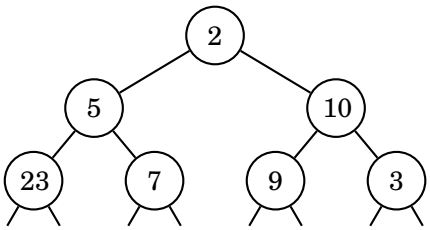
Corrigé du devoir n°6

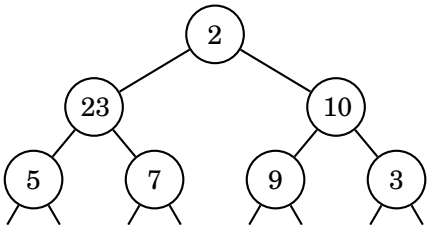
1 Exercice 0

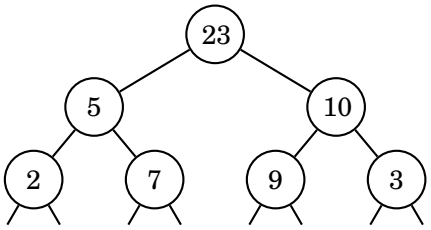
Dans cet exercice, on considère le tableau suivant : $t = [2; 5; 10; 23; 7; 9; 3]$. On va faire quelques manipulations simples sur les tas max.

1. *En utilisant la méthode de complexité linéaire vue en cours, construire un tas max qui contient les valeurs de ce tableau.*
   On construit d'abord un arbre binaire complet avec les valeurs.

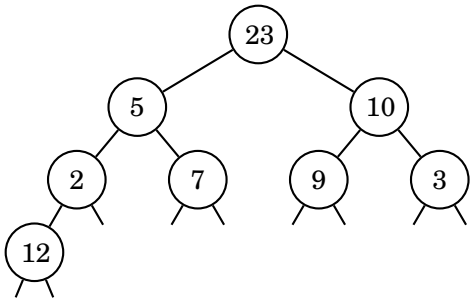


   Ensuite on percole tous les sommets vers le bas de bas en haut de droite à gauche. Il n'y a que deux étapes intéressantes, pour le noeud 5 et le noeud 2 :

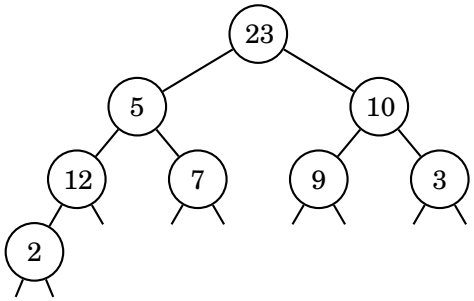


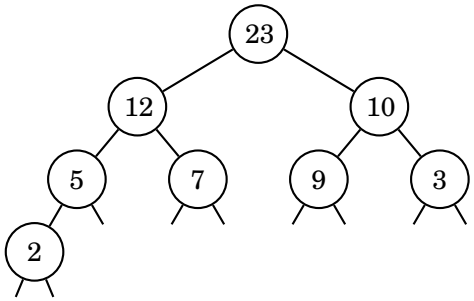


2. *Ajouter à votre tas la valeur 12.*
   On place la valeur 12 à la prochaine feuille :



   On percole ensuite 12 vers le haut :





   Et c'est fini.

3. *Effectuer le tri par tas sur le tableau t originel.*
   On donne les étapes dans le tableau :
   $[2; 5; 10; 23; 7; 9; 3]$ devient le tas $[23; 5; 10; 2; 7; 9; 3]$

Ensuite on "retire" le 23 et on percole le 3 vers le bas : $[3;5;10;2;7;9||23]$ devient $[10;5;3;2;7;9||23]$ puis $[10;5;9;2;7;3||23]$.
On recommence : $[3;5;9;2;7||10;23]$ devient $[9;5;3;2;7||10;23]$.
On recommence : $[7;5;3;2||9;10;23]$.
On recommence : $[2;5;3||7;9;10;23]$ devient $[5;2;3||7;9;10;23]$.
On recommence : $[3;2||5;7;9;10;23]$
On recommence : $[2||3;5;7;9;10;23]$
Et finalement on obtient $[2;3;5;7;9;10;23]$

2 Exercice 1

1. *Dans le contexte des tables de hachage, qu'est-ce qu'une fonction de hachage ?*
   C'est une fonction qui prend en entrée une clé (ici le tableau du numéro de sécurité sociale) et renvoie un numéro entre 0 et $m-1$ (une case de la table de hachage).
2. *Pourquoi la fonction de hachage qui consiste à prendre le premier chiffre modulo m est elle mauvaise ?*
   Comme il n'y a que deux valeurs, tous les patients seraient hachés soit dans la case 1, soit dans la case 2. Il y a donc énormément de **collisions**.
3. *Proposer une autre fonction de hachage h_2 qui répartit bien les données.*
   Il y a plein de possibilités, il faut juste faire attention à avoir plus de 200000 valeurs possibles avant d'appliquer le modulo.

$$h_2 : tab \mapsto \sum_{i=0}^{12} tab[i] * 10^i \mod m$$

$tab \mapsto \sum_{i=0}^{12} tab[i] * 10^i$ est une injection dans $\mathbb{N}$ car la représentation en base 10 est unique (et les bits de vérification sont intégralement définis par les autres chiffres). En appliquant le modulo m, on répartit optimalement, au sens que chaque i a à peu près le même nombre de clés qui sont hachées en i.

La table

4. *Implémenter la fonction `void ajoute(tableHachage* t, int* cle)` qui ajoute un nouveau patient à la table. Un nouveau patient est un patient qui achète son premier mois de traitement.*

```c
void ajoute(tableHachage* t, int* cle){
  int hash = h1(cle); //Premier hachage
  if (t->donnees[hash].cle==NULL){
    t->donnees[hash].cle = cle;
    t->donnees[hash].valeur = 1;
  }
  else{
    int hash2 = h2(cle); //Deuxième hachage
    if (t->donnees[hash2].cle==NULL){
      t->donnees[hash2].cle = cle;
      t->donnees[hash2].valeur = 1;
    }
    else{ //Les deux hachages ne marchent pas
      fprintf(stderr, "les deux hachages ont échoué");
      assert(false);
    }
  }
  t->remplissage += 1;
}
```

5. *Implémenter la fonction qui ajoute un mois au patient donné. Si le patient n'est pas trouvé, une erreur sera levée.*
   Pour cette question il nous faut une fonction auxiliaire (ou un bout de fonction) qui teste si deux tableaux sont égaux (et non ça ne marche pas avec ==)

```c
bool egalite_tableaux(int* tab1, int* tab2){
  //Attention si un des deux est NULL
  if (tab1==NULL || tab2==NULL){return false;}
  res = true;
  for (int i=0; i<14; i+=1){
```

```
    res = res && (tab1[i]==tab2[i]);
  }

  return res;
}

void modifie(tableHachage* t, int* cle){
  int hash = h1(cle); //Premier hachage
  if (egalite_tableaux(cle, t->donnees[hash].cle)){
    t->donnees[hash].valeur += 1;
  }
  else{
    int hash2 = h2(cle); //Deuxième hachage
    if (egalite_tableaux(cle, t->donnees[hash2].cle)){
      t->donnees[hash2].valeur += 1;
    }
    else{ //Les deux hachages ne marchent pas
      fprintf(stderr, "clé non trouvée");
      assert(false);
    }
  }
}
```

6. Pour finir, on suppose que le dictionnaire a été rempli et mis à jour durant toute la durée de l'étude, écrire une fonction `float resultat_etude(tableHachage* t)` qui calcule le temps moyen de traitement (en mois).

```
float resultat_etude(tableHachage* t){
  int res = 0;
  int nb_cases = 0;
  for(int i=0;i<m;i+=1){ //On parcourt tout le tableau données
    if (t->donnees[i].cle!=NULL){ //Les cases vides ne nous intéressent pas
      res+=t->donnees[i].valeur;
      nb_cases+=1;
    }
  }
  return res/nb_cases;
}
```

3 Exercice 2

D'après le sujet CCINP option MP 2014

1. *Considérons que l'un des Sphinx fait une suite de n déclarations, proposer une formule du calcul des propositions qui représente la règle qu'il respecte.*
   A_0 et A_{n-1} doivent être soit toutes les deux vraies, soit toutes les deux fausses. Les autres A_i ont la valeur de vérité contraire.
   On peut écrire ceci ainsi :
   $$(A_0 \wedge A_{n-1} \wedge \bigwedge_{i=1}^{n-2} \neg A_i) \vee (\neg A_0 \wedge \neg A_{n-1} \wedge \bigwedge_{i=1}^{n-2} A_i)$$
   On peut remarquer que les deux clauses ne peuvent pas arriver en même temps, donc le ou est gratuitement exclusif.

2. *Représenter les déclarations des deux Sphinx sous la forme de formules du calcul des propositions P_1, P_2, S_1 et S_2 dépendant des variables g, m et d.*

$$P_1 = g$$

$$P_2 = m \vee \neg d$$

$$S_1 = \neg g \wedge \neg m$$

$$S_2 = (g \vee d) \rightarrow m$$

3. *Appliquer la règle respectée par les Sphinx , comme vous l'avez proposé pour la question 1.*
   On utilise la question 1 :

$$P = (P_1 \wedge P_2) \vee (\neg P_1 \wedge \neg P_2)$$

$$S = (S_1 \wedge S_2) \vee (\neg S_1 \wedge \neg S_2)$$

   Et les deux sphinx doivent vérifier la règle en même temps :

$$R = P \wedge S$$

4. *En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer quel est (ou quels sont) le (ou les) escalier(s) qui est (ou sont) sûr(s) ?*

$[\|g\|]_v$	$[\|m\|]_v$	$[\|d\|]_v$	$[\|P_1\|]_v$	$[\|P_2\|]_v$	$[\|P\|]_v$	$[\|S_1\|]_v$	$[\|S_2\|]_v$	$[\|S\|]_v$	$[\|R\|]_v$
V	V	V	V	V	V	F	V	F	F
V	V	F	V	V	V	F	V	F	F
V	F	V	V	F	F	F	F	V	F
V	F	F	V	V	V	F	F	V	V
F	V	V	F	V	F	F	V	F	F
F	V	F	F	V	F	F	V	F	F
F	F	V	F	F	V	V	F	F	F
F	F	F	F	V	F	V	V	V	F

   Ici on trouve que R n'est vrai que pour une seule valuation. Donc le seul moyen pour que les Sphinx aient bien respecté les règles est que l'escalier de gauche soit sûr et les deux autres soient pas sûrs.
   On se dirige donc vers l'escalier gauche pour continuer l'aventure.

5. *Représenter les déclarations du Sphinx sous la forme de formules du calcul des propositions P_3, P_4 et P_5 dépendant des variables r, v et b.*

$$P_3 = \neg r \vee v$$
$$P_4 = (r \wedge v) \rightarrow \neg b$$
$$P_5 = \neg v \wedge b$$

6. *Appliquer la règle respectée par le premier Sphinx que vous avez proposée pour la question 1.*

$$P' = (P_3 \wedge \neg P_4 \wedge P_5) \vee (\neg P_3 \wedge P_4 \wedge \neg P_5)$$
$$= [(\neg r \vee v) \wedge \neg((r \wedge v) \rightarrow \neg b) \wedge (\neg v \wedge b)] \vee [\neg(\neg r \vee v) \wedge ((r \wedge v) \rightarrow \neg b) \wedge \neg(\neg v \wedge b)]$$

7. *En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer quelle est (ou quelles sont) la (ou les) porte(s) qui est (ou sont) sûre(s).*
   On simplifie d'abord les non, puis on va distribuer les et et les ou. Quand on croise une tautologie ou une antilogie évidente, on remplace par $\top$ et $\bot$ (et on les supprime ensuite).

$$\begin{aligned}
P' &= [(\neg r \vee v) \wedge (r \wedge v \wedge b) \wedge (\neg v \wedge b)] \vee [(r \wedge \neg v) \wedge (\neg r \vee \neg v \vee \neg b) \wedge (v \vee \neg b)] \\
&= [(\neg r \vee v) \wedge (r \wedge v \wedge b \wedge \neg v \wedge b)] \vee [(r \wedge \neg v) \wedge (\neg r \vee \neg v \vee \neg b) \wedge (v \vee \neg b)] \\
&= [(\neg r \vee v) \wedge \bot] \vee \dots \\
&= \bot \vee \dots \\
&= (r \wedge \neg v) \wedge (\neg r \vee \neg v \vee \neg b) \wedge (v \vee \neg b) \\
&= [(r \wedge \neg v \wedge \neg r) \vee (r \wedge \neg v \wedge \neg v) \vee (r \wedge \neg v \wedge \neg b)] \wedge (v \vee \neg b) \\
&= [(r \wedge \neg v \wedge \neg v) \vee (r \wedge \neg v \wedge \neg b)] \wedge (v \vee \neg b) \\
&= (r \wedge \neg v \wedge (v \vee \neg b)) \vee (r \wedge \neg v \wedge \neg b \wedge (v \vee \neg b)) \\
&= (r \wedge \neg v \wedge v) \vee (r \wedge \neg v \wedge \neg b) \vee (r \wedge \neg v \wedge \neg b \wedge v) \vee (r \wedge \neg v \wedge \neg b \wedge \neg b) \\
&= (r \wedge \neg v \wedge \neg b)
\end{aligned}$$

   La seule solution correcte est donc que la porte rouge soit sûre et que les deux autres portes ne soient pas sûres.

4 Problème

D'après un sujet d'Antoine Grospellier

1. Préambule

1. *Donner la définition formelle d'un graphe non orienté.*
   Une graphe est la donnée d'un couple (S, A) où S est l'ensemble des sommets (les noms qu'on leur donne du moins) et A est l'ensemble des arêtes. Une arêtes est un ensemble de taille 2 $\{x, y\}$. Ici on a dit qu'on forçait $x \neq y$.

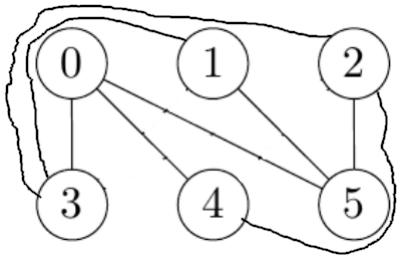
2. *Donner, en Ocaml, les listes d'adjacence associées aux graphes des figures 2 et 3.*

```
let fig2 = [|[1;2;3;4];[0;2;3;4];[0;1;3;4];[0;1;2];[0;1;2]|];;

let fig3 = [|[1;2;3;4];[0;2;3;4];[0;1;3;4];[0;1;2];[0;1;2]|];;
```

   On peut remarquer que c'est la même chose : les graphes des figures 2 et 3 représentent le même graphe, dessiné un peu différemment.

3. *Montrer que le graphe de la figure 5 est planaire.*
   Il suffit de bouger les arêtes pour ne plus qu'elles se croisent. Voici un magnifique dessin d'illustration :



2. Transformation de graphes

Opérations sur les listes

4. *Écrire une fonction* `supprimer : int list -> int -> int list` *qui prend en entrée une liste l et un élément i_0 et renvoie une liste contenant les mêmes éléments que l où i_0 a été supprimé (toutes ses occurences).*

```
let rec supprimer l i0 = match l with
  |[] -> []
  |t::q when t=i0 -> supprimer q i0
  |t::q -> t::(supprimer q i0);;
```

5. *Écrire une fonction* `decaler : int list -> int -> int list` *qui prend en entrée une liste l et un élément i_0 et renvoie une liste contenant les mêmes éléments que l en remplaçant tous les éléments $i > i_0$ par $i - 1$.*

```
let rec decaler l i0 = match l with
  |[] -> []
  |t::q when t>i0 -> (t-1) :: (decaler q i0)
  |t::q -> t::(decaler q i0);;
```

6. *Écrire une fonction* `supprime_doublons : int list -> int list` *qui prend en entrée une liste l et renvoie une liste qui est similaire à l mais où aucun élément n'est présent en double.*
   Pour obtenir la complexité linéaire, en notant n la taille de la liste, il faut créer un tableau de taille n tel que sa case i contient l'information de si on a déjà vu i.

```
let supprime_doublons l =
  let tab = Array.make (List.length l) false in
  let rec aux l = match l with
    |[] -> []
    |t::q when tab.(t) -> aux q
    |t::q -> tab.(t)<-true; t::(aux q)
  in aux l;;
```

   La création du tableau de taille n est linéaire en n et la fonction auxiliaire ne fait que parcourir la liste, ce qui est linéaire.

Opérations sur les graphes

7. *Écrire une fonction* `supprime_arrete : graphe -> int -> int -> graphe` *qui prend en entrée* G_1, s_1 *et* s_2 *et renvoie le graphe obtenu après suppression de l'arête* $\{s_1, s_2\}$.

```
let supprime_arrete g1 s1 s2 =
  let n = Array.length g1 in
  let g2 = Array.make n [] in
  for i=0 to n-1 do //On copie les listes originelles, sauf pour s1 et s2 ou on supprime un element.
    if i=s1 then g2.(i) <- supprime g1.(i) s2
    else if i=s1 then g2.(i) <- supprime g1.(i) s1
    else g2.(i)<-g1.(i)
  done;
  g2;;
```

8. *Écrire une fonction* `supprime_sommet : graphe -> int -> graphe` *qui prend en entrée* G_1 *et* s_0 *et renvoie le graphe obtenu après suppression du sommet* s_0.

```
let supprime_sommet g1 s0 =
  let n = Array.length g1 in
  let g2 = Array.make (n-1) [] in //Le graphe résultat a un sommet de moins
  for i=0 to n-1 do //On décale les indices et on supprime toute mention de s0
    if i<s0 then g2.(i)<- decaler (supprime g1.(i) s0) s0
    else if i>s0 then g2.(i-1)<- decaler (supprime g1.(i) s0) s0
    //Il faut aussi décaler la positions des listes des sommets >s0
  done;
  g2;;
```

9. *Écrire une fonction* `contracte_arrete : graphe -> int -> int -> graphe` *qui prend en entrée* G_1, s_1 *et* s_2 *et renvoie le graphe obtenu après contraction de* $\{s_1, s_2\}$.

```
let contracte_arrete g s1 s2 =
  let t = min s1 s2 in
  let s0 = max s1 s2 in
  let n = Array.length g1 in
  let g2 = Array.make (n-1) [] in //Le graphe résultat a un sommet de moins

  //La liste de t : on concatene celle de s1 et s2, on supprime les doublons et on supprime s1 et s2
  g2.(t) <- decaler (supprime_doublons (supprime g1.(s1) s2)@(supprime g1.(s2) s1)) s0;

  for i=0 to n-1 do
    if i<s0 && i<>t then g2.(i) <- decaler (remplacer g1.(i) s0 t) s0
    else if i>s0 then g2.(i-1) <- decaler (remplacer g1.(i) s0 t) s0
    //Il faut aussi décaler la positions des listes des sommets >s0
  done;

  g2;;
```

3. Graphes complets

Graphes non-bipartis

10. *Écrire une fonction* `make_complet : int -> graphe` *qui prend en entrée n et renvoie le graphe complet à n noeuds (sous forme de liste d'adjacence).*

```
let make_complet n =
  let rec presque_range i exclus = (* Crée la liste qui contient tous les entiers de 0 à
n-1, sauf exclus *)
    if i=n then []
    else if i=exclus then presque_range (i+1) exclus
    else i::(presque_range (i+1) exclus)
  in
  let g = Array.make n [] in
  for i = 0 to n-1 do
    g.(i) <- presque_range 0 i
  done;
  g;;
```

11. *Écrire une fonction* `est_complet : graphe -> bool` *qui prend en entrée une graphe et indique s'il est complet.*

```
let est_complet g n =
  let m = Array.length g in
```

```
  let res = ref (m=n) in (*On vérifie que g a n noeuds*)
  for i=0 to m-1 do (*Il suffit de vérifier que chaque noeud a n-2 voisins (pas de doublons) *)
    if List.length g.(i) <> m-2 then res:=false
  done;
  !res;;
```

12. *Quelle est la complexité de votre fonction ?* Le calcul de la longueur est en O de la taille de la liste. Si le graphe est complet, la taille est toujours un $O(n)$.
On fait donc n fois un $O(n)$. La complexité est quadratique.

Graphes bipartis

13. *Parmi les graphes d'exemples, lesquels sont bipartis ? (sans justification)*
Les graphes 4,5 et 6.
14. *Montrer que le graphe de la figure 1 n'est pas biparti.*
La meilleure manière de le voir est qu'il y a une cycle 0 - 2 - 3. Supposons qu'on mette 0 dans S_1 (ou S_2 sans perte de généralité), alors 2 et 3 doivent aller dans S_2, et c'est impossible car ils sont connectés.
15. *Écrire une fonction* `make_biparti_complet : int -> int -> graphe` *qui prend en entrée deux entiers n_1 et n_2 et renvoie un graphe biparti complet avec $|S_1| = n_1$ et $|S_2| = n_2$.*

```
let make_biparti_complet n1 n2 =
  let g = Array.make (n1+n2) [] in
  (*Comme ça nous arrange, les sommets de 0 à n1-1 seront ceux de S1*)
  let rec range a b = (*Crée la liste des entiers entre a et b inclus*)
    if a=b then [b]
    else a::(range (a+1) b)
  in

  for i=0 to n1-1 do
    g.(i) <- range n1 (n1+n2-1)
  done;
  for i=n1 to n1+n2-1 do
    g.(i) <- range 0 (n1-1)
  done;
  g;;
```

Reconnaissance des graphes bipartis complets

16. *Appliquer le pseudo code au graphe de la figure 1 et de la figure 5.*
Sur le graphe de la figure 1, on colorie 2,3 et 4 avec la couleur 1, puis peu importe celui qui est en haut de la file, il a un voisin de la même couleur. Donc res=false à la fin.
Sur le graphe de la figure 5, on obtient res=true et couleur = [0 ;0 ;0 ;1 ;1 ;1].
17. *Quel est l'algorithme sous-jacent à cette méthode ?*
C'est un parcours en largeur.
18. *Une fois qu'on a les couleurs, comment déduire les deux parties de l'ensemble S ?*
On prend S_1 comme les sommets coloriés en 0 et S_2 comme les sommets coloriés en 1.
19. *Implémenter le pseudo code en Ocaml.*

```
let est_biparti g =
  let f = Queue.create () in
  Queue.push 0 f;
  let n = Array.length g in
  let couleur = Array.make n -1 in
  couleur.(0) <- 0;
  let res = ref true in

  let rec parcours_voisins l c = match l with (*On parcours la liste des voisins*)
    |[] -> ()
    |t::q -> if couleur.(t) = c then res:=false
             else if couleur.(t) = -1 then begin
                                             couleur.(t)<-1-c;
                                             Queue.push t f;
                                             parcours_voisins q c
                                             end
  in
```

```
  while !res && not (Queue.is_empty f) do
    let s = Queue.pop f in
    let c = couleur.(s) in
    parcours_voisins g.(s) c
  done;

  (res,couleur);;
```

20. *En déduire une fonction* `est_biparti_complet : graphe -> bool` *qui prend en entrée un graphe g et deux entiers n_1 et n_2 et indique si le graphe est le biparti complet avec $|S_1| = n_1$ et $|S_2| = n_2$.*

```
let est_biparti_complet g n1 n2=
  let res, couleur = est_biparti g in
  let n = Array.length g in
  if not res then false (*graphe pas biparti*)
  else begin
    let m1 = ref 0 in
    for i=0 to n-1 do
      if couleur.(i) = 0 then m1:=!m1+1
    done;
    let m2 = n-!m1 in

    let res2 = ref true in
    for i=0 to n-1 do
      if couleur.(i)=0 then res := !res && (List.length g.(i) = m2)
      else res := !res && (List.length g.(i) = !m1)
    done;
    !res2 && (!m1=n1) && (m2=n2)
  end;;
```

4. Caractérisation des graphes planaires

21. *Montrer qu'il est possible qu'un graphe biparti ait comme mineur K_1 et qu'il est possible qu'une graphe non-biparti ait comme mineur K_2.* Si on considère le graphe biparti complet avec $n_1 = 5$ et $n_2 = 5$, alors en contractant chaque sommet de S_1 avec un sommet de S_2 (en faisant une bijection entre les deux), on obtient K_1.
Si on considère un graphe complet à 6 sommets, on peut supprimer des arêtes et obtenir K_2.

22. *En utilisant le théorème, répondre aux questions suivantes :*

(a) *Soit $n \in \mathbb{N}$ et G un graphe complet à n sommets. Le graphe G est-il planaire ?*
Si $n \geq 5$, en retirant un sommet $n-5$ fois on obtient K_1 comme mineur de G.
Si $n < 5$ alors comme K_1 et K_2 ont plus de 5 sommets et que les transformations ne permettent pas de rajouter des sommets, le graphe est nécessairement planaire.

(b) *Soit $n_1, n_2 \in \mathbb{N}$ et G une graphe biparti complet dont les parties sont de taille n_1 et n_2. Le graphe G est-il planaire ?*

- Si n_1 et n_2 sont plus grands que 3, alors on peut supprimer des sommets jusqu'à avoir S_1 de taille 3 et S_2 de taille 3. La complétude du graphe est préservée, donc K_2 est un mineur du graphe.
- Si $n_1 + n_2 < 5$ alors on a pas assez de sommets pour que K_1 ou K_2 soit un mineur et on ne peut pas en rajouter.
- Enfin si $n_1 + n_2 \geq 5$ et n_1 ou n_2 est strictement inférieure à 3, effectuer une suppression d'arête ou de sommet garde un graphe biparti mais on a toujours n_1 ou n_2 strictement inférieur à 3. Si on effectue une contraction, le graphe n'est plus biparti.
  On est donc assurés que K_2 n'est pas un mineur.
  On va également montrer que K_1 n'est pas un mineur.
  Supposons sans perte de généralité que c'est n_1 qui est strictement inférieur à 3. Si $n_1 = 0$, le graphe n'a pas d'arête et est planaire.
  Sinon considèrons un mineur quelconque à 5 sommet de G, nommé G', alors parmi ces 5 sommets il y en a au plus deux qui sont issus de la contraction des sommets de S_1 avec des sommets de S_2. On les laisse de côté pour considérer deux autres sommets (qui étaient dans S_2) a et b. a et b ne sont pas connectés par une arête dans G et ils ne le sont pas non plus dans G'. En effet les opérations qu'on aurait pu leur appliquer entre G et G' sont :
  - les supprimer, ce qui n'est pas arrivé

- supprimer une de leur arêtes (y compris en supprimant un de leur voisins), ce qui n'aide pas
- les contracter avec leurs voisins. Or leurs voisins sont les sommets de S_1 et par hypothèse la contraction n'a pas eu lieu.

Comme a et b ne sont pas connectés, G' n'est pas K_1.

Conclusion, G est planaire.

23. *Écrire une fonction* `mineurs_ordre1 : graphe -> graphe list` *qui prend en entrée G et renvoie la liste des mineurs d'ordre 1 de G.*

```
let mineurs_ordre1 g =
  let n = Array.length g in
  let res = ref [] in

  let rec traite_aretes s1 l = match l with
    |[] -> ()
    |s2::q -> if s2<s1 then (res:=(supprime_arrete g s1 s2)::!res;
                             res:=(contracte_arrete g s1 s2)::!res;)
              (*En testant si s2<s1, on évite de rajouter deux fois le même mineur*)
              traite_aretes s1 q
  in

  for i=0 to n-1 do
    res:= (supprime_sommet g i)::!res; (*Supprimer les sommets*)
    (*On en profite pour supprimer les arêtes et contracter les arêtes*)
    traite_aretes g.(i)
  done;

  !res;;
```

24. *En déduire une fonction* `est_planaire: graphe -> bool` *qui détermine si le graphe donné en entrée est planaire.*

```
let rec est_planaire g =
  let rec aux l = match l with
    |[] -> true
    |gg::q -> if Array.length g < 5 then false (*le graphe est trop petit*)
              else if est_complet gg 5 then false (*c'est K1*)
              else if est_biparti_complet gg 3 3 then false (*c'est K2*)
              else if not est_planaire gg then false
              else aux q
  in
  aux mineurs_ordre1 g;;
```