Devoir Non Surveillé 1

À rendre le 6 novembre 2023

INFORMATIQUE MPI/MPI*

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 3 parties indépendantes, utilisant les langages de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I se concentre sur l'étude des couplages de cardinal maximum, leur lien avec les couvertures par sommet, puis propose des jeux à 2 joueurs sur ce thème.
- La partie II s'intéresse aux jeux d'accessibilité à deux joueurs sur un graphe et à leur formalisme, ainsi qu'à leur implémentation en OCaml.
- La partie III s'intéresse aux tests de primalité probabilistes, et à leur implémentation en C.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

Partie I. Couplages dans les graphes bipartis

On s'intéresse au problème de la *couverture minimale par sommets* (ou problème du transversal minimum) : étant donné un graphe $G = (S, A)$, déterminer le nombre de sommets minimal d'un ensemble $C \subset S$ tel que $\forall \{x, y\} \in A, x \in C \vee y \in C$. Un tel ensemble est appelé une couverture par sommets du graphe G.

☐ 1 – Démontrer que le nombre de sommets d'une couverture minimale par sommets est toujours supérieur ou égal au nombre d'arêtes d'un couplage de cardinal maximum.

Réponse 1. Soit C une couverture par sommets et M un couplage. Par définition de C, on a $\forall a \in M. \exists x \in a. x \in C$. Donc si on note $|M| = m$, $\exists (x_i)_{i=1\dots m}$, $x_i \in C$, et par définition de M, les x_i sont deux à deux distincts, donc $|\{x_i, i = 1 \dots m\}| = m$, et $|C| \geq m$. En particulier, c'est vrai pour une couverture minimale par sommets et un couplage de cardinal maximum.

Dans la suite de cette partie, on s'intéresse à un graphe biparti non orienté $G = (S, A)$, avec $S = X \cup Y$, $X \cap Y = \emptyset$ et $A \subset \{ \{x, y\} | \ x \in X, y \in Y\}$.

Soit M un couplage de cardinal maximum pour G. On note L l'ensemble des sommets accessibles par un chemin alternant [1] depuis un sommet libre de X.

☐ 2 – Montrer que $C = (X \setminus L) \cup (Y \cap L)$ constitue une couverture par sommets de G.

Réponse 2. Analysons d'abord un peu les chemins alternants depuis un sommet libre de X. Un tel chemin est de la forme :

$$x_0 - y_0 - x_1 - \dots$$

On a nécessairement $\{x_0, y_0\} \notin M$ car x_0 est libre. Par récurrence immédiate, on a pour ce chemin : $\forall i, \{x_i, y_i\} \notin M$ et $\{y_i, x_{i+1}\} \in M$.

Revenons à la question sur C. Soit $a = \{x, y\} \in A$ une arête quelconque, avec $x \in X$ et $y \in Y$. On va montrer que $x \in L \Rightarrow y \in L$. Procédons par disjonction de cas :

$a \in M$ **:** Tout chemin alternant depuis un sommet libre de X terminant en x se termine par une arête du couplage, donc a. Donc si $x \in L$, alors nécessairement $y \in L$, en suivant le même chemin sans l'arête a.

$a \notin M$ **:** Tout chemin alternant depuis un sommet libre de X terminant en x se termine par une arête du couplage, donc on peut le prolonger avec l'arête a pour obtenir un nouveau chemin alternant terminant en y, et $y \in L$.

Dans tous les cas, on a soit $x \notin L$ et donc $x \in C$, soit $y \in L$ et donc $y \in C$. C est donc bien une couverture par sommets.

☐ 3 – Justifier que L ne contient aucun sommet libre de Y.

Réponse 3. Si L contient un sommet libre de Y, on a un chemin alternant joignant un sommet libre de X et un sommet libre de Y, c'est-à-dire un chemin augmentant, ce qui contredit la maximalité de M.

☐ 4 – Montrer qu'il n'existe pas d'arête de M reliant un sommet de $X \setminus L$ et un sommet de $Y \cap L$.

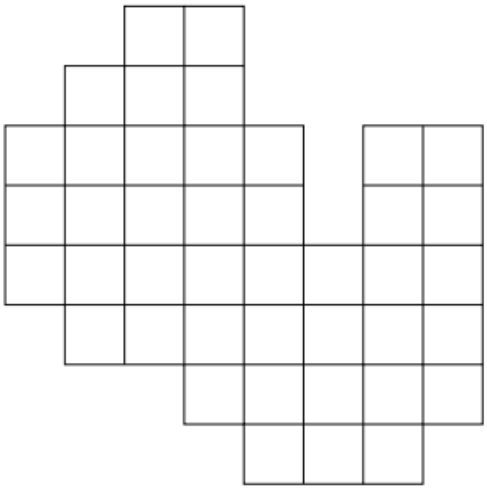
Réponse 4. Par l'absurde, soit $a = \{x, y\} \in M$, avec $x \in X \setminus L$ et $y \in Y \cap L$. Alors il existe un chemin alternant joignant un sommet libre de X et y. Comme vu précédemment, ce chemin se termine nécessairement par une arête hors de M. Donc on peut étendre ce chemin avec a en un chemin alternant joignant un sommet libre de X et x, et $x \in L$, ce qui est une contradiction.

☐ 5 – Démontrer le théorème de König (1931) :
Pour tout graphe biparti, le cardinal maximum d'un couplage est égal au cardinal minimum d'une couverture par sommets.

Réponse 5. C ne contient aucun sommet libre, donc chaque arête de M possède au moins un sommet dans C. De plus, aucune arête de M ne possède ses deux sommets dans C (question 4). Donc chaque arête de M possède exactement un sommet dans C, qui sont tous différents, et $|C| = |M|$.

☐ 6 – Peut-on paver la figure suivante à l'aide de dominos ? Donner un pavage ou une preuve concise qu'aucun tel pavage ne peut exister.

1. On appelle *chemin* alternant un chemin qui alterne les arêtes dans M et les arêtes dans $A \setminus M$



Réponse 6. Si on colorie les cases de la figure en blanc et noir comme un damier, on peut remarquer qu'un domino couvre toujours une case noire et et une case blanche. On a donc bien un graphe biparti. Une couverture par des dominos correspond à un couplage dans ce graphe car chaque case ne peut être recouverte que par un seul domino. Il y a 42 cases. On peut trouver une couverture du graphe avec 20 sommets, donc on ne peut pas recouvrir toute la figure à l'aide de dominos, ce qui constituerait un couplage parfait de 21 dominos.

☐ 7 – Démontrer le théorème de Hall (1935) :
G possède un couplage de cardinal $|X|$ si et seulement si pour tout $P \subset X$, on a $|V(P)| \geq |P|$
où $V(P) = \{y \in Y \mid \exists x \in P, \{x, y\} \in A\}$ est le voisinage de P.

Réponse 7. Soit M un couplage de cardinal $|X|$. Considérons $\{y \in Y \mid \exists x \in P, \{x, y\} \in M\}$.
Cet ensemble est inclus dans $V(P)$, et de cardinal $|P|$ par définition d'un couplage.
Soit C une converture par sommets minimale, dont le nombre de sommets dans X est maximal. S'il n'existe pas de couplage de cardinal $|X|$, alors $|C| < |X|$. Soit $x \in X \setminus C$, si x est isolé, alors $P = \{x\}$ vérifie $|V(P)| < |P|$. Sinon, $C \cap Y \neq \emptyset$, et $P = \{x \mid \exists y \in C, \{x, y\} \in A\}$ vérifie $|V(P)| < |P|$ par minimalité/maximalité de C.

☐ 8 – On considère le jeu à 2 joueurs suivant sur un graphe (pas nécessairement biparti) $G = (S, A)$. Les joueurs 1 et 2 choisissent chacun leur tour une arête pour former un chemin simple avec les arêtes déjà sélectionnées (on rappelle qu'un chemin simple ne peut pas passer plusieurs fois par le même sommet). Le premier joueur qui ne peut pas choisir d'arête a perdu. Montrer que si G possède un couplage parfait, le joueur 1 possède une stratégie gagnante.

Réponse 8. La stratégie du joueur 1 consiste à choisir toujours une arête du couplage parfait.
Pour prolonger un chemin, le joueur 2 ne peut pas choisir une arête du couplage parfait, puisque les deux extrémités du chemin terminent par une arête du couplage. Le joueur 1 en revanche, peut toujours choisir de prolonger le chemin en ajoutant à la suite de l'arête choisie par le joueur 2 une arête du couplage. Puisque le joueur 1 peut toujours jouer, et que le partie ne peut pas être infinie, il gagne nécessairement.

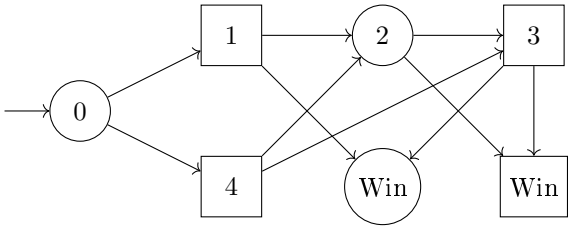
☐ 9 – On considère le jeu à 2 joueurs suivant sur un graphe biparti. On commence avec un couplage vide $M := \emptyset$. Chacun leur tour, les joueurs choisissent un chemin augmentant I pour le couplage M, et on met à jour $M := M \oplus I$, où $\oplus$ est la différence symétrique. Si un joueur ne peut plus jouer, il perd et l'autre joueur gagne. Déterminer si un joueur possède une stratégie gagnante, et si oui lequel.

Réponse 9. On peut remarquer que pour tout chemin augmentant I pour un couplage M, $|M \oplus I| = |M| + 1$.
Donc quels que soient les choix des joueurs, la partie se terminera au bout de n tours, où n est le cardinal d'un couplage de cardinal maximal. Si n est impair, le joueur 1 gagne, si n est pair, le joueur 2 gagne.

Partie II. Calcul des attracteurs (OCaml)

Implémentation d'un jeu d'accessibilité sur un graphe à deux joueurs en OCaml

On appelle *jeu* un graphe de ce genre :



Où les ronds sont les états appartiennant au joueur 1 et les carrés ceux du joueur 2. Quand un joueur possède un état, c'est lui qui choisit dans lequel des états possibles on va ensuite. Les objectifs sont pour le joueur 1 d'arriver à l'état (Win) et à l'état [Win] pour le joueur 2.
Formellement un jeu peut-être représenté par un 6-uplet $\langle V_1, V_2, i, E, \Omega_1, \Omega_2 \rangle$. Où $(V_1 \cup V_2, E)$ est un graphe avec V_j représentant les états controlés par le joueur j, $i \in V_1 \cup V_2$ est l'état initial, $\Omega_j \subset V_1 \cup V_2$ représente l'objectif du joueur j. On dit que le joueur j gagne si un des états de Ω_j est atteint à la fin de la partie.
On représentera un jeu de la façon suivante en OCaml, sans préciser pour l'instant le type que prendront nos états.

```
type 'a game =
  { state1 : 'a list ;
    state2 : 'a list ;
    init : 'a ;
    trans: 'a -> 'a list ;
    win1 : 'a list ;
    win2 : 'a list }
```

Le champ trans associe à un état, la liste de ses successeurs possibles.

☐ 10 – Donner la représentation en OCaml de l'exemple du dessin.

```
val example : int game
```

Réponse 10.

```
let example =
  let trans = function
  | 0 -> [1;4]
  | 1 -> [2;5]
  | 2 -> [3;6]
  | 3 -> [6]
  | 4 -> [2;3]
  | 5 | 6 -> []
  | _ -> raise (Invalid_argument "trans")
  in {
      state1 = [0;2;5];
      state2 = [1;3;4;6];
      init = 0;
      trans = trans;
      win1 = [5];
      win2 = [6];
    }
```

Une stratégie (sans mémoire) pour le joueur 1 est une fonction qui à chaque état controlé par joueur 1 associe une transition possible à partir de cet état. Une strategie serait représentée en OCaml par le type suivant

```
type 'a strategy = ('a -> 'a)
```

Une stratégie est dite *gagnante* si quel que soit la stratégie du second joueur, on atteint un de nos objectifs. Par exemple dans le jeu représenté sur le dessin, joueur 2 a une stratégie gagnante mais pas joueur 1.

☐ 11 – Représenter en OCaml la stratégie gagnante de joueur 2.

```
val winning_strat : int strategy
```

Réponse 11.

```
let winning_strat = function
  | 1 | 4 -> 2
  | 3 -> 6
  | _ -> raise (Invalid_argument "strat")
```

☐ 12 – Écrire un fonction execute qui étant donné un jeu et deux stratégies fait avancer la partie en suivant les instructions donnés par les stratégies et s'arrête quand un objectif a été atteint, dans ce cas là on renvoit cet état. Dans le cas où une stratégie propose une transition non autorisé il faudra lever une exception.

```
val execute : 'a game -> 'a strategy -> 'a strategy -> 'a
```

Si vous le voulez, vous pouvez d'abord écrire une fonction intermédiaire qui réalise seulement une étape. Vous pouvez tester cette fonction avec la strategie gagnante de joueur 2 et d'autre stratégie de joueur 1 et vérifier que l'on arrive toujours dans l'état gagnant de joueur 2.

Réponse 12.

```
(* Fonction mem du module List, recodée ici par souci de complétion
utilisée en tant que List.mem plus tard *)
let rec mem x = function
  | [] -> false
  | t::q -> x=t || mem x q

let execute g s1 s2 =
  let rec aux i =
    if List.mem i g.win1 || List.mem i g.win2
    then i
    else
      let t = if List.mem i g.state1 then s1 i else s2 i
      in
      if List.mem t (g.trans i) then aux t else failwith "not allowed"
  in aux g.init
```

Calcul des attracteurs

Pour décider si un joueur à une stratégie gagnante on procède par un calcul d'attracteur. L'ensemble $\mathcal{A}_i^j$ représente l'ensemble des états du jeu à partir desquels le joueur j possède une stratégie qui le fait gagner en i étapes ou moins. Le calcul se fait récursivement :

- Au rang 0, c'est exactement l'objectif : $\mathcal{A}_0^j = \Omega_j$
- Au rang $i+1$ on ajoute les états controlés par j où au moins un succeseur est gagnant et les états controlés par l'adversaire où tous (et au moins un) successeur est gagnant :

$$\mathcal{A}_{i+1}^j = \mathcal{A}_i^j \cup \{s \in V_j \mid \exists (s,s') \in E.\ s' \in \mathcal{A}_i^j\} \cup \{s \in V_{3-j} \mid \exists (s,s') \in E \text{ et } \forall (s,s') \in E.\ s' \in \mathcal{A}_i^j\}$$

On s'arrête à partir du rang i où l'attracteur ne grandit plus : $\mathcal{A}_{i+1}^j = \mathcal{A}_i^j$, car à ce moment on a atteint un point fixe et les états gagnant de j sont exactement les états de $\mathcal{A}_i^j$.
Pour représenter les ensembles et tester efficacement l'appartenance d'un élément à un ensemble, on implémente un ensemble par une table de hachage dont les clés sont les éléments de l'ensemble et les valeurs des booléens. Si un élément appartient à l'ensemble, il est associé à la valeur true. Si un élément n'appartient pas à l'ensemble, soit il est associé à la valeur false , soit il n'est pas utilisé comme clé dans la table de hachage. En OCaml elles ont le type Hashtbl.t et se manipule grace aux fonctions suivantes :

- Hashtbl.create n crée une table déstiné à contenire environ n élements ;
- Hashtbl.add t a b ajoute dans la table t l'élement ayant pour clé a et pointant vers b ;
- Hashtbl.find t a renvoie l'élement pointé par a, lève l'exception Not_found si la clé n'est pas présente dans la table ;
- Hashtbl.fold f t init calcul (f kN dN (... (f k1 d1 init))) où les k1 ... kN sont les clés et les d1 ... dN sont les valeurs de tout les élemements présent dans la table.

Pour representer les ensembles on pourra donc utiliser une table à valeur booléenne.

```
type 'a set = ('a,bool) Hashtbl.t
```

☐ 13 – Implémenter les fonctions de base sur les ensembles.

```
    val empty_set : unit -> 'a set
    val is_empty : 'a set -> bool
    val add : 'a set -> 'a -> unit
    val mem : 'a set -> 'a -> bool
    val remove : 'a set -> 'a -> unit
    val to_list : 'a set -> 'a list
```

Réponse 13.

```
let empty_set () = Hashtbl.create 10

let is_empty t = Hashtbl.fold (fun k v acc -> acc && not v) t true

let add t x = Hashtbl.add t x true

let mem t x = try Hashtbl.find t x with Not_found -> false

(* Je pense qu'il s'agit de ce qui était attendu, vu la doc qu'on nous donne de Hashtbl *)
(* compatible avec mem mais pas avec is_empty ou to_list *)
(* let remove t x = Hashtbl.add t x false *)

let remove t x = Hashtbl.remove t x

let to_list t = Hashtbl.fold (fun k v acc -> if v then k::acc else acc) t []
```

Pour implémenter le calcul de l'attracteur on va d'abord écrire deux fonctions qui serviront à verifier respectivement que tout les successeurs et au moins un successeur vérifie une certaine propriété.

☐ 14 – Écrire la fonction forall_succ qui étant donné un jeu, un état et une fonction booléene, renvoie vrai si tout les successeurs de cet états sont évalués à vrai par la fonction. De même écrire une fonction exists_succ qui renvoie vrai si au moins un successeur de l'état est évalué à vrai.

```
val forall_succ : 'a game -> 'a -> ('a -> bool) -> bool
val exists_succ : 'a game -> 'a -> ('a -> bool) -> bool
```

Réponse 14.

```
let forall_succ g s p =
  let rec parcours = function
    | [] -> true
    | t::q -> p t && parcours q
  in parcours (g.trans s)

let exists_succ g s p =
  let rec parcours = function
    | [] -> false
    | t::q -> p t || parcours q
  in parcours (g.trans s)
```

☐ 15 – Écrire une fonction step qui prend en argument un jeu, un joueur et l'attracteur au rang i, ajoute à cet ensemble les états qui doivent appartenir à l'attracteur du joueur donné au rang $i+1$ et renvoie vrai si des élements ont effectivement été ajouté.

```
val step : 'a game -> int -> 'a set -> bool
```

Réponse 15.

```
(* Fonction filter du module List, recodée ici par souci de complétion
utilisée en tant que List.filter plus tard *)
let rec filter p = function
  | [] -> []
  | t::q when p t -> t::filter p q
  | _::q -> filter p q

(* Fonction iter du module List, recodée ici par souci de complétion
utilisée en tant que List.iter plus tard *)
let rec iter f = function
  | [] -> ()
  | t::q -> f t; iter f q

let states g j =
  if j = 1 then g.state1 else g.state2

let step g j attr =
  let to_add1 =
    List.filter
      (fun s -> not (mem attr s) && exists_succ g s (mem attr))
      (states g j)
  and to_add2 =
    List.filter
      (fun s -> not (mem attr s) && (g.trans s <> []) && forall_succ g s (mem attr))
      (states g (3-j))
  in
  List.iter (add attr) to_add1;
  List.iter (add attr) to_add2;
  to_add1 <> [] || to_add2 <> []
```

☐ 16 – Écrire une fonction qui calcul l'attracteur pour un joueur donné. Tester la ensuite sur l'exemple déjà écrit.

```
val attractor : 'a game -> int -> 'a set
```

Réponse 16.

```
let wins g j =
  if j = 1 then g.win1 else g.win2

let attractor g j =
  let attr = empty_set () in
  List.iter (add attr) (wins g j);
  while step g j attr do () done;
  attr
```

Partie III. Test de primalité (C)

Dans cette partie, on utilisera le langage C en supposant qu'il n'y a pas de dépassement de capacité sur les entiers, et que la fonction `rand()` renvoie un nombre aléatoire entre 0 et M, où M est suffisamment grand pour que `rand() % k` renvoie un nombre uniformément aléatoire entre 0 et $k-1$.

Test de primalité de Fermat

Si un nombre n est premier, alors tout nombre a premier avec n vérifie $a^{n-1} = 1 \mod n$. Le test de primalité de Fermat est un algorithme probabiliste basé sur cette observation.

☐ 17 – Écrire une fonction `int exp(int a, int b)` qui calcule a^b en complexité en temps $O(\log(b))$, en supposant que l'addtion et la multiplication d'entiers s'effectue en temps constant.

Réponse 17.

C
```c
int exp(int a, int b){
  int acc = 1;
  while (b > 0){
    if (b % 2 == 1) {
      acc = acc * a;
    }
    a = a * a;
    b = b / 2;
  }
  return acc;
}
```

☐ 18 – Écrire une fonction `bool fermat_witness(int n, int a)` qui renvoie `true` si a est un témoin de primalité de Fermat de n (i.e., $a^{n-1} = 1 \mod n$) et `false` sinon.

Réponse 18.

C
```c
int mod_exp(int a, int b, int n){
  int acc = 1;
  while (b > 0) {
    if (b % 2 == 1) {
      acc = (acc * a) % n;
    }
    a = (a * a) % n;
    b = b / 2;
  }
  return acc;
}

bool fermat_witness(int n, int a){
  return mod_exp(a, n-1, n) == 1;
}
```

☐ 19 – Écrire une fonction `bool fermat_primality_test(int n, int k)` qui tire aléatoirement k nombres entre 2 et $n-1$ inclus et qui renvoie `true` si tous ces nombres sont des témoins de primalité de Fermat de n, et `false` si au moins l'un d'entre eux ne l'est pas.

Réponse 19.

```C
bool fermat_primality_test(int n, int k){
  for (int i = 0; i < k; i = i + 1){
    int a = 2 + (rand() % (n-2));
    if (!fermat_witness(n, a)) {
      return false;
    }
  }
  return true;
}
```

Cet algorithme se révèle très efficace, car la moitié des nombres a entre 2 et $n-1$ sont des témoins de non-primalité si n n'est pas premier, sauf pour une famille de nombres particuliers : les nombres de Carmichael. Le test suivant permet d'éviter ces faux nombres premiers.

Test de primalité de Miller-Rabin

Soit p un nombre premier impair. Soit s non nul et d impair deux entiers vérifiant $p-1=2^s d$. Alors pour tout entier a qui n'est pas un multiple de p, on a :
$a^{p-1}=1 \mod p$, et 1 et -1 sont les seules racines carrées de 1 modulo p. Donc

$$a^d = 1 \mod p \quad \vee \quad \exists r \in [\![0, s-1]\!], \quad a^{2^r d} = -1 \mod p.$$

Donc pour tout entiers s non nul, d impair, $n=2^s d+1$ et $1<a<n$,

$$\left(a^d \neq 1 \mod n \wedge \forall r \in [\![0, s-1]\!], a^{2^r d} \neq -1 \mod n\right) \rightarrow n \text{ est composite.}$$

Dans ce cas, on appelle a un témoin de non-primalité de Miller pour n.

☐ 20 – Écrire une fonction `void decompose(int n, int *s, int *d)` qui à partir d'un nombre n donné, calcule les valeurs de s et de d et change les valeurs pointées par `s` et `d`.

Réponse 20.

```C
void decompose(int n, int *s, int *d){
  *s = 0;
  *d = n;
  while (*d % 2 == 0){
    *s = *s + 1;
    *d = *d / 2;
  }
}
```

☐ 21 – Écrire une fonction `bool not_miller_witness(int n, int a)` qui étant donné n et a renvoie `false` si a est un témoin de non-primalité de n, et `true` sinon.

Réponse 21.

C
```c
bool not_miller_witness(int n, int a){
  int s, d;
  decompose(n, &s, &d);
  /* Remarque : on aurai pu prendre s et d en argument pour éviter
     de les recaculter à chaque test */
  int acc = mod_exp(a, d, n);
  if (acc == 1) { return true; }
  for (int i = 0; i < s; i = i + 1){
    if (acc == -1) { return true; }
    acc = (acc * acc) % n;
  }
  return false;
}
```

Si n est un nombre impair composé, au moins 3/4 des entiers entre 2 et $n-1$ inclus sont des témoins de non-primalité de Miller pour n. On peut donc en déduire un algorithme de type Monte-Carlo dont la probabilité de correction est supérieure à 0.95.

☐ 22 – Écrire une fonction `bool miller_rabin95(int n)` qui renvoie `true` si n est premier, et qui renvoie `false` avec une probabilité supérieure à 0.95 si n est composé.

Réponse 22. Si n est composé, un seul test de `not_miller_witness` avec un a aléatoire donnera un mauvais résultat avec une probabilité au plus $\frac{1}{4}$, donc avec 3 tests indépendants, notre algorithme se trompera avec une probabilité $\frac{1}{64} < 0.05$.

C
```c
bool miller_rabin95(int n){
  for (int i = 0; i < 3; i = i + 1){
    int a = 2 + (rand() % (n-2));
    if (!not_miller_witness(n, a)) {
      return false;
    }
  }
  return true;
}
```

Ce test de primalité à été proposée par Michael Rabin comme variante de l'algorithme proposé par Gary L. Miller dont la preuve de correction repose sur l'hypothèse de Riemann généralisée. Il s'agit dans la version originale de tester les entiers compris entre 2 et $O(\log(n)^2)$.

Fin de l'épreuve