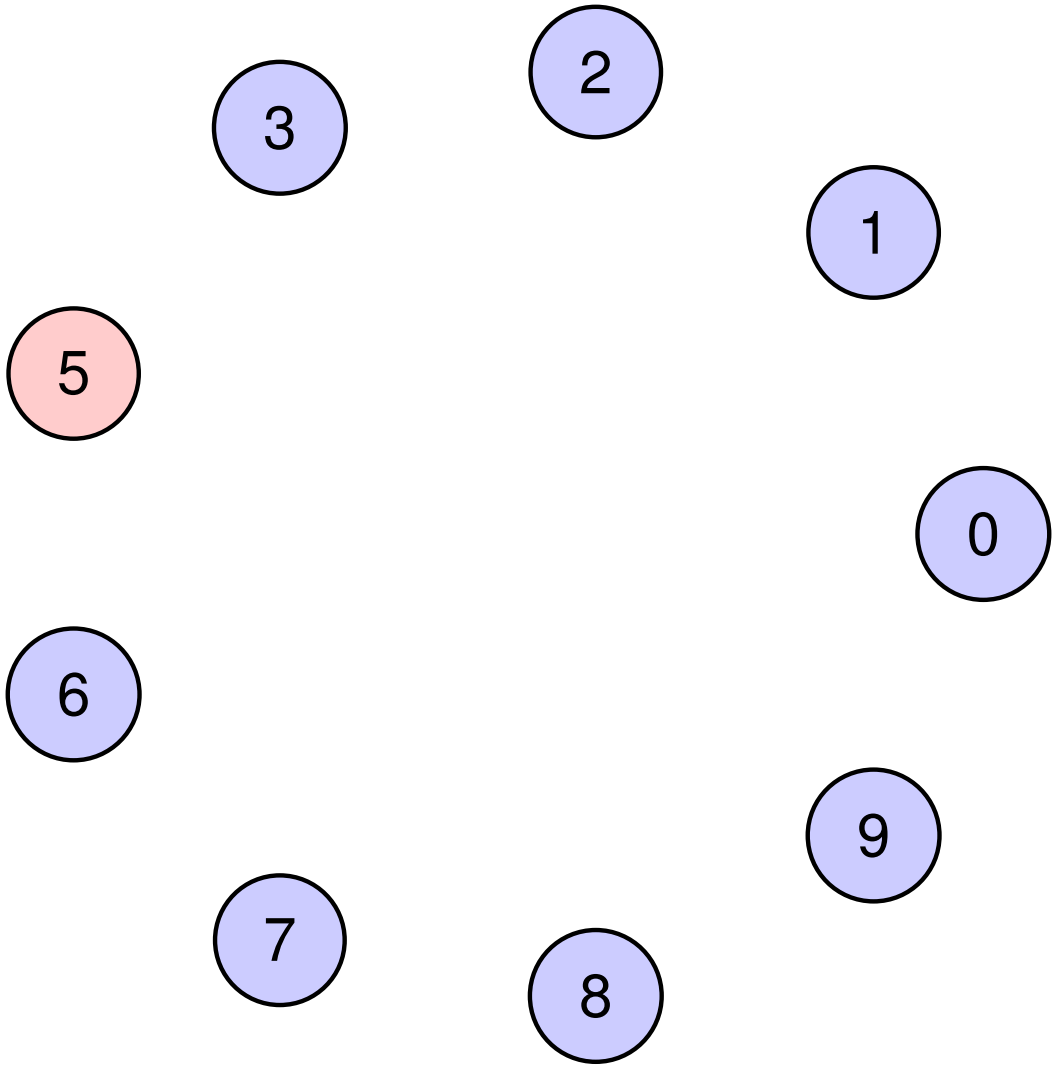


Déroulement du Jeu : Il reste 10 joueurs



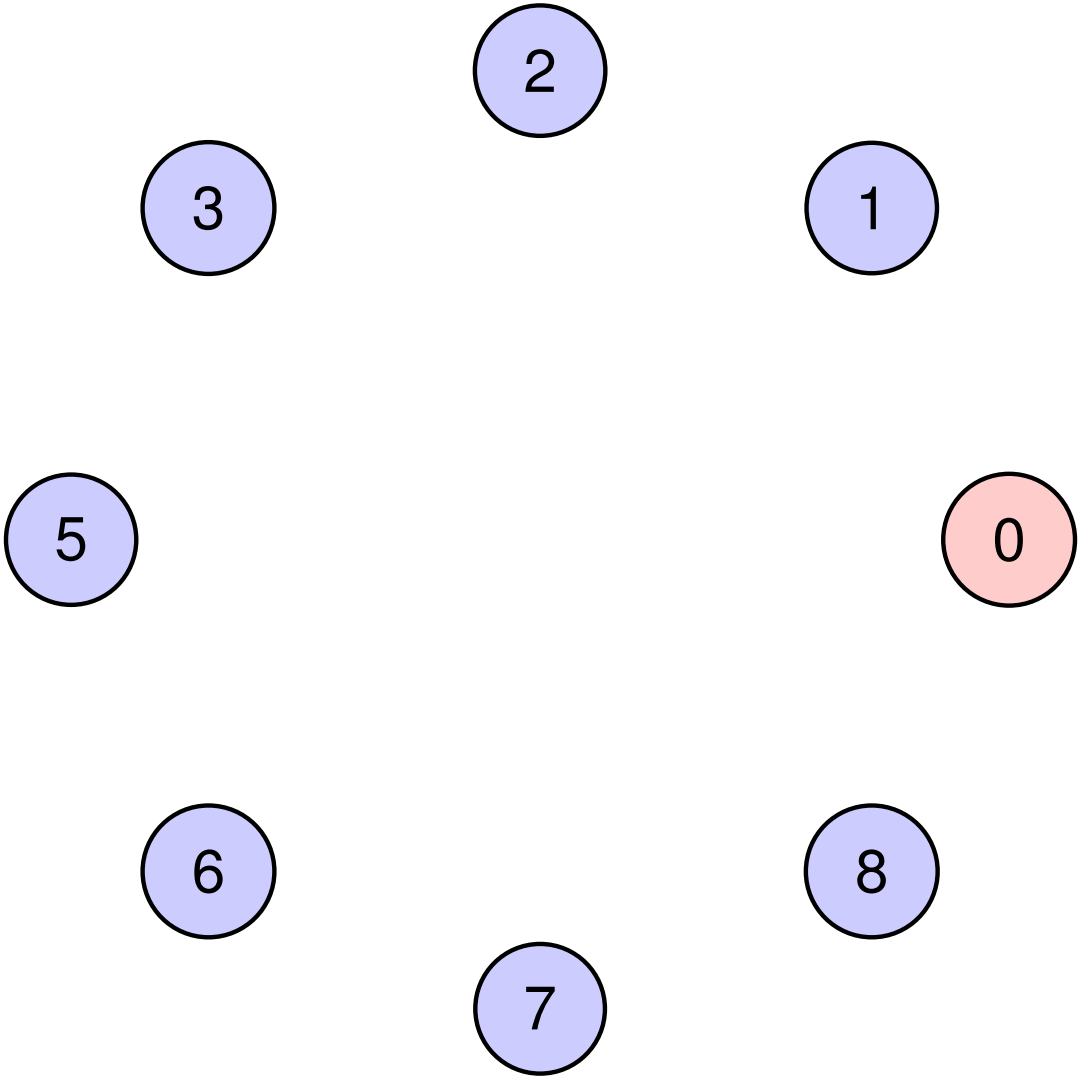
Le joueur 4 sera éliminé.

Déroulement du Jeu : Il reste 9 joueurs



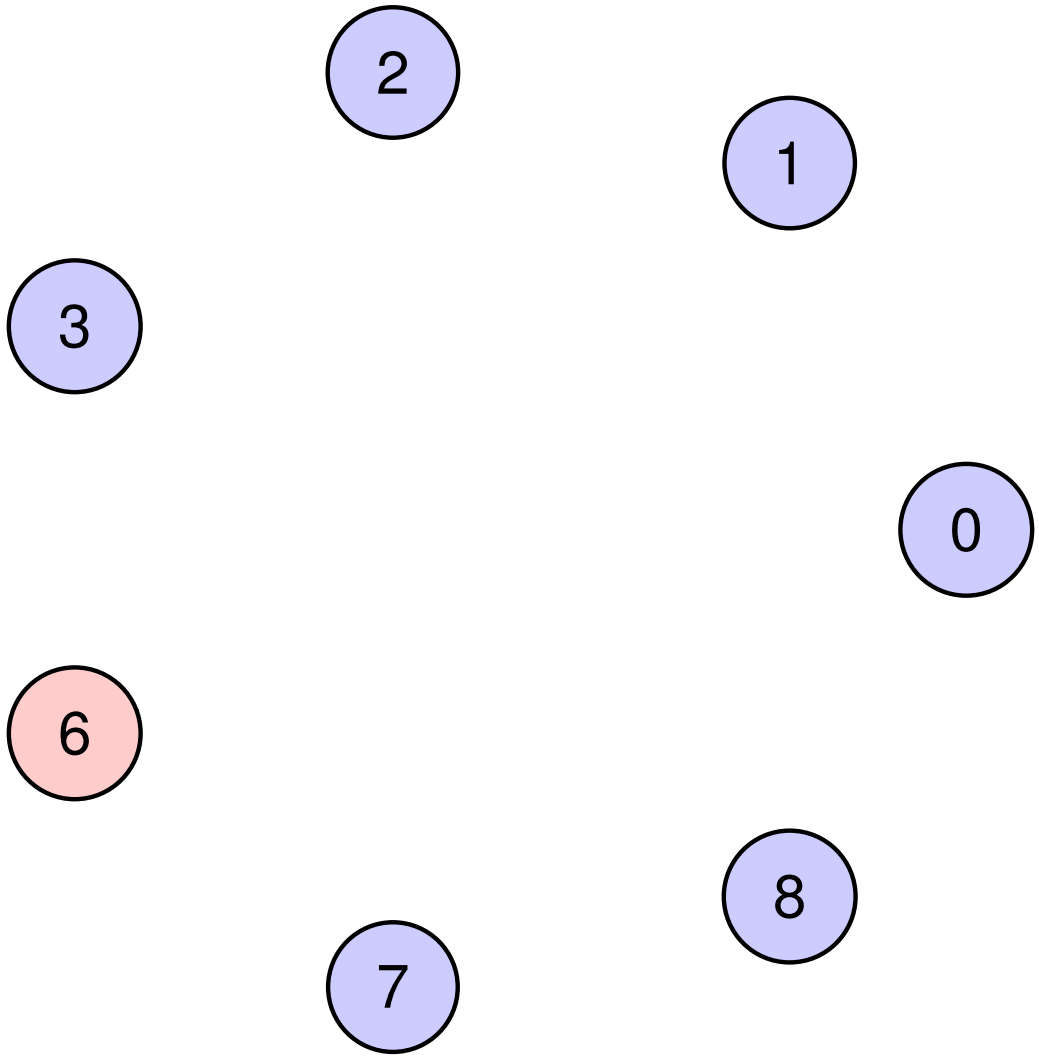
Le joueur 9 sera éliminé.

Déroulement du Jeu : Il reste 8 joueurs



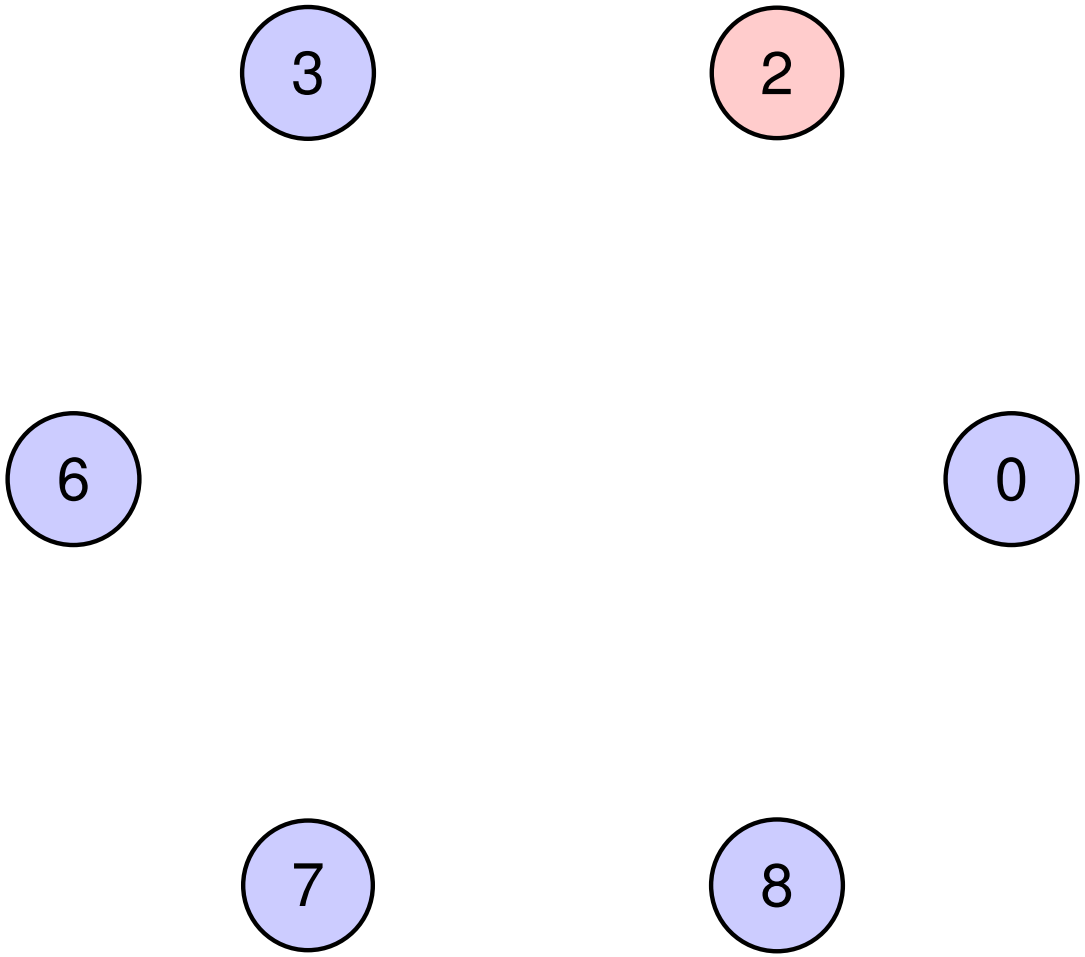
Le joueur 5 sera éliminé.

Déroulement du Jeu : Il reste 7 joueurs



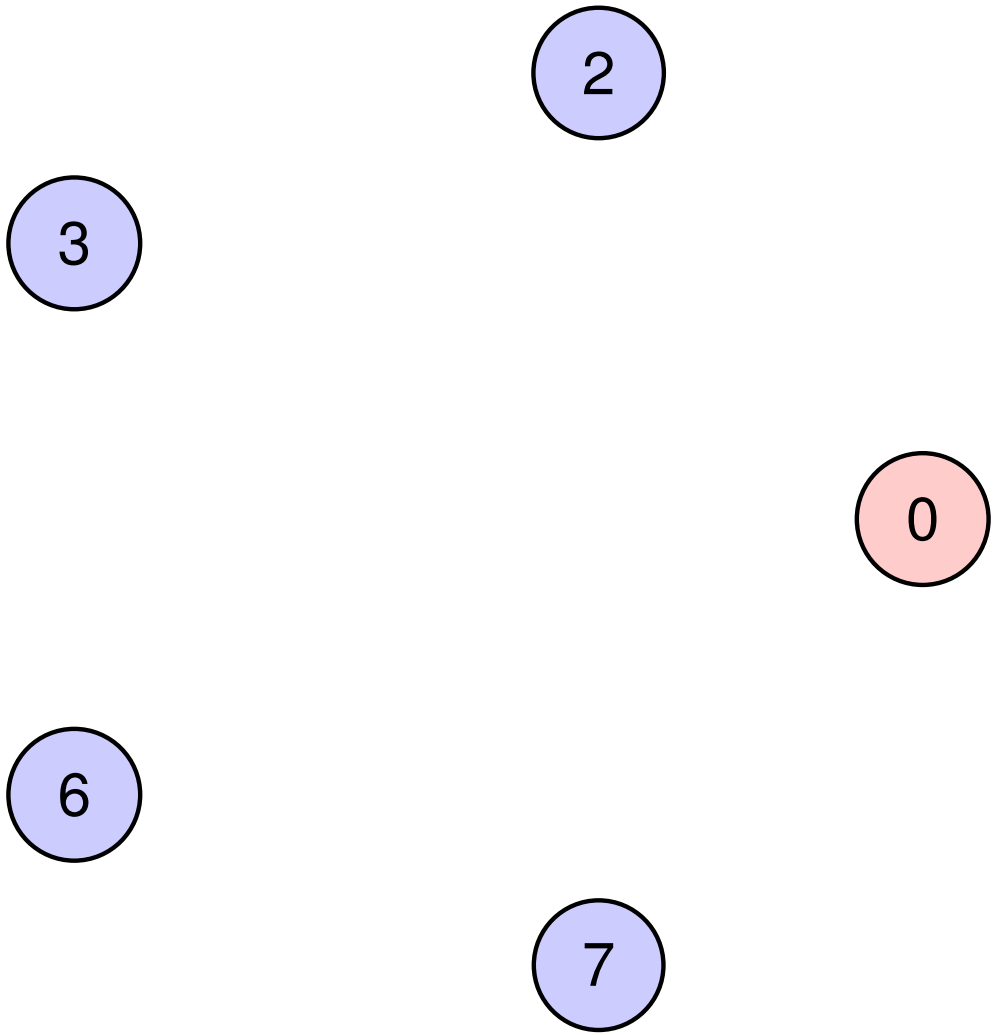
Le joueur 1 sera éliminé.

Déroulement du Jeu : Il reste 6 joueurs



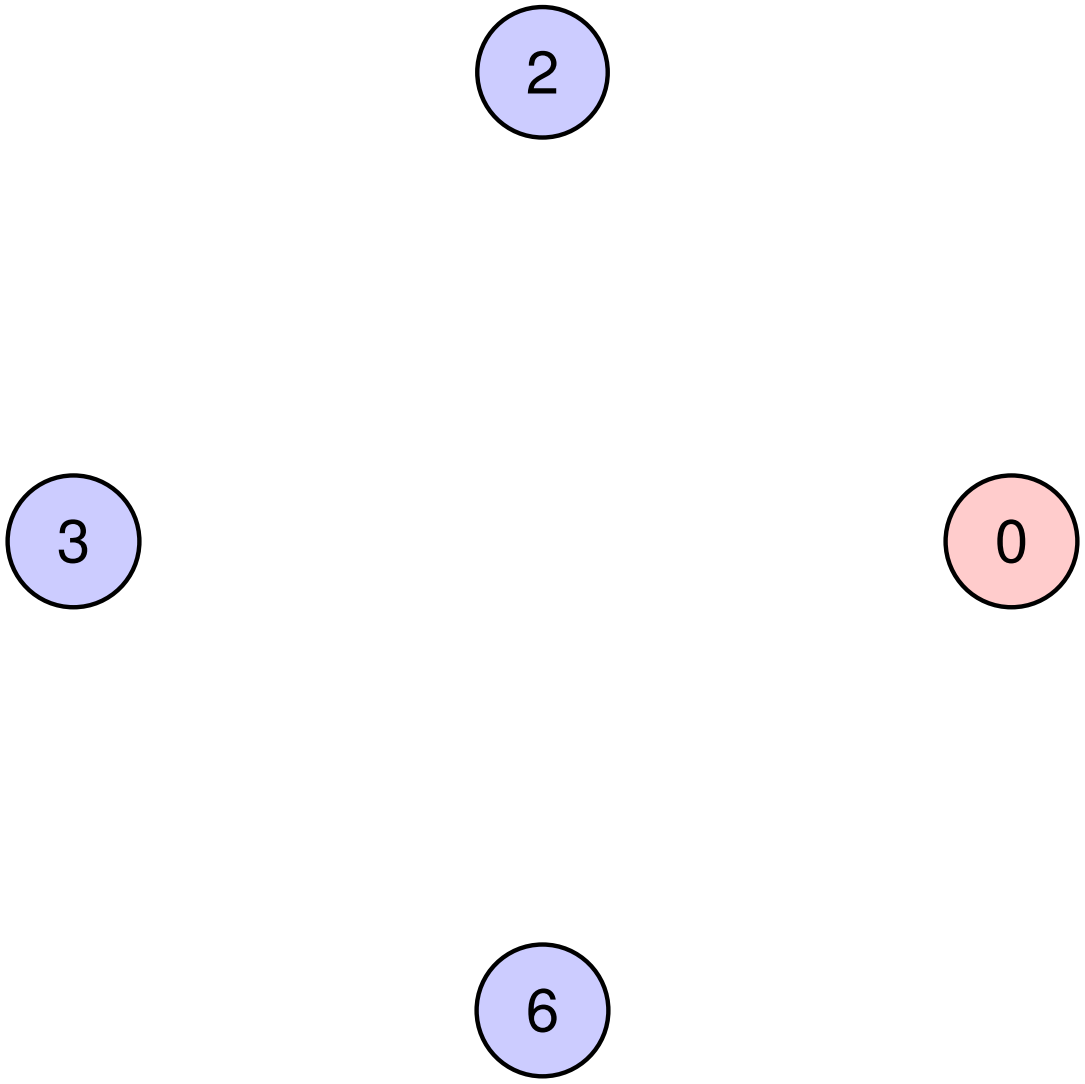
Le joueur 8 sera éliminé.

Déroulement du Jeu : Il reste 5 joueurs



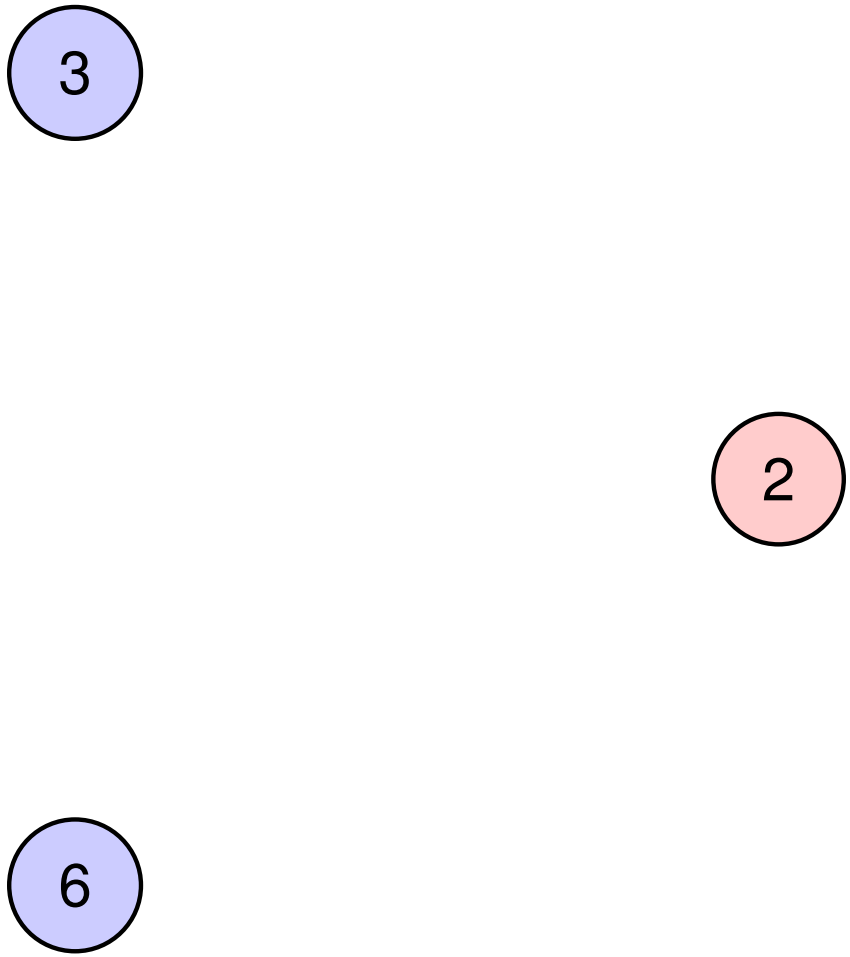
Le joueur 7 sera éliminé.

Déroulement du Jeu : Il reste 4 joueurs



Le joueur 0 sera éliminé.

Déroulement du Jeu : Il reste 3 joueurs



Le joueur 3 sera éliminé.

Déroulement du Jeu : Il reste 2 joueurs

Le joueur 6 sera éliminé.

Déroulement du Jeu : Il reste un et un seul joueur

Le gagnant est :

Fonction Jeu: en langage C

```
2  int Jeu (int N, int M)
3  {
4    // Declaration des variables
5    int Tab[Taille];
6    int cpt_M=0, i=0, NB_j=N;
7    //Initialisation du tableau des joueurs
8    for (i=0; i<N; i++) Tab[i]=1;
```

Fonction Jeu: en langage C

```
  i=0;
  // Tant qu'il y a plus de deux joueurs
  while (NB_j >1)
  {
  //Si i est un joueur non-elimine
    if (Tab[i]==1)
    { //Si on est en presence du joueur a eliminer
      cpt_M = cpt_M + 1;
      if (cpt_M == M)
      {
        NB_j = NB_j - 1;
        Tab[i]=0;
        cpt_M = 0;
      }
    }
    i=(i+1) % N;
  }
  for (i=0; i<N; i++) if (Tab[i]==1) return i;
}
```

Fonction Jeu

Remarque : On peut faire une autre variante avec des décallages et des modulos.

Pomme, pêche, poire, abricot : avec le modulo

```
int Jeu (int N, int M)
{
 int Tab[Taille];
 int i=0,j;
 for (i=0; i<N; i++) Tab[i]=i;
 while (N>1)
 {
  i=(i + M-1) % N;
  printf ("\n Est elimine le numero : %d \n", Tab[i]);
  for (j=i; j<(N-1); j++) Tab[j]=Tab[j+1];
  N=N-1;
 }
 return (Tab[0]);
}
```

Avantages des tableaux

Avantages des tableaux

- Simplicité

Avantages des tableaux

- Simplicité
- Efficacité : Recherche rapide si le tableau est trié!

Avantages des tableaux

- Simplicité
- Efficacité : Recherche rapide si le tableau est trié!
- Adressage et accès direct (base des tables de hachages).

Allocation dynamique vs allocation statique

- Une allocation de la mémoire est statique si elle se produit avant l'exécution et demeure inchangée pendant toute la durée de l'exécution du programme.

Allocation dynamique vs allocation statique

- Une allocation de la mémoire est statique si elle se produit avant l'exécution et demeure inchangée pendant toute la durée de l'exécution du programme.
- Une allocation de la mémoire est dynamique si elle se produit pendant l'exécution du programme.

Limitations des tableaux

La mémoire statique

- Déclaration d'une variable d'un type T = allocation statique d'une zone mémoire

Limitations des tableaux

La mémoire statique

- Déclaration d'une variable d'un type T = allocation statique d'une zone mémoire
- Ceci est particulièrement vrai lorsque l'on déclare un tableau.

Limitations des tableaux

La mémoire statique

- Déclaration d'une variable d'un type T = allocation statique d'une zone mémoire
- Ceci est particulièrement vrai lorsque l'on déclare un tableau.
  - Si N représente la taille d'un tableau
  - et que M est la taille (en octet) de l'élément à stocker,
  - alors au niveau de la compilation un espace de N * M octets est réservé indépendamment si cet espace sera entièrement utilisée ou non.

Limitations des tableaux

La mémoire statique

- Il faut absolument connaître à l'avance le nombre maximum N d'éléments à stocker.

Limitations des tableaux

La mémoire statique

- Il faut absolument connaître à l'avance le nombre maximum N d'éléments à stocker.
- Typiquement on déclare plus d'éléments que ce qui est nécessaire (donc perte d'espace mémoire).

Limitations des tableaux

La mémoire statique

- Il faut absolument connaître à l'avance le nombre maximum N d'éléments à stocker.
- Typiquement on déclare plus d'éléments que ce qui est nécessaire (donc perte d'espace mémoire).
- Le programme ne fonctionnerait pas correctement si le nombre d'éléments réel dépasse le nombre d'éléments déclarés

Limitations des tableaux

La mémoire statique

- Il faut absolument connaître à l'avance le nombre maximum N d'éléments à stocker.
- Typiquement on déclare plus d'éléments que ce qui est nécessaire (donc perte d'espace mémoire).
- Le programme ne fonctionnerait pas correctement si le nombre d'éléments réel dépasse le nombre d'éléments déclarés
- Certaines opérations sont coûteuses avec les tableaux comme supprimer un élément dans un tableau trié.