Entraînements CCINP : corrigés

Pour les exercices B, les codes sources sont également disponibles.

Exercices de type B

Exercice 1 *HORNSAT (exo 2024)*

1. F_1 et F_3 sont des formules de Horn mais pas F_2, ce qui est implicitement suggéré par le code.
2. Avec notre modélisation, la clause vide est `(None, [])` d'où :

```
let avoir_clause_vide (f:formule_horn) :bool =
  List.mem (None, []) f
```

3. La formule F_1 est satisfiable d'après cet algorithme mais pas F_3.

   Techniquement on peut arrêter les propagations dès qu'on produit une clause vide.

4. On propose :

```
let rec trouver_clause_unitaire (f:formule_horn) :int option = match f with
|[] -> None
|(Some v,l)::q -> if l = [] then Some v else trouver_clause_unitaire q
|_::q -> trouver_clause_unitaire q
```

5. La fonction auxiliaire `enlever_neg` supprime si elle existe la seule occurrence d'un entier dans une liste d'entiers (on utilise ici l'hypothèse selon laquelle il n'y a pas de littéral en doublon dans nos clauses de Horn) puis on applique le principe décrit par l'énoncé.

```
let rec propager (f:formule_horn) (v:int) :formule_horn =
let rec enlever_neg (l:int list) :int list = match l with
  |[] -> []
  |t::q when t = v -> q
  |t::q -> t::(enlever_neg q)
in
match f with
|[] -> []
|(i,l)::q -> match i with
               |Some t when t = v -> propager q v
               |_ -> (i, enlever_neg l)::(propager q v)
```

   La formule résultant d'une propagation dans une formule de Horn reste une formule de Horn : c'est toujours une FNC et on ne fait que supprimer des clauses / littéraux (donc s'il y avait au plus un littéral positif par clause, en supprimant des choses cette propriété est conservée).

6. On applique l'algorithme donné par l'énoncé en imbriquant les fonctions précédentes :

```
let rec etre_satisfiable (f:formule_horn) :bool =
match (trouver_clause_unitaire f) with
|None -> not (avoir_clause_vide f)
|Some v -> etre_satisfiable (propager f v)
```

7. La fonction `trouver_clause_unitaire` est linéaire en n la taille de la formule en entrée. Une propagation se fait aussi linéairement en la taille de la formule dans laquelle on propage. Comme une

propagation supprime au moins une clause de la formule (la clause unitaire responsable de la propagation), on fera au plus m étapes de "recherche de clause unitaire + propagation" où m est le nombre de clauses.

De plus, la recherche d'une clause vide dans une formule se fait linéairement en le nombre de clauses. On aboutit donc grossièrement (c'est-à-dire sans compter qu'au fil des propagations la taille de la formule réduit - de toutes façons ça ne changerait probablement pas grand chose) à une complexité pour `etre_satisfiable` en $O(mn+m) = O(n^2)$.

On en déduit que `HORN-SAT` $\in$ P alors qu'on sait que `SAT` est NP-complet. Sous réserve que P $\neq$ NP, étudier la satifiabilité de formules de Horn est donc un problème bien plus facile que lorsqu'il n'y a pas d'hypothèse sur les formules.

8. a) La valuation qui assigne faux à toutes les variables convient. En effet, s'il n'y a ni clause unitaire ni clause vide, toutes les clauses contiennent au moins une variable niée.

   b) Si F est une formule de Horn, notons $\Pi(F)$ la formule de Horn obtenue après toutes les propragations successives de la boucle tant que dans l'algorithme $\mathcal{A}$. La propriété de l'énoncé permet de montrer par récurrence que F et $\Pi(F)$ sont équisatisfiables.

   Or $\Pi(F)$ est satisfiable si et seulement si cette formule de Horn ne contient pas la clause vide : le sens direct est immédiat puisque une FNC contenant une clause vide n'est jamais satisfiable (cette clause ne l'étant pas) ; le sens réciproque est fourni par la question 7a.

9. Le principe est de garder trace des variables que l'on propage : on les assigne toutes à vrai. Toutes les variables non propagées sont quant à elles assignées à faux.

Exercice 2 *Mots de Dyck (exo 2024)*

1. Une solution et d'implémenter une factorielle. On peut aussi simplifier la formule donnée (ce qui permet de repousser un peu plus loin le dépassement).

```
uint64_t fact(int n) {
    uint64_t r = 1;
    for(int i = 2; i <= n; i = i + 1){
        r = r * i;
    }
    return r;
}

uint64_t catalan1(int n) {
    return fact(2*n)/fact(n+1)/fact(n);
```

2. On obtient un dépassement d'entier. Expérimentalement, si on utilise la fonction factorielle, le résultat devient faux pour $n = 11$. En simplifiant la formule en $2n \times ... \times (n+2)/n!$, c'est pour $n = 16$ que ça coince. Dans tous les cas, le dépassement finira par arriver provoquant un comportement indéfini.

3. On utilise un compteur qui indique le nombre de parenthèses ouvertes : on l'incrémente lorsqu'on rencontre une parenthèse ouvrante et on le décrémente lorsqu'on rencontre une parenthèse fermante. Si ce compteur devient négatif, le mot n'est pas bien parenthésé puisqu'il y a trop de parenthèses fermantes et on peut donc interrompre l'exécution (même si on peut se passer de cette subtilité). En fin de décompte, le compteur devrait être nul si le mot est bien parenthésé.

```
bool verification(char * mot) {
    int i = 0;
    bool fin = false;
```

```
    int compteur = 0;
    bool resultat = false;
    while (!fin && (mot[i] != '\0')) {
        if (mot[i]=='(') {
            compteur++;
        }
        else if (mot[i] == ')') {
            compteur = compteur - 1;
        }
        if (compteur<0) {
            fin = true;
        }
        i = i + 1;
    }
    if (compteur == 0) {
        resultat = true;
    }
    return resultat;
}
```

4. On obtient un algorithme linéaire en la taille du mot en entrée.

5. Il y a 4^n mots à générer (puisqu'un mot à n couples de parenthèses possède $2n$ lettres). Pour chacun, la vérification de s'il est bien formé se fait en $O(2n) = O(n)$. À supposer que la construction des mots ne soit pas coûteuse, on obtient une complexité pour l'algorithme naïf en $O(4^n \times n)$.

6. Voici une proposition qui répond aux questions 6, 7 et 8.

```
void dyck(char s[N], int o, int f, int n, uint64_t* r,
          struct liste_s ** liste) {
  if (f == n) {
      if (liste != NULL) {
          struct liste_s * c = malloc(sizeof(struct liste_s));
          strcpy(c->chaine, s);
          c->suivant = *liste;
          *liste = c;
      }
      else {
          printf("%s\n", s);
      }
      (*r) = (*r) + 1;
      return;
  }
  if (o<n) {
      // printf("on ajoute (\n");
      char res[N];
      strcpy(res, s);
      strcat(res, "(");
      dyck(res, o+1, f, n, r, liste);
  }
  if (f<o) {
      // printf("on ajoute )\n");
      char res[N];
```

```
      strcpy(res, s);
      strcat(res, ")");
      dyck(res, o, f+1, n, r, liste);
  }
}
```

7. Une façon de faire est d'ajouter un paramètre passé par pointeur pour compter ce nombre. On pourrait aussi utiliser une variable globale. On trouve 35357670 mots corrects pour $n = 16$.

8. Voir la proposition en question 6.

Exercice 3 *Chemins simples sans issue (exo 2023)*

1. Voici le code demandé :

```
let est_sommet g a =  (0 <=a) && (a < Array.length g)
```

Jury : Le candidat ne doit pas oublier de tester la positivité de l'entier en entrée.

2. Une proposition de code :

```
let rec appartient liste a = match liste with
 | [] -> false
 | b::suite -> (b = a) || (appartient suite a)
```

Jury : Le jury s'attend à une solution récursive. Dans cet exercice une approche impérative n'est néanmoins pas pénalisée.

3. Une proposition de code :

```
let rec est_chemin g liste = match liste with
  | [] -> true
  | [a] -> est_sommet g a
  | a::b::suite -> (appartient (g.(a)) b) && (est_chemin g (b::suite))
```

Jury : Même remarque que pour la question précédente.

4. Voici le code demandé :

```
let est_chemin_simple_sans_issue g liste =
 let n = Array.length g in
 let visites = Array.make n false in
 let rec test_aux liste = match liste with
     | [] -> false
     | [a] -> [] = (List.filter (fun x -> not visites.(x)) (g.(a)))
     | a::b::suite ->
         begin
             visites.(a) <- true ;
             (appartient g.(a) b) && (not visites.(b))
             && (test_aux (b::suite))
         end
     in
     test_aux liste
```

Jury : Plusieurs variantes de réponses à cette question sont possibles.

5. Voici un exemple de solution :

```
let genere_chemins_simples_sans_issue (g:graphe) =
  let taille = Array.length g in
  (*garde en mémoire les chemins déjà trouvés*)
  let liste_chemins = ref [] in
  (*garde en mémoire les sommets en cours de visite *)
  let visites = Array.make taille false in
  (*garde en mémoire le début d'un chemin*)
  let chemin_courant_envers = ref [] in

  (*trouve tous les chemins simples sans issue commençant par s*)
  let rec profondeur s =
   if not visites.(s) then begin
     visites.(s) <- true ;
     chemin_courant_envers := s::(!chemin_courant_envers) ;
     let voisins_libres =
       List.filter (fun x -> not visites.(x)) g.(s)
     in
     if voisins_libres = [] then begin
       liste_chemins := (List.rev !chemin_courant_envers)::(!liste_chemins)
     end else begin
       List.iter profondeur voisins_libres
     end;
     (*pour revenir en arrière *)
     visites.(s) <- false ;
     chemin_courant_envers := List.tl !chemin_courant_envers ;
   end
  in
  for i = 0 to (taille-1) do
   profondeur i
  done ;
  !liste_chemins
```

Jury : Le jury s'attend à ce que les ajouts dans une liste soient faits en tête, quitte à renverser la liste à la fin.

6. Voici un exemple de solution :

```
let liste1 = genere_chemins_simples_sans_issue g1
let liste2 = genere_chemins_simples_sans_issue g2
```

Jury : Le jury souhaite voir le résultat des tests demandés dans cette question : les candidats qui souhaitent compiler leur code doivent donc coder des fonctions d'affichage pertinentes pour obtenir tous les points ici.

7. La complexité de `appartient` est en $O(|l|)$: en effet, on effectue un parcours de liste.

   On note A l'ensemble des arêtes du graphe en argument. La complexité de `est_chemin_sans_issue` est en $O(|l| + |A|)$. En effet, pour chaque sommet s de la liste l, on effectue en terme de calcul de l'ordre de $1 + |g.(s)|$. En sommant sur tous les éléments de l (dans le cas où il serait tous distincts), on trouve une complexité en $O(|l| + |A|)$. Si un même sommet apparaît deux fois, le calcul est interrompu lors de la visite d'une première répétition et on retrouve la même complexité.

   Jury : Le jury attend une borne précise sur ces complexités. Si le candidat propose une borne en $\mathcal{O}(|l||A|)$ pour `est_chemin_simple_sans_issue`, il est invité à l'affiner.

Exercice 4 *Récolte dynamique de fleurs (exo 2023)*

1. On trouve 11 fleurs.

   Jury : On attend une réponse orale ; si elle est correcte, le jury ne demande même pas de justification.

2. La récolte en une case dépend récursivement de celle de la case du haut et de celle de la case de gauche. La formule générique est donc à adapter lorsqu'on se trouve sur le bord gauche ou le bord haut du champ.

```c
int recolte(int champ[m][n], int i, int j){
  /* Cas de base */
  if ( (i == 0) && (j == 0) )
   return champ[0][0];
  if (i == 0)
    return champ[0][j] + recolte(champ,0, j - 1);
  if (j == 0)
    return champ[i][0] + recolte(champ,i - 1, 0);
  /* Cas général */
  return champ[i][j] + max(recolte(champ,i - 1, j),recolte(champ,i, j - 1));
}
```

   Jury : Il est attendu des candidats qu'ils soient attentifs aux cas de base. Ils peuvent factoriser la présentation de leur code et l'établissement de la relation de récurrence.

3. On trouve 6 appels à `recolte`.

   Jury : Plusieurs méthodes sont acceptées pour répondre à cette question ; le candidat peut par exemple compter le nombre d'appels en modifiant la fonction précédente ou à la main en déployant l'arbre d'appels.

4. On calcule d'abord les valeurs des cases sur les bords haut et gauche puis on propage soit en remplissant les lignes de gauche à droite, soit en remplissant les colonnes de haut en bas.

   Jury : Une réponse orale claire peut suffire. Le candidat peut aussi s'aider d'un schéma indiquant les dépendances entre les différents termes à calculer.

5. Il s'agit d'une traduction des questions 2 et 4. Le code ci-dessous corrige aussi la question 7.

```c
int recolte_iterative(int champ[m][n], int i, int j,int fleurs[m][n]){
  int x, y;
  fleurs[0][0] = champ[0][0];
  /* Bord haut */
  for (x = 1; x <= i; x++) {
    fleurs[x][0] = champ[x][0] + fleurs[x - 1][0];
  }
  /* Bord gauche */
  for (y = 1; y <= j; y++) {
    fleurs[0][y] = champ[0][y] + fleurs[0][y - 1];
  }
  /* Autres cases */
  for (y = 1; y <= j; y++) {
    for (x = 1; x <= i; x++) {
      fleurs[x][y] = champ[x][y] + max(fleurs[x - 1][y], fleurs[x][y - 1]);
    }
  }

  deplacements(fleurs,i, j);
  return fleurs[i][j];
}
```

6. On commence par distinguer les cas limites. Puis on appelle récursivement `deplacements` en observant quelle est la case voisine qui avait permi d'obtenir le plus de fleurs.

```c
void deplacements(int fleurs[m][n], int i, int j){
  if (i == 0 && j == 0) {
       printf("Case A, ");
       return;
  }
  if (i == 0) {
       deplacements(fleurs,0, j - 1);
       printf("Aller à droite, ");
       return;
  }
  if (j == 0) {
       deplacements(fleurs,i - 1, 0);
       printf("Descendre, ");
       return;
  }
  if (fleurs[i - 1][j] > fleurs[i][j - 1]) {
     deplacements(fleurs,i - 1, j);
     printf("Descendre, ");
  }
  else {
     deplacements(fleurs,i, j - 1);
     printf("Aller à droite, ");
  }
}
```

Jury : On attend bien l'affichage des déplacements, et pas uniquement leur calcul. La mise en forme de l'affichage (avec sauts de lignes, tabulations, ...) est en revanche libre.

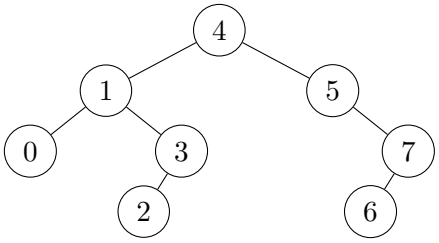
7. Voir le code proposé à la question 5.

   Jury : Le jury est attentif à l'endroit où l'appel à cette fonction est effectué.

Un code source bouquet_corrige.c est aussi disponible.

Exercice 5 *Tri par pile (exo 0)*

1. On obtient l'étiquetage suivant :



   La permutation correspondante est `[4;1;0;3;2;5;7;6]`.

2. Une solution (non linéaire) :

```
let rec parcours_prefixe (a:arbre) :int list = match a with
  |V -> []
  |N(g,x,d) -> x::(parcours_prefixe g)@(parcours_prefixe d)
```

   Une solution linéaire. Le principe est d'utiliser une liste auxiliaire implémentant une pile :

```
let parcours_prefixe_lineaire (a:arbre) :int list =
  let rec parcours_pile (a:arbre) (acc:int list) :int list = match a with
  |V -> acc
  |N(g,x,d) -> let parcours_droit = parcours_pile d acc in
               let parcours_gauche_et_droit = parcours_pile g parcours_droit in
               x::parcours_gauche_et_droit
  in parcours_pile a []
```

   Voir le code compagnon corrigé pour s'en convaincre.

3. La fonction `ecrire_e` fonctionne comme suit : elle étiquette les éléments dans le fils gauche et récupère l'étiquette suivante `next` à mettre : c'est celle qui devra être à la racine. Le fils droit quant à lui doit être étiqueté à partir de `next + 1`.

```
let etiquette (a:arbre) :arbre =
  let rec ecrire_e (a:arbre) (e:int) :arbre*int = match a with
    |V -> V, e
    |N(g,x,d) -> let gauche, next = ecrire_e g e in
                 let droite, fin = ecrire_e d (next +1) in
                 N(gauche,next,droite), fin
  in let res, _ = ecrire_e a 0 in res
```

4. Les opérations sont successivement : empiler $4, 1, 0$; dépiler 0 puis 1 ; empiler $3, 2$; dépiler 2 puis 3 puis 4 ; empiler 5 ; dépiler 5 ; empiler $7, 6$; dépiler 6 puis 7. Cette permutation est triable par pile.

5. On suit l'indication de l'énoncé en se servant d'une fonction `etape`. Remarquons qu'on peut effectivement implémenter une pile par une liste : le sommet de la pile est la tête de la liste.

```
let trier (l:int list) :int list =
  let rec etape (entree:int list) (pile:int list) (sortie:int list) :int list =
    match entree, pile with
    |[], [] -> sortie
    |[], t::q -> etape entree q (t::sortie)
    |t::q, [] -> etape q (t::pile) sortie
    |te::qe, tp::qp when te > tp -> etape entree qp (tp::sortie)
    |te::qe, tp::qp -> etape qe (te::pile) sortie
  in etape l [] []
```

6. Supposons par l'absurde que σ est une permutation triable par pile telle qu'il existe $i < j < k$ tel que $\sigma_k < \sigma_i < \sigma_j$. Alors l'énoncé indique qu'elle sera forcément triée par l'algorithme proposé. Ce dernier traitera d'abord σ_i puis σ_j puis σ_k. Alors :

   - σ_i est empilé en premier.
   - Au moment de traiter σ_j, σ_i est toujours dans la pile sinon il a été dépilé et comme σ_k est traité ultérieurement, donc en particulier dépilé ultérieurement, on aurait dépilé un élément supérieur à σ_k avant σ_k : c'est impossible. Donc σ_j est empilé au dessus de σ_i.
   - Au moment de traiter σ_k, même raisonnement : σ_j est toujours dans la pile sinon σ_j apparaîtrait avant σ_k ce qui contredit $\sigma_k < \sigma_j$. Donc σ_k est empilé au-dessus de σ_j.

   Mais le fonctionnement d'une pile fait alors que nos trois éléments seront dépilés dans l'ordre σ_k, σ_j, σ_i ce qui contredit $\sigma_i < \sigma_j$.

7. (a) On montre par induction sur les arbres que la permutation associée à un arbre binaire est triable par pile. C'est le cas de l'arbre vide. Soit $N(g, x, d)$ un arbre binaire et σ la permutation associée. L'étiquetage de l'arbre indique que $x = |g|$ et la permutation σ est donc la concatéantion de x suivie de la permutation σ_g associée à l'arbre g qui est triable par pile, suivie de la permutation décalée $\sigma_d + x + 1$ correspondant à la permutation associée à d où on a ajouté $x + 1$ à chaque terme ; qui est tout aussi triable par pile que σ_d. De plus, la racine x est supérieure aux éléments à gauche et inférieure au éléments à droite.

   Si on applique l'algorithme du sujet à σ on va donc :

   - Empiler x.
   - Appliquer l'algorithme à σ_g qui est triable par pile sans jamais dépiler x puisqu'il est plus grand que tous les éléments qui sont dans le sous arbre gauche.
   - Considérer le premier élément de d, qui est plus grand qur x, donc dépiler x.
   - Appliquer l'algorithme à σ_d.

   On obtient bien une liste triée à la fin de cette exécution donc σ est triable par pile.

   (b) À rédiger proprement, mais le principe de la récurrence proposée dans l'indication semble reposer sur la contraposée de Q6.

Exercice 6 *Maximum des degrés d'un graphe (exo 0)*

1. Il suffit de récupérer le plus grand degré des sommets présents dans `partie` :

```
int degre_max(graph* g, bool* partie)
{
  int dmax = -1;
  for (int i = 0; i < g->n; i++)
```

```
  {
    if (partie[i] && g->degre[i] > dmax)
    {
      dmax = g->degre[i];
    }
  }
  return dmax;
}
```

On utilise la paresse du `&&` dans la condition : si le sommet `i` ne fait pas partie du sous-ensemble considéré, on ne mettra effectivement pas `dmax` à jour car `partie[i]` sera faux.

2. L'objectif ici est de faire un parcours depuis le sommet `s`. On utilise pour cela une fonction auxiliaire récursive de parcours en profondeur :

```
void parcours(graph* g, int s, bool* vus)
{
  if (!vus[s])
  {
    vus[s] = true;
    for (int i = 0; i < g->degre[s]; i++)
    {
      parcours(g, g->voisins[s][i], vus);
    }
  }
}
```

Il ne reste plus qu'à l'appeler depuis `s` avec un tableau de sommets vus rempli de `false` :

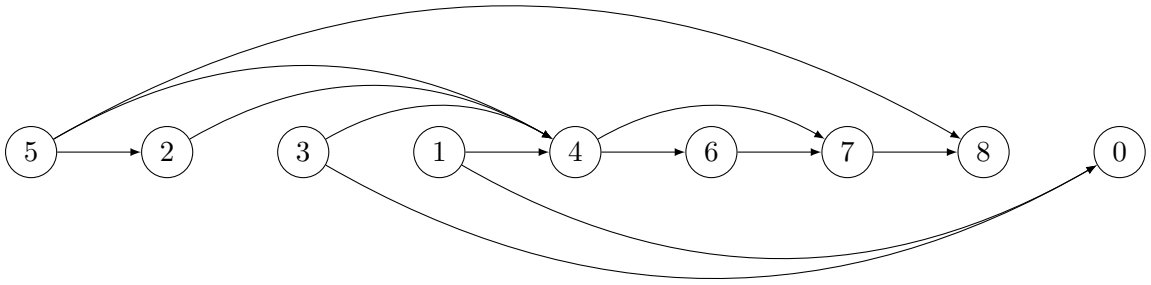
```
bool* accessibles(graph* g, int s)
{
  int nb_sommets = g->n;
  bool* vus = malloc(nb_sommets * sizeof(bool));
  for (int i = 0; i < nb_sommets; i++)
  {
    vus[i] = false;
  }
  parcours(g, s, vus);
  return vus;
}
```

3. On emboîte les deux fonctions précédentes :

```
int degre_etoile(graph* g, int s)
{
  bool* accessibles_s = accessibles(g,s);
  int d_etoile = degre_max(g, accessibles_s);
  free(accessibles_s);
  return d_etoile;
}
```

Le calcul des sommets accessibles se fait en $O(|S| + |A|)$ et `degre_max` est linéaire en $|S|$ d'où une complexité pour `degre_etoile` en $O(|S| + |A|)$.

4. En faisant un parcours en profondeur de manière à toujours parcourir le sommet de plus petit numéro lorsqu'on a le choix, on obtient la linéarisation suivante (d'autres sont possibles) :



5. Notons tout d'abord que notre réponse à la question 3 ne convient pas : on obtiendrait une complexité en $O(|S|(|S|+|A|))$ qui n'est pas suffisante. On remarque tout d'abord que :

$$d^*(s) = \begin{cases} 0 \text{ si } s \text{ est de degré sortant nul} \\ \max(d^+(s), \max\{d^*(u) \mid u \text{ voisin de } s\}) \text{ sinon} \end{cases}$$

Dans un graphe acyclique, cela signifie qu'on peut calculer les $d^*(s)$ par programmation dynamique :

(1) On calcule un ordre topologique des sommets du graphe.

(2) On calcule les $d^*(s)$ en considérant les sommets dans l'ordre topologique inverse (donc, en commençant par les sommets de droite avec les conventions de Q4). Cet ordre garantit qu'on aura calculé tous les $d^*(s')$ pour s' voisin de s au moment de calculer $d^*(s)$ donc que le calcul de $d^*(s)$ consitera à calculer un maximum parmi $d^+(s)+1$ valeurs.

La complexité de l'étape (1) est en $O(|S|+|A|)$ puisqu'elle revient à faire un parcours en profondeur. Dans l'étape (2), chaque sommet s induit un coût en $O(d^+(s))$ d'où un coût total pour traiter tous les sommets en $O(|A|)$. On atteint donc bien la complexité attendue.

6. Dans une composante fortement connexe d'un graphe, tous les sommets ont le même d^* puisque les mêmes sommets sont accessibles. On calcule donc le graphe des composantes fortement connexes (ce qui se fait linéairement en la taille du graphe en entrée avec Kosaraju), on applique la méthode précédente (qui sera linéaire en la taille du graphe des CFC donc linéaire en la taille du graphe initial) et on fixe le d^* de chaque sommet comme étant celui de sa CFC.

Exercice 7 *Sac à dos (exo 0)*

1. Il suffit de stocker les poids dans un tableau d'entiers de taille n, idem pour les valeurs. Les indicateurs x_i peuvent être stockés dans un tableau de booléens de taille n.

2. Voici une proposition :

```
bool* KP_glouton(int n, int pmax, int* poids)
{
    bool* indicateurs = malloc(n * sizeof(bool));
    for (int i = 0; i < n; i++)
    {
        indicateurs[i] = false;
    }

    int poids_courant = 0;
    for (int i = 0; i < n; i++)
    {
```

```
        int nouveau_poids = poids_courant + poids[i];
        if (nouveau_poids <= pmax)
        {
            indicateurs[i] = true;
            poids_courant = nouveau_poids;
        }
    }
    return indicateurs;
}
```

3. On rappelle que `scanf` prend une chaîne formatée et des adresses vers des objets de type cohérent avec celui des formats. Compléter le `main` par le code suivant répond à la question, mais il serait utile de faire des affichages indiquant successivement ce qu'il faut entrer :

```
int n;
scanf("%d\n", &n);
int pmax;
scanf("%d\n", &pmax);
int* v = malloc(n * sizeof(int));
for (int i = 0; i < n; i++)
{
    scanf("%d", &v[i]);
}
int* p = malloc(n * sizeof(int));
for (int i = 0; i < n; i++)
{
    scanf("%d", &p[i]);
}

bool* x = KP_glouton(n, pmax, p);
afficher_tableau(n, x);
printf("valeur du sac = %d, poids utilisé = %d\n", valeur(x,v,n), valeur(x,p,n));
free(x);
free(p);
free(v);
```

La fonction `void afficher_tableau(intn bool* tab)` affiche le contenu d'un tableau de booléens et la fonction `int valeur(bool* x, int* v, int n)` permet de faire le calcul de $\sum_{i=0}^{n-1} x_i v_i$ étant donnés les tableaux `x` et `v`.

4. Non à toutes les questions, construisons des contre-exemples :

   (a) Avec

numéro	0	1
poids	10	10
valeur	1	42

   et $p_{max} = 10$, l'algorithme glouton fournit un sac de valeur 1 (seul l'objet 0 est pris) alors qu'un sac de valeur 42 est possible en prenant uniquement l'objet 1.

   (b) Avec

numéro	0	1	2
poids	10	5	5
valeur	10	9	9

   et $p_{max} = 10$, on obtient un sac de valeur 10 (seul l'objet 0 est pris) alors qu'un sac de valeur 18 est possible en prenant les deux derniers objets.

(c) Avec

numéro	0	1
poids	1	10
valeur	1	10

et $p_{max} = 10$, on obtient un sac de valeur 1 (seul l'objet 0 est pris) alors qu'un sac de valeur 10 est possible en prenant uniquement l'objet 1.

(d) Avec

numéro	0	1
poids	1	10
valeur	5	10
v/p	5	1

et $p_{max} = 10$, on obtient un sac de valeur 5 (seul l'objet 0 est pris) alors qu'un sac de valeur 10 est possible en prenant uniquement l'objet 1.

5. Le problème de décision associé est :

$$\begin{cases} \textbf{Entrée :} \ n, p_{max}, v_{seuil} \in \mathbb{N}; (v_0, v_1, ..., v_{n-1}) \text{ et } (p_0, ..., p_{n-1}) \in \mathbb{N}^n. \\ \textbf{Question :} \text{ Existe-t-il } (x_0, ..., x_{n-1}) \in \{0,1\}^n \text{ tel que } \sum_{i=0}^{n-1} x_i p_i \leq p_{max} \text{ et } \sum_{i=0}^{n-1} x_i v_i \geq v_{seuil} \, ? \end{cases}$$

Ce problème est dans NP car pour toute instance positive, le tuple $(x_0, ..., x_{n-1})$ fournit un certificat de taille polynomiale en la taille de l'entrée et calculer les deux sommes puis les comparer à p_{max} et v_{seuil} respectivement se fait en temps polynomial.

6. Il faudrait considérer les 2^n choix pour $(x_0, x_1, ...x_{n-1})$ possibles. Pour chacun, il faudrait calculer le poids occupé dans le sac et sa valeur, ce qui se fait en $O(n)$, et conserver le tuple réalisant la meilleure valeur qui rentre dans le sac : la complexité de cette recherche exhaustive est en $O(n \times 2^n)$.

Pour améliorer cet algorithme en pratique (mais pas dans le pire cas), on peut mettre en place une stratégie par retour sur trace dans laquelle on élague les branches dès que le poids maximal est dépassé.

Exercice 8 *Tableaux autoréférents (exo 0)*

1. Pas de difficulté :

```
let somme (t:int array) (i:int) :int =
  let s = ref 0 in
  for k = 0 to i do
    s := !s + t.(k)
  done;
  !s
```

2. Pour $n = 1$, le seul tableau à valeurs dans $[\![0, n-1]\!]$ est `[|0|]` qui n'est pas autoréférent. Pour $n = 2$, les tableaux possibles sont `[|0;0|]`, `[|0;1|]`, `[|1;0|]` et `[|1;1|]` et aucun ne convient. De même, pour $n = 3$ aucun tableau ne convient :

   - Si $t.(0) = 0$ alors le tableau contient au moins un zéro donc n'est pas autoréférent.
   - Si $t.(0) = 1$ alors soit $t.(1) = 0$ mais dans ce cas $t.(2)$ vaut forcément 2 et t n'est pas autoréférent, soit $t.(2) = 0$ mais dans ce cas $t.(1)$ vaut forcément 1 et il y a deux 1 dans t : contradiction.
   - Si $t.(0) = 2$ alors nécessairement t vaut `[|2;0;0|]` qui n'est pas autoréférent.

   On constate que pour $n = 4$, le tableau `[|2;0;2;0|]` est autoréférent.

3. L'idée est de calculer un tableau `occ` contenant en case i le nombre d'occurrences de la valeur i dans le tableau `t` en entrée : cela peut se faire linéairement en la taille de `t`. Le tableau `t` est autoréférent si et seulement si il est égal au tableau `occ` et cette égalité se teste aussi en $O(|t|)$.

   Ci-dessous, on anticipe la question 5 en proposant une fonction qui calcule un tableau contenant en case i le nombre d'éléments valant i dans le morceau du tableau `t` compris entre les indices 0 et `fin` : il suffira de l'appeler avec `fin` = la taille du tableau `t` pour obtenir `est_auto`.

```
let occurrences (t:int array) (fin:int) :int array =
  let n = Array.length t in
  let occ = Array.make n 0 in
  for i = 0 to fin do
    occ.(t.(i)) <- occ.(t.(i)) +1
  done;
  occ

let est_auto (t:int array) :bool =
  t = occurrences t (Array.length t -1)
```

On indique ci-dessous les lignes qui vont être ajoutées lors des questions 4, 5, 6 entre les deux commentaires indiquant le début et la fin de l'élagage dans le code compagnon initial :

```
let s = somme t i in
if s > n then raise Echec;
let occ = occurrences t i in
for k = 0 to i do
  if occ.(k) > t.(k) then raise Echec
done;
if s + (n-i-1) -(t.(0) - occ.(0)) > n then raise Echec;
```

Les explications suivent.

4. La somme des éléments d'un tableau t autoréférent de taille n est nécessairement n puisque dans un tableau autoréférent on a par définition :

$$\sum_{i=0}^{n-1} t.(i) = \sum_{i=0}^{n-1} (\text{nombre de cases valant } i) = n$$

car les cases contiennent toutes un élément de $[\![0, n-1]\!]$ donc la deuxième somme revient à compter le nombre de cases. *Remarque : pas besoin de justifier cette affirmation pour poursuivre : on voit très bien que c'est le cas en observant la somme des tableaux obtenus avec* `gen_auto` *sur de petits* n*.*

On en déduit que, si après avoir rempli la case i du tableau en cours de construction, la somme jusqu'à la case i du tableau dépasse n, on peut élaguer, d'où les lignes 1 et 2.

5. Pour que le tableau soit autoréférent, il faut au moins qu'il soit autoréférent "entre les indices 0 et i", c'est-à-dire qu'il y ait $t.(k)$ occurrences de k dans le tableau. Si après avoir rempli $t.(i)$ il y a un $0 \leq k \leq i$ pour lequel le nombre d'occurrences de k est plus grand que $t.(k)$ strictement, on sait que le tableau ne sera pas autoréférent car on ne peut qu'ajouter des occurrences de k dans le tableau en le remplissant. On en déduit les lignes 3 à 6. Le calcul de `gen_auto 15` devient instantané.

6. En notant n la taille du tableau, après avoir rempli la case i, il en reste $n-1-i$ à remplir. On sait par ailleurs qu'il y aura $t.(0)$ éléments valant zéro dans le tableau si on veut avoir une chance qu'il soit autoréférent donc qu'il faut encore en placer ($t.(0)$ - (le nombre de 0 qui ont déjà été placés)). Dans les autres cases, il y aura une valeur valant au moins 1, donc on sait que la somme du morceau de tableau au delà (strict) de i vaut au moins :

$$(n - i - 1) - (t.(0) - \text{nombre de 0 placés entre les indices 0 et i})$$

Or on sait que la somme totale du tableau ne doit pas dépasser n donc si en sommant cette quantité à la somme du tableau entre les indices 0 et i on dépsse n le tableau n'est pas autoréférent. Cela donne lieu à la ligne 7 et le calcul pour $n = 30$ devient instantané.

On calcule une bonne fois pour toutes `somme t i` *et* `occurrences t i` *plutôt que de les calculer une fois pour Q4 et une fois pour Q6 (respectivement une fois pour Q5 et une fois pour Q6) : cela limite les calculs.*

7. Soit $n \geq 7$. Montrons que le tableau suivant est autoréférent de taille n (comme $n \geq 7$, la case d'indice $n-4$ porte bien un indice plus grand que 3, avec égalité pour $n = 7$, auquel cas la première plage de zéros est simplement vide) :

Indice	0	1	2	3	...	$n-5$	$n-4$	$n-3$	...	$n-1$
Valeur	$n-4$	2	1	0	...	0	1	0	...	0

Il y a bien une occurrence de $n-4$ dans ce tableau, 2 occurrences de 1, une occurrence de 2 et toutes les autres cases (donc $n-4$) contiennent des zéros : c'est gagné.

Exercice 9 *Calcul avec les flottants (exo 2023)*

1. Il suffit d'ajouter dans le `main` :

```
printf("(a+b)+c = %f\n", (a + b) + c);
printf("a+(b+c) = %f\n", a + (b + c));
```

Le résultat attendu est 1 : on l'obtient avec la première ligne mais pas la seconde. Ceci est du à un arrondi : $|b|$ est très grande devant $|c|$ si bien que lorsqu'on calcule $b+c$, il y des chances que le résultat soit représenté de la même façon que b. Or b et a sont opposés donc les sommer donne 0.

Jury : Le jury attend une brève explication du comportement constaté.

2. On teste si $1 + 2^{-n}$ est égal (on ne teste jamais l'égalité de flottants, mais on va faire une exception ici) à 1 jusqu'à ce que ce soit le cas. Attention, quand on sort de la boucle, c'est parce qu'on a trouvé n tel que $1 + 2^{-n} = 1$, le ε machine est donc le n juste avant :

```c
int epsilon()
{
    int n = 1;
    while (1 + pow(2, -n) != 1)
    {
        n++;
    }
    return (n - 1);
}
```

On trouve 52 ce qui est a priori normal : la mantisse est de taille 52 donc le plus petit nombre qu'on peut représenter après 1 consiste à mettre le bit le plus à droite de la mantisse à 1 et donc à ajouter à 1 la quantité 2^{-52}.

Jury : Le jury attend une explication faisant le lien entre la valeur trouvée et le format de représentation des flottants rappelé dans l'énoncé.

3. Une version récursive naïve (on pourrait utiliser le même principe de mémoïsation que pour le calcul des termes de la suite de Fibonacci pour gagner en complexité temporelle) :

```c
double u(int n)
{
    if (n == 0) {return 2; }
    if (n == 1) {return -4; }
    return 111 - 1130 / u(n - 1) + 3000 / (u(n - 1) * u(n - 2));
}
```

Jury : Une fonction récursive naïve suffit à obtenir tous les points à cette question mais le candidat est bien entendu libre de stocker les valeurs de la suite demandée pour éviter d'avoir à les recalculer.

4. On ajoute une petite boucle dans le `main` :

```c
for (int i = 0; i < 22; i++)
{
    printf("u_%d = %lf\n", i, u(i));
}
```

Avec seulement 22 termes, difficile à prévoir, mais il semblerait que la suite converge vers 100.

Jury : La question ne demande pas de justifier la limite théorique. Le jury doit voir apparaître les premiers termes correctement et clairement affichés.

5. Pas de difficulté :

```c
double somme(struct nb *liste)
{
    double s = 0.;
    while (liste != NULL)
    {
        s = s + liste->x;
        liste = liste->suivant;
    }
```

```
    return s;
}
```

6. Le code compagnon est pénible à manipuler, je propose une version de l'algorithme demandé qui s'efforce de conserver le plus du code d'origine. Une conséquence de ces modifications est que la question sur le cas d'insertion non pris en compte est caduque.

```
double somme2(struct nb *tab)
{
    struct nb *temp = tab;
    while (temp != NULL)
    {
        if (temp->suivant != NULL)
        {
            // Création du maillon s contenant la somme des 2 premiers éléments
            struct nb *s = malloc(sizeof(struct nb));
            double u = temp->x;
            double v = temp->suivant->x;
            double w = u + v;
            s->x = w;
            s-suivant = NULL;
            // temp pointe vers le 3e élément de la liste
            struct nb *t = (temp->suivant)->suivant;
            free(temp->suivant);
            free(temp);
            temp = t;

            // recherche de l'emplacement de s
            if (t == NULL)
            {
                free(s);
                return w;
            }
            else
            {

                while (t->suivant != NULL && t->suivant->x < w)
                {
                    t = t->suivant;
                }
                // insertion de s
                if (t->suivant == NULL)
                {
                    t->suivant = s;
                    s->suivant = NULL;
                }
                else
                {
                    s->suivant = t->suivant;
                    t->suivant = s;
                }
            }
```

```
46          }
47          else
48          {
49              return temp->x;
50          }
51      }
52      return temp->x; // n'arrive pas
53  }
```

Quelques remarques :

- On n'atteint jamais la ligne 53 puisqu'on suppose que la liste en entrée n'est pas vide (et heureusement car à cet endroit `temp` vaut `NULL` donc n'a pas de champ `x`).
- Aux lignes 18 et 19, on libère les maillons dont on vient de faire la somme pour éviter les fuites mémoire. Au passage, cela détruit la liste en entrée mais c'est inévitable vu l'algorithme qui demande de toutes façons de la modifier au fur et à mesure.
- Pour trouver l'emplacement du nouveau maillon, on commence par vérifier qu'il reste des maillons dans la liste en ligne 23 ; si non, on peut directement renvoyer la valeur attendue.
- Puis, on parcourt la liste à horizon d'un maillon de sorte à ce que lorsqu'on sorte de la boucle `while` en ligne 31, on doive placer le maillon `s` juste derrière `t` ; ce qu'on fait entre les lignes 36 et 45 en distingant le cas selon qu'il faille juste brancher `c` en fin de la liste ou non.

On ne voit pas grande différence sur le test proposé par le code compagnon (qui demande de sommer les $1/i$ pour i allant de 1 à 10^5), si ce n'est que `somme2` est bien plus lent : c'est normal, cette fonction est quadratique en le nombre d'éléments à sommer quant `somme` n'est que linéaire.

Jury : À cette question, un candidat qui aurait fait le choix d'implémenter `somme2` directment et sans utiliser le code déjà fourni et en expliquant sa démarche aurait eu tous les points.

Exercice 10 *Langages locaux (exo 2023)*

1. On obtient $P(L) = \{a\}$, $D(L) = \{a, b\}$, $F(L) = \{aa, ab, ba\}$ et donc $N(L) = \{bb\}$. Le langage L n'est pas local : s'il l'était, il devrait contenir le mot aba ce qui n'est pas.

   Jury : On attend du candidat qu'il donne les ensembles demandés sans justification, possiblement à l'oral uniquement puis qu'il exhibe un mot montrant la non localité avec une brève justification.

2. La seule difficulté est de justifier la disjonction de cas pour les concaténations. Si $e = e_1e_2$ et $\varepsilon \notin L(e_1)$, une première lettre d'un mot de e est nécessairement une première lettre d'un mot de e_1 (la réciproque étant évidente). Sinon, un mot de $L(e)$ peut aussi commencer de la même façon qu'un mot de $L(e_2)$.

   Jury : Le jury attend principalement deux arguments : le fait que l'on raisonne par induction et la justification de la disjonction de cas dans le cas d'une concaténation.

3. On filtre sans grande difficulté.

```
let rec contains_epsilon (e:regexp) :bool = match e with
|Epsilon -> true
|Letter _ -> false
|Union (e1, e2) -> (contains_epsilon e1) || (contains_epsilon e2)
|Concat (e1, e2) -> (contains_epsilon e1) && (contains_epsilon e2)
|Star _ -> true
```

4. Comme pour `compute_P`, la seule difficulté est le cas d'une concaténation.

```
let rec compute_D (e:regexp) :string list  = match e with
|Epsilon -> []
|Letter a -> [a]
|Union (e1, e2) -> union (compute_D e1) (compute_D e2)
|Concat (e1, e2) when contains_epsilon e2 -> union (compute_D e1) (compute_D e2)
|Star e1 |Concat (_, e1) -> (compute_D e1)
```

5. Soit e une expression régulière et L le langage qu'elle dénote. On calcule $F(L)$ inductivement :
   - Si $e = \varepsilon$ ou $e = a$ avec $a \in \Sigma$, $F(L) = \emptyset$.
   - Si $e = e_1 + e_2$, $F(L) = F(L_1) \cup F(L_2)$ où L_i est le langage dénoté par e_i.
   - Si $e = e_1 e_2$, un facteur de taille 2 d'un mot de L est : soit un facteur de taille 2 de L_1, soit un facteur de taille 2 de L_2, soit un facteur "à cheval entre L_1 et L_2" c'est-à-dire dont la première lettre est la fin d'un mot de L_1 et la deuxième est le début d'un mot de L_2. Autrement dit :
   $$F(L) = F(L_1) \cup F(L_2) \cup D(L_1)P(L_2)$$
   - Si $e = e_1^\star$, on obtient similairement au cas précédent $F(L) = F(L_1) \cup D(L_1)P(L_1)$.

   Jury : Le jury attend un argument inductif et que le candidat précise les cas délicats (étoile et concaténation). Cette question peut être abordée en même temps que la question 7.

6. On peut remplacer le `List.map` par une petite fonction auxiliaire au besoin.

```
let rec prod (l1:string list) (l2:string list) = match l1, l2 with
|[], _ |_, [] -> []
|t::q, l -> union (List.map (fun x -> t^x) l) (prod q l)
```

   Jury : Le jury souhaite une fonction récursive et la demande si cette approche n'est pas proposée. Le candidat est libre de proposer une solution faisant intervenir les fonctions du module `List` ou de s'aider d'une fonction auxiliaire.

7. C'est une traduction de la question 5.

```
let rec compute_F (e:regexp) :string list = match e with
|Epsilon |Letter _ -> []
|Union (e1, e2) -> union (compute_F e1) (compute_F e2)
|Concat (e1, e2) ->
   let d_e1 = compute_D e1 and p_e2 = compute_P e2 in
   union (union (compute_F e1) (compute_F e2)) (prod d_e1 p_e2)
|Star e1 ->
   let d_e1 = compute_D e1 and p_e1 = compute_P e1 in
   union (compute_F e1) (prod d_e1 p_e1)
```

8. On remarque qu'une fois calculé $F(L)$ on en déduit immédiatement $N(L)$ en calculant tous les facteurs de taille 2 de Σ et en éliminant ceux qui sont dans $F(L)$.

   Il est ensuite très facile de concevoir un automate pour chacun des langages $P(L)\Sigma^\star$, $\Sigma^\star D(L)$ et $\Sigma^\star N(L)\Sigma^\star$; les ensembles $P(L)$, $D(L)$ et $N(L)$ étant finis. De plus, on sait algorithmiquement construire l'intersection (automate produit) et le complémentaire (déterminisation puis échange des états finaux et non finaux) d'automates donc on peut construire un automate reconnaissant
   $$P(L)\Sigma^\star \cap \Sigma^\star D(L) \cap (\Sigma^\star N(L)\Sigma^\star)^c = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$$

   Jury : Une description haut niveau de la construction de cet automate (exploitant par exemple les stabilités des langages rationnels) suffit.

9. On propose la méthode suivante :

   - Calculer un automate A reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ via la question 8.
   - Calculer un automate B reconnaissant $L \setminus \{\varepsilon\}$ à partir de l'expression régulière dénotant L, par exemple à l'aide de l'algorithme de Berry-Sethi.
   - Calculer l'automate $A\Delta B$ reconnaissant la différence symétrique de ces deux langages (la différence symétrique se construisant à partir d'unions, intersections et complémentaires, opérations qu'on sait faire sur des automates finis).
   - A l'aide d'un parcours, déterminer si un état final de $A\Delta B$ est accessible depuis un de ses états initiaux. Si non, le langage reconnu par $A\Delta B$ est vide donc les langages reconnus par A et B sont égaux donc L est local. Si oui, le langage reconnu par $A\Delta B$ n'est pas vide donc les langages reconnus par A et B son différents et L n'est pas local.

   Jury : Le jury s'attend à ce que l'égalité de ces deux langages soit vérifiée par le biais d'automates les reconnaissant, à nouveau par une description de haut niveau.

10. Si e est un expression régulière telle que $L(e) \neq \emptyset$, il existe une expression régulière e' ne contenant pas le symbole $\emptyset$ telle que $L(e') = L(e)$. Cela se montre aisément par induction sur e.

    Si on autorise les expressions régulières contenant $\emptyset$, et qu'on souhaite à partir de e déterminer si $L(e)$ est local, on pourrait donc :

    - Déterminer si $L(e)$ est vide à l'aide d'un automate reconnaissant ce langage.
    - S'il est vide, il est local. Sinon, on peut inductivement transformer e en e' telle que $L(e) = L(e')$ et e' ne contient pas le symbole $\emptyset$ et on applique l'algorithme de la question 9 à e'.

    Jury : Là encore une justification haut niveau du fait qu'on peut se ramener à une expression régulière ne contenant pas le symbole vide suffit (on ne demande par exemple pas une preuve par induction de ce fait, simplement le fait qu'on pourrait le faire).

Un code source localite_corrige.ml est aussi disponible.

Exercice 11 *Nombre d'occurrences (exo 0)*

1. On parcourt le tableau en accumulant le nombre de `x` trouvés pour une complexité en $O(n)$.

```
int nb_occurrences(int n, int* tab, int x)
{
  int nb_occ = 0;
  for (int i = 0; i < n; i++)
  {
    if (tab[i] == x) {nb_occ = nb_occ +1;}
  }
  return nb_occ;
}
```

2. Les variables `d` (début) et `f` (fin) donnent la tranche du tableau dans laquelle on effectue la recherche. Si la valeur au milieu vaut `x` alors on a trouvé une occurrence et on s'arrête. Si elle est plus petite que `x`, comme le tableau est trié, il faut chercher `x` à sa droite donc augmenter `d`. Si elle est plus grande, il faut chercher `x` à sa gauche donc diminuer `f`.

```
int une_occurrence(int n, int* tab, int x)
{
  int d = 0;
```

```
  int f = n - 1;
  int res = -1;
  while (d <= f)
  {
    int m = (d + f) / 2;
    if (x == tab[m]) {res = m; break;}
    else if (tab[m] < x) {d = m+1;}
    else {f = m-1;}
  }
  return res;
}
```

3. On reprend l'idée de `une_occurrence` mais sans arrêter la recherche lorsqu'on trouve `x` : dans ce cas, on continue de le chercher à gauche :

```
int premiere_occurrence(int n, int* tab, int x)
{
  int d = 0;
  int f = n - 1;
  int res = -1;
  while (d <= f)
  {
    int m = (d + f) / 2;
    if (tab[m] == x) {res = m; f = m - 1;}
    else if (tab[m] < x) {d = m + 1;}
    else {f = m - 1;}
  }
  return res;
}
```

4. On écrit de manière symétrique une fonction `derniere_occurrence` donnant l'indice de la dernière occurrence de `x` dans le tableau et ainsi on obtient :

```
int nombre_occurrences(int n, int* tab, int x)
{
  int debut = premiere_occurrence(n, tab, x);
  int fin = derniere_occurrence(n, tab, x);
  return fin-debut +1;
}
```

5. La quantité $f - d + 1$ (qui correspond à la taille de la tranche de tableau examinée) est un variant de boucle. Un invariant possible est le suivant : si x est dans le tableau, alors `tab[res]` vaut x ou x est dans le sous-tableau compris entre les indices `d` et `f` inclus.

   Preuves faites en sup', à retravailler.

6. La quantité $f - d + 1$ est divisée par deux à chaque itération (disjonction de cas) donc on effectue $O(\log_2 n)$ itérations de la boucle qui sont chacune en temps constant.

Exercice 12 *SAT (exo 0)*

1. On évalue chacun des littéraux ; la clause est satisfaite si l'un s'évalue en `true` :

```
let evalue (c : clause) (v : valuation) : bool =
  let evalue_litteral (l:litteral) :bool = match l with
    |V i -> v.(i)
    |NV i -> not v.(i)
  in
  let evaluation_litteraux = Array.map (fun l -> evalue_litteral l) c in
  Array.mem true evaluation_litteraux
```

2. Si le programme renvoie `None`, la formule peut tout de même être satisfiable : on a pu échouer à trouver une valuation qui satisfait f. En revanche, si la fonction ne renvoie pas `None`, il est certain que f est satisfiable : on est en présence d'un algorithme Monte-Carlo à erreur unilatérale.

3. On teste sur les entrées suivantes :

```
let _ = random_sat [] 3 10
let _ = random_sat [[||]] 3 10
let _ = random_sat [[|V 0; NV 0|]] 3 10
let _ = random_sat [[|V 0; V 1|]; [|V 2; NV 0; V 1|]] 3 10
let _ = random_sat [[|V 0; V 1|]; [|NV 1|]; [|NV 0|]] 3 10
```

   On teste ainsi, dans cet ordre, le fonctionnement de `random_sat` sur :

   (1) Une FNC vide, qui est une tautologie (cas de base).

   (2) Une FNC contenant une clause vide, qui est une antilogie (cas de base).

   (3) Une tautologie non triviale (hum).

   (4) Une formule satisfiable sans être une tautologie.

   (5) Une antilogie non triviale.

4. On constate que `random_sat` échoue aux tests (2) — on obtient une erreur — et (5) — la fonction s'exécute mais indique qu'une antilogie est satisfiable. Voici une version corrigée :

```
let random_sat (f : formule) (n : int) (k : int) : valuation option =
  assert (k >= 1);
  let v = initialise n in
  let rec test (g : formule)  (k : int) : valuation option =
    match g with
    | [] -> Some v
    | c :: cs ->
       if evalue c v then
         test cs k
       else if k < 1 || Array.length c = 0 then
         None
       else begin
         let i = Random.int (Array.length c) in
         v.(indice c.(i)) <- not v.(indice c.(i));
         test f (k - 1)
       end
  in
  test f k
```

Les modifications apportées sont :

- En ligne 10 : dans une clause vide, on ne peut pas modifier la valeur d'un littéral puisqu'il n'y en a pas ; et si une formule contient une clause vide, elle ne peut pas être satisfaite.
- En ligne 15 : lorsqu'on modifie la valuation, il faut vérifier qu'elle satisfait toutes les clauses de la formule initiale `f` et pas seulement celles de la fin de la formule, `g`.

5. On suit le principe donné par le sujet :

```
let maxsat (f:formule) (n:int) (k:int) :int* valuation =
  let meilleure_val = ref (initialise n) in
  let meilleur_nb = ref (nb_sat f !meilleure_val) in
  for k = 0 to n-2 do

    let nouvelle_val = initialise n in
    let nouveau_nb = nb_sat f nouvelle_val in
    if nouveau_nb > !meilleur_nb then begin
        meilleure_val := nouvelle_val;
        meilleur_nb := nouveau_nb
      end
  done;
  !meilleur_nb, !meilleure_val
```

Une initialisation se fait en $O(n)$. Un appel à `nb_sat` se fait en $O(|f|)$ puisque `evalue` est linéaire en la taille de la clause en entrée. Si on suppose qu'on n'appelle cette fonction qu'avec n étant le vrai nombre de variables dans la formule (quitte à les renommer), on en déduit qu'une itération coûte $O(|f|)$ donc que la complexité de cet algorithme est en $O(k|f|)$.

6. Si c'était le cas, on pourrait résoudre CNF-SAT en temps polynomial (en calculant une valuation via MAX-SAT et en vérifiant que le nombre de clauses qu'elle satisfait est égal au nombre de clauses de la formule). Mais CNF-SAT est NP-complet et cela impliquerait que P = NP, d'où contradiction.

Exercices de type A

Exercice 13 *Une relation d'équivalence (exo 2024)*

1. La réflexivité correspond au cas $k = 0$, la symétrie consiste à réindexer à l'envers, la transitivité consiste à mettre bout à bout les deux séquences.

2. On trouve comme classes : $C_1 = \{1, 5\}, C_2 = \{2, 3, 6\}, C_3 = \{4, 7, 8, 9\}$.

3. Il existe alors un chemin commençant par x et terminant par y. Sachant que le graphe est en réalité symétrique, calculer les classes d'équivalence revient alors à calculer les composantes connexes du graphe correspondant (on peut aussi calculer les composantes fortement connexes si on n'a pas remarqué la symétrie).

4. Dans le cas où on remarque la symétrie : un parcours du graphe suffit (n'importe lequel).

```
   Entrées : graphe G
   Sorties : tableau numComp avec numComp[i] égal au numéro d'une composante connexe
 1 numComp = [0, 0, ..., 0] (indexation de 1 à n inclus)
 2 parcours_largeur(i,num)
 3 file = [i]
 4 tant que file non vide
 5 |  a = defiler(file)
 6 |  si numComp[a] vaut 0 alors
 7 |  |  numComp[a] := num
 8 |  |  pour  x voisin de a
 9 |  |  |  enfiler(x)
10 num = 0
11 pour i = 1 à n
12 |  si numComp[i] = 0 alors
13 |  |  num := num + 1
14 |  |  parcours_largeur (i,num)
15 retourner numComp
```

   Dans le cas où la symétrie n'est pas remarquée, on peut utiliser un algorithme de calcul des composantes fortement connexes, comme l'algorithme de Kosaraju.

5. On reconnaît un graphe biparti où chaque sommet est de degré 1 (c'est le graphe induit par un couplage parfait).

6. On obtient deux types de composantes connexes :

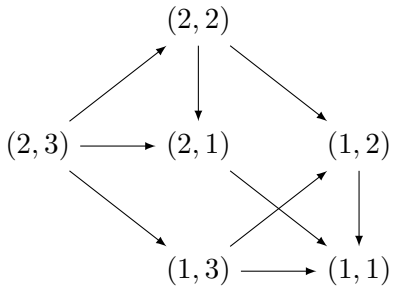
   - composantes à deux sommets (même image par φ et ψ)
   - des cycles ayant un nombre pair de sommets (on peut remarquer qu'il y a alternance entre arêtes créees par φ et créees par ψ).

   On pourra remarquer qu'un sommet est soit d'arité 1, soit d'arité 2. Lorsqu'il est d'arité 1, cela signifie que son voisin et lui ont même image par φ et ψ. Donc ces deux sommets constituent une composante connexe.

   Dans le cas où un sommet est d'arité 2, par la contraposition de la remarque précédente, tous les sommets dans sa composante connexe sont d'arité 2. Ainsi, on a uniquement des cycles ayant un nombre pair de sommets.

Exercice 14 *Jeu de Chomp (exo 2024)*

1. On obtient le graphe suivant (plus lisible avec des tablettes) :



2. Pour le premier graphe, le noyau est (s_2, s_4, s_5). Pour le second il y a deux noyaux (s_1, s_3) et (s_2, s_4).

3. Soit G un graphe acyclique sans puits. Soit $(s_0 \cdots s_k)$ un chemin dans G. Comme s_k n'est pas un puits, il existe un arc (x_k, y) qui prolonge le chemin précédent. Il y a donc un chemin de longueur arbitraire dans G. Par le principe des tiroirs, tout chemin assez long contient nécessairement deux sommets qui coïncident, on a donc un cycle, en contradiction avec G acyclique. Donc il existe au moins un puits dans G.

4. Supprimer des sommets d'un graphe acyclique le laisse acyclique donc la question 3 assure qu'on trouvera toujours un sommet s convenable en ligne 3. Par conséquent, le nombre de sommets dans G décroît strictement à chaque itération ce qui assure la terminaison de l'algorithme.

   Par ailleurs, un sommet sans successeur doit être dans un noyau à cause de la deuxième propriété et donc tous ses prédécesseurs doivent être en dehors à cause de la première. Ces deux contraintes nécessaires sont garanties par l'algorithme. Elles sont suffisantes dans le cadre d'un graphe acyclique car tout sommet qui devrait nécessairement être dans un noyau et nécessairement ne pas y être ferait partie d'un cycle.

5. Travaillons par récurrence forte sur le nombre n de sommets de G. Si $n = 1$ ce seul sommet (puits) constitue le noyau.

   Sinon, soit s un puits de G (qui existe d'après la question 3). Ce sommet fait nécessairement partie de tous les noyaux de G. On note $P(s)$ l'ensemble des prédécesseurs de s. Le graphe G privé de s et des sommets de $P(s)$ reste acyclique et donc, par hypothèse de récurrence, a un noyau unique N. L'ensemble $N \bigcup \{s\}$ est un noyau de G et par unicité de N et nécessité du fait que s soit dans tout noyau, c'est le seul.

6. Les sommets (2,2) et (1,1) constituent le noyau. La position (1,1) est perdante pour le joueur qui doit jouer (il mange le chocolat empoisonné), les éléments du noyau sont les sommets perdants et les autres sommets les positions gagnantes.

7. On montre qu'un joueur qui peut choisir s_1 dans le noyau ne peut pas perdre. Si s_1 n'a pas de successeur, l'adversaire ne peut plus jouer, il a perdu. Sinon, l'adversaire va choisir s_2 dans les successeurs de s_1. On a donc $s_2 \in S \setminus N$ donc s_2 admet au moins un successeur dans N.

8. $\mathcal{A}_i$ est l'ensemble des sommets de S à partir desquels J_1 peut forcer la partie à arriver en F_1 en moins de i coups.

9. $\mathcal{A}_0 = \{S_6^2\}, \mathcal{A}_1 = \{S_6^2, S_2^1, S_4^1\} = \mathcal{A}$. L'attracteur contient exactement toutes les positions gagnantes pour J_1.

Exercice 15 *Déduction naturelle (exo 0)*

1. L'arbre de preuve A_1 suivant convient :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}\,\text{AX} \quad \dfrac{}{A, \neg A \vdash \neg A}\,\text{AX}}{A, \neg A \vdash \bot}\,\neg_e}{\neg A \vdash A \to \bot}\,\to_i}{\vdash \neg A \to (A \to \bot)}\,\to_i$$

2. L'arbre de preuve A_2 suivant convient :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \to \bot \vdash A}\,\text{AX} \quad \dfrac{}{A, A \to \bot \vdash A \to \bot}\,\text{AX}}{A, A \to \bot \vdash \bot}\,\to_e}{A \to \bot \vdash \neg A}\,\neg_i}{\vdash (A \to \bot) \to \neg A}\,\to_i$$

3. Dans cet ordre : l'introduction, l'élimination à gauche et celle à droite :

$$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge_i \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge_{eg} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge_{ed}$$

   Le séquent demandé se dérive immédiatement des arbres A_1 et A_2 et de l'application de $\wedge_i$.

4. On peut par exemple dresser une table de vérité :

A	B	$A \to B$	$(A \to B) \to A$	P
0	0	1	0	1
0	1	1	0	1
1	0	0	1	1
1	1	1	1	1

   On constate que P est effectivement une tautologie.

5. On complète la branche de gauche avec l'arbre suivant (l'idée étant de prouver $A \to B$ à partir de Γ. En effet, il ne suffit plus ensuite que d'utiliser $(A \to B) \to A$ pour obtenir A) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash A}\,\text{AX} \quad \dfrac{}{\Gamma, A \vdash \neg A}\,\text{AX}}{\Gamma, A \vdash \bot}\,\neg_e}{\Gamma, A \vdash B}\,\bot_e}{\Gamma \vdash A \to B}\,\to_i \qquad \dfrac{}{\Gamma \vdash (A \to B) \to A}\,\text{AX}}{\Gamma \vdash A}\,\to_e$$

Exercice 16 *SQL (exo 0)*

1. A mon avis, la seule clé primaire pertinente pour `Achat` serait un nouvel attribut permettant d'identifier de manière unique chaque achat. En effet :

   - Aucun attribut seul ne peut être une clé primaire.
   - La couple (`NumClient`, `NomFabricant`) ne convient pas non plus puisqu'un client pourrait acheter plusieurs objets différents chez le même fabricant.
   - Le triplet (`NumClient`, `NomFabricant`, `Modele`) ne convient pas vraiment non plus car un même client pourrait acheter un même matériel au même fabricant plusieurs fois. Ou alors il faudrait qu'à chaque fois que cela arrive le champ `Quantite` soit modifié.
   - Prendre les quatre attributs comme clé primaire n'est pas très indiqué non plus : un client pourrait acheter le même matériel au même fabricant dans les mêmes quantités plusieurs fois.

2. - `Modele` est une clé étrangère dans `Production` référant à `Modele` dans les tables de matériels.
   - `NomFabricant` est une clé étrangère dans `Production` référant à `Nom` dans la table `Fabricant`.
   - `NumClient` est une clé étrangère dans `Achat` référant à `Num` dans la table `Client`.
   - `NomFabricant` est une clé étrangère dans `Achat` référant à `Nom` dans la table `Fabricant`.
   - `Modele` est une clé étrangère dans `Achat` référant à `Modele` dans les tables de matériels.
   - (`NomFabricant`, `Modele`) est une clé étrangère dans `Achat` référant à la clé primaire du même nom dans la table `Production`.

3. (a) Une solution :

```
SELECT Modele FROM Production AS p
WHERE p.Nom = 'Durand'
```

   (b) Pour rappel, le renommage de table peut se faire sans le mot clé `AS`. Le renommage de colonne lui, demande ce mot clé. On pourrait aussi faire un produit cartésien.

```
SELECT Nom, Adresse FROM
  Fabricant f JOIN Production prod ON f.Nom = prod.NomFabricant
              JOIN Portable p ON p.Modele = prod.Modele
WHERE p.Dd >= 500
```

   (c) On calcule pour chaque fabricant combien de modèles d'imprimantes il fabrique :

```
SELECT Nom, COUNT(DISTINCT Modele) AS c FROM
  Fabricant f JOIN Production p ON f.Nom = p.NomFabricant
              JOIN Imprimante i ON i.Modele = p.Modele
GROUP BY Nom
HAVING c >= 10
```

   Il ne reste plus qu'à projetter sur le premier attribut de la table obtenue.

   (d) On calcule le numéro des clients qui ont acheté une imprimante puis on les supprime des numéros de l'ensemble des clients avec `EXCEPT` :

```
SELECT Num FROM Client
EXCEPT
SELECT DISTINCT Num FROM
  Client c JOIN Achat a ON c.Num = a.NumClient
           JOIN Production p ON p.NomFabricant = a.NomFabricant
           JOIN Imprimante i ON i.Modele = p.Modele
```

Exercice 17 *Activation de processus (exo 2023)*

1. Oui pour $k = 2$ car on peut activer P_2 et P_3 en même temps. Non pour $k = 3$ car si on active trois processus, au moins deux utilisent la ressource r_1.

   Jury : On attend du candidat une réponse en oui / non avec une précision de quels processus activer dans le cas où la réponse est oui et une très rapide justification dans le cas où la réponse est non. Une réponse orale suffit.

2. On suppose que l'entrée de l'algorithme est un ensemble de n couples (numéro de la machine, numéro de la ressource qu'elle utilise). On peut alors proposer l'algorithme suivant :

```
Initialiser un tableau T à m cases contenant faux
nb_machines = 0
pour chaque couple (n_machine, n_ressource)
    si T[n_ressource] = faux alors
        T[n_ressource] = vrai
        Incrémenter nb_machines
renvoyer nb_machines
```

Le principe est que le tableau T indique en case i si la i-ème ressource est utilisée ou non. On pourrait même calculer sans surcoût les machines activer en plus de leur nombre avec cette méthode. La complexité de cette méthode est en $O(n+m)$ (à cause de l'initialisation de T) mais on pourrait la faire passer à un $O(n)$ en stockant les informations stockées dans T dans un dictionnaire (les clés seraient les numéros des ressources, et les valeurs n'ont pas d'importance).

Jury : De multiples réponses sont possibles sur cette question et sont acceptées du moment qu'elles sont clairement décrites et correctement analysées. On n'attend pas une complexité optimale.

3. Un sous-ensemble de numéros de machines convient (et il est de taille au plus n donc polynomial en la taille de l'entrée de `ACTIVATION`).

   Jury : Une courte phrase décrivant la nature d'un tel certificat et indiquant sa polynomialité en la taille de l'entrée suffit à obtenir tous les points.

4. On propose le code de vérification suivant étant donné un certificat c :

```
Ressources_utilisées = ∅
pour toute machine i
    si appartient(c, i) alors
        si intersecte(P_i, Ressources_utilisées) alors
            renvoyer faux
        Ressources_utilisées = ajoute(P_i, Ressources_utilisées)
renvoyer |c| ≥ k
```

   Pour chaque machine, si elle appartient à l'ensemble des machines qu'on souhaite activer simultanément, on vérifie qu'elle n'entre pas en conflit sur les ressources avec une des machines qui la précède (ce qui suffit par symétrie de la relation "entrer en conflit"). La variable Ressources_utilisées représente l'ensemble des ressources utilisées par les machines considérées jusque là. S'il n'y a pas de conflit, on compte le nombre de processus dans c et on renvoie vrai s'il y en a au moins k, faux sinon.

   Jury : On attend un algorithme utilisant à bon escient les primitives proposées par le sujet.

5. Les questions 3 et 4 montrent déjà que `ACTIVATION` est dans NP puisque l'algorithme en Q4 est de complexité polynomiale en la taille de l'entrée du problème d'après l'énoncé

   Soit $I_S = \{G = (S, A), k\}$ une instance de `STABLE`. Considérons alors l'instance suivante I_A de `ACTIVATION` : on créé une machine i par sommet de S, une ressource r par arête et la liste de ressources nécessaires à la machine i est $\{r \text{ ressource } | \, r \text{ correspond à une arête incidente à } i\}$; on conserve par ailleurs k. Elle est constructible à partir de I_S en temps $O(|S| + |A|)$. De plus S' est un stable de taille k dans G si et seulement si les k machines de S' sont activables en même temps.

   Ceci montre que `STABLE` $\leq$ `ACTIVATION` et comme le premier est NP-difficile, il en va de même pour le second, ce qui achève la preuve de NP-complétude de `ACTIVATION`.

   Jury : Le jury est attentif au fait que tous les arguments soient bien présents : description de la réduction, preuve que c'en est une et justification de son caractère polynomial pour le caractère NP-difficile ; caractère NP pour pouvoir conclure quant à la NP-complétude.

Exercice 18 *Minima locaux dans des arbres (exo 2023)*

1. Les nœuds d'étiquettes -4, 0 et 1 sont les trois minima locaux.

   Jury : On attend simplement du candidat qu'il propose les minima trouvés sans justification. Une réponse orale suffit. En cas d'erreur, le candidat est invité à expliquer son raisonnement.

2. Un arbre est soit un arbre vide soit un nœud formé d'une étiquette et de deux sous-arbres. La hauteur est la profondeur maximale d'une feuille, c'est-à-dire la longueur maximale d'un chemin de la racine à une feuille. La hauteur de l'arbre (b) est 3.

   Jury : Dans le cas où le candidat propose une définition inductive des arbres avec une feuille pour cas de base, il est guidé vers une définition dont le cas de base est l'arbre vide de sorte à rendre compte du fait que les arbres considérés ne sont pas binaires stricts.

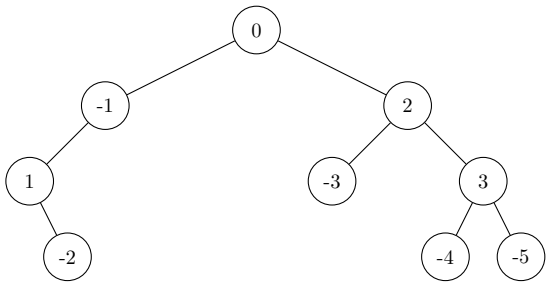
3. L'arbre possède un nombre fini non vide d'étiquettes et donc une étiquette de valeur minimale, qui est un minimum global et donc local.

   Jury : Plusieurs stratégies sont ici acceptées (preuve par induction ou existence d'un minimum global qui est local par exemple).

4. Si la racine de l'arbre est un minimum local on a trouvé notre minimum local. Sinon, un des deux fils est non vide, avec une étiquette à sa racine plus petite que celle de la racine de l'arbre. Un appel récursif permet d'obtenir un minimum local de ce sous-arbre, qui est également un minimum local de l'arbre (que ce soit la racine du sous-arbre ou un descendant strict). La complexité est linéaire en la hauteur de l'arbre.

   Jury : Si le candidat propose une solution linéaire en la taille de l'arbre, il est guidé vers une solution linéaire en la hauteur.

5. On propose l'étiquetage à la figure (c) dans lequels les 5 minima locaux sont étiquetés par des entiers strictement négatifs.



(c)

   Jury : Proposer un étiquetage correct suffit à obtenir tous les points.

6. On propose une approche récursive qui pour un arbre a en entrée calcule $m(a)$ le nombre maximal de nœuds qui peuvent être des minima locaux dans un étiquetage de a, ainsi que, en même temps, la quantité $m_-(a)$ correspondant à cette même quantité mais en supposant de plus que la racine n'est pas un minimum local. Pour un arbre vide, ces deux valeurs valent 0. Pour un arbre a de fils gauche f_g et de fils droit f_d, on peut obtenir par appels récursifs les quantités $m(f_g)$, $m_-(f_g)$, $m(f_d)$ et $m_-(f_d)$. On a alors $m_-(a) = m(f_g) + m(f_d)$ et $m(a) = \max\{m_-(a), 1 + m_-(f_g) + m_-(f_d)\}$. La complexité est linéaire en la taille de l'arbre, chaque nœud est visité exactement une fois avec un nombre d'opérations constant.

   Jury : Le jury attend une complexité linéaire. Des indications peuvent être apportées pour aiguiller le candidat vers une telle solution.

7. Le résultat est vrai pour $n = 0$ et on peut donc supposer que $n \geq 1$. Considérons un étiquetage et notons X l'ensemble des nœuds qui sont des minima locaux et Y ceux qui ne le sont pas. On remarque

que deux nœuds adjacents ne peuvent pas être tous les deux des minima locaux, puisque toutes les étiquettes sont deux à deux distinctes. Ainsi toute arête de l'arbre est extrémité d'au moins un nœud de Y et l'ensemble Y couvre donc toutes les arêtes. Comme chaque nœud de Y est incident à au plus 3 arêtes et qu'il y a exactement $n-1$ arêtes dans l'arbre, il faut au moins $\frac{n-1}{3}$ nœuds pour couvrir toutes les arêtes, c'est-à-dire $|Y| \geq \frac{n-1}{3}$. On a donc $|X| = n - |Y| \leq n - \frac{n-1}{3} = \frac{2n+1}{3}$ et donc $|X| \leq \left\lfloor \frac{2n+1}{3} \right\rfloor$.

Jury : Très peu de candidats ont pu aborder cette question.

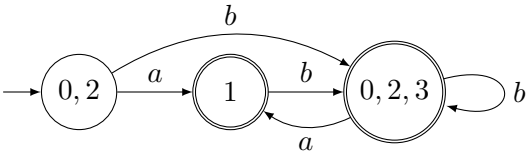
Exercice 19 *Langages réguliers (exo 0)*

1. Un langage est régulier s'il est dénoté par une expression régulière sur un certain alphabet Σ. Les expressions régulières sur Σ sont définies par induction par :
   - $\emptyset$, ε et a pour $a \in \Sigma$ sont des expressions régulières.
   - Si f et g sont des expressions régulières, $(f+g)$, fg et $f^\star$ aussi.
2. (a) L_1 est régulier car il est dénoté par $a^\star b a^\star$.

   (b) L_2 n'est pas régulier. Par l'absurde, s'il l'était, le lemme de l'étoile fournirait $N \in \mathbb{N}$ tel que le mot $a^N b a^N \in L_2$ se factoriserait en $a^{n_1} a^{n_2} a^{N-n_1-n_2} b a^N$ avec $n_1 \in \mathbb{N}$, $n_2 \in \mathbb{N}^\star$ et de sorte à ce que pour tout $k \in \mathbb{N}$ on ait $a^{N+(k-1)n_2} b a^N \in L_2$. Pour $k = 2$, on aurait $a^{N+n_2} b a^N \in L_2$ ce qui n'est pas car $N + n_2 > N$ en vertu du fait que n_2 est non nul.

   (c) L_3 n'est pas régulier. S'il l'était, le langage $L_3^c \cap a^\star b a^\star$ le serait aussi par stabilité des langages réguliers par le complémentaire et l'intersection ; or ce langage est L_2 qui n'est pas régulier.

   (d) L_4 est le langage des mots de la forme $a^n b a^m$ avec n, m tous les deux pairs ou n, m tous les deux impairs. Il est donc dénoté par $(a^2)^\star b (a^2)^\star + a(a^2)^\star b a (a^2)^\star$ et est donc régulier.
3. (a) On trouve l'automate des parties suivant (si on ne complète pas) :



   (b) La seule méthode au programme pour conclure de manière indiscutable est l'élimination des états. Sinon on peut aussi tricher, répondre à la question suivante et en déduire une expression rationnelle ; par exemple $b^\star(ab+b)^\star(a+\varepsilon)$ semble convenir.

   (c) Les mots de ce langage sont les mots dans lesquels il n'y a jamais deux a consécutifs.

Exercice 20 *Mutex (exo 0)*

1. A priori, toute valeur entre 1 et 100 (inclus de deux côtés) est possible.
2. Pour obtenir 100 de manière certaine, on protège la section critique comme suit :

```
let m = Mutex.create ()

let f i =
  Mutex.lock m;
  compteur := !compteur + 1
  Mutex.unlock m;
```

3. Avec `f_a` on peut avoir famine avec la trace d'exécution suivante :
   - Le fil 0 calcul `autre` puis indique que `veut_entrer.(0)` vaut `true`.

- Le fil 1 calcul `autre` puis indique que `veut_entrer.(1)` vaut `true`.
- Les deux fils sont alors bloqués dans la boucle `while`.

Avec `f_b` il n'y a pas exclusion mutuelle ce qui s'observe via la trace suivante :

- Le fil 0 calcule `autre`, indique qu'il veut entrer, fixe `tour` à 0 puis entre en section critique (ce qui est possible puisque ce n'est pas le tour du fil 1).
- Le fil 1 fait exactement la même chose.
- Les deux fils sont alors en même temps en section critique.

4. Il s'agit de reconstruire l'algorithme de Peterson :

```
let f_c i =
  let autre = 1-i in
  veut_entrer.(i) <- true;
  tour := autre;
  while veut_entrer.(autre) && !tour = autre do () done;
  (* section critique *)
  veut_entrer.(i) <- false;
```

5. L'algorithme de la boulangerie de Lamport permet de généraliser à n fils. On y utilise un tableau `veut_entrer` à n cases et un tableau d'entiers `tickets` à n cases. Lorsqu'un fil i veut entrer en section critique, il modifie sa case `veut_entrer.(i)` puis calcule et stocke son ticket comme étant (le maximum du tableau `tickets`) $+1$. Il attend ensuite d'être le fil souhaitant aller en section critique disposant du couple (`tickets.(i)`, i) le plus petit pour l'ordre lexicographique.

Exercice 21 *Grammaires algébriques (exo 2023)*

1. Cette grammaire est ambiguë car le mot $babab \in L(G)$ admet les deux arbres syntaxiques suivants :

Or ces derniers sont différents.

Jury : Le jury attend du candidat qu'il exhibe un mot montrant l'ambiguïté, en justifiant cette dernière via deux arbres syntaxiques différents ou bien deux dérivations gauches (ou droites) différentes.

2. On constate que $L(G)$ est rationnel car dénoté par l'expression rationnelle $(ba)^\star b$.

Jury : Il est attendu que le candidat exhibe une expression régulière pour ce langage (qui est donc régulier). Plusieurs sont possibles et toutes sont acceptées. Aucune justification n'est nécessaire à ce stade, sauf si l'expression proposée est démesurément complexe, auquel cas le jury invite le candidat à lui expliquer. Certains candidats n'ont pas compris ce qui était attendu par « la plus petite classe de langages », le jury a alors précisé ses attentes sans pénaliser pour autant ces candidats.

Au moins un candidat a répondu à cette question en affirmant que $L(G)$ est en fait local. Il a été invité à poursuivre l'exercice avec cette réponse.

3. Montrons par récurrence forte sur $n \in \mathbb{N}^\star$ la propriété $H(n)$ suivante : si $u \in L$ est un mot de taille n alors u est engendré par la grammaire G. C'est bien sûr le cas pour $n = 1$ puisque le seul mot de L de cette taille est b, engendré par la deuxième règle de G.

Soit donc $n \in \mathbb{N}^\star$ et u un mot de L de taille $n+1$. Comme $|u| \geq 2$, ba est nécessairement préfixe de u et il existe donc $v \in (ba)^\star b = L$ tel que $u = bav$. Par hypothèse, ce mot v est engendré par G : il existe une dérivation telle que $S \Rightarrow^\star v$. On en déduit que

$$S \Rightarrow SaS \Rightarrow baS \Rightarrow^\star bav = u$$

est une dérivation licite et donc que $u \in L(G)$. Cette récurrence montre que $L \subset L(G)$.

Montrons réciproquement que $L(G) \subset L$ en montrons par récurrence forte sur $n \in \mathbb{N}^\star$ la propriété $H(n)$ suivante : si $u \in \Sigma^\star$ se dérive de S en n dérivations alors $u \in L$. C'est acquis pour $n = 1$: le seul mot de $\Sigma^\star$ qu'on peut obtenir en une dérivation est $b \in L$.

Soit donc $n \in \mathbb{N}^\star$ et u un mot dans $L(G)$ tel que $S \Rightarrow^{n+1} u$. Comme ce mot est obtenu en au moins 2 dérivations, les règles de G nous informent que la première est nécessairement $S \to SaS$ (sans quoi ce serait $S \to b$ et dans ce cas u serait obtenu en une seule dérivation). Donc la dérivation permettant d'obtenir u se décompose en :

$$S \Rightarrow SaS \Rightarrow^n u$$

On en déduit qu'il existe $v, w \in \Sigma^\star$ et $k_1, k_2 \in [\![1, n]\!]$ tels que $u = vaw$, $S \Rightarrow^{k_1} v$, $S \Rightarrow^{k_2} w$ et $k1 + k2 = n$. L'hypothèse de récurrence (forte) s'applique à v et w et on en déduit que ces deux mots appartiennent au langage dénoté par $(ba)^\star b$ donc qu'il existe $r_1, r_2 \in \mathbb{N}$ tels que $v = (ba)^{r_1} b$ et $w = (ba)^{r_2} b$. Par conséquent, $u = (ba)^{r_1} ba (ba)^{r_2} b = (ba)^{r_1 + r_2 + 1} b \in L$.

Jury : On attend une preuve précise et rigoureuse, par exemple par double inclusion. Les explications vagues et les arguments d'évidence ne satisfont pas le jury sur cette question.

4. On sait à présent que $L(G) = (ba)^\star b$; il s'agit donc de trouver une grammaire non ambiguë engendrant ce langage. On peut proposer par exemple la grammaire dont les règles sont :

$$S \to Tb \qquad T \to baT \,|\, \varepsilon$$

les règles sur T permettant de générer le facteur dans $(ba)^\star$ et la première de rajouter le b final.

Cette grammaire G' est non ambiguë car pour tout mot dans $L(G')$, il existe une unique dérivation permettant de le construire (donc évidemment un seul arbre syntaxique) ; cette unicité découlant du fait que dans cette grammaire un non terminal se dérive toujours en un mot qui contient au plus un seul non terminal.

Jury : Comme pour la deuxième question, plusieurs grammaires sont ici possibles. On n'attend pas une preuve rigoureuse de non ambiguïté.

5. La question demande de montrer que tout langage rationnel peut être engendré par une grammaire non ambiguë. Soit donc L un langage rationnel. Par le théorème de Kleene, il existe un automate fini $A = (\Sigma, Q, \{q_0\}, F, \delta)$ qui reconnaît L qu'on peut loisiblement supposer déterministe.

Considérons alors la grammaire dont les non terminaux sont $\{V_q \,|\, q \in Q\}$, l'axiome est V_{q_0}, les terminaux sont les lettres de Σ et dont les règles sont données par :

- Pour toute transition $q \xrightarrow{a} q'$ dans A, on ajoute la règle $V_q \to aN_{q'}$.
- Pour tout $q \in F$, on ajoute la règle $V_q \to \varepsilon$.

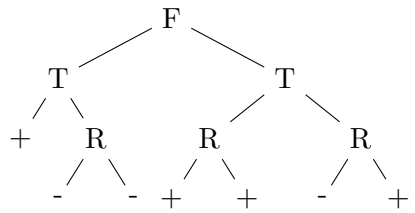
Cette grammaire engendre L de manière non ambiguë graâce au déterminisme de A.

Jury : Proposer une construction correcte d'une grammaire non ambiguë pour un langage régulier (a priori à partir de l'automate associé) suffit à obtenir tous les points. Le jury n'hésitait pas à aiguiller le candidat si nécessaire pour cette question.

Exercice 22 *ID3 (exo 0)*

1. Dans un problème d'apprentissage supervisé, on dispose d'objets ayant des attributs et une étiquette. L'objectif est de construire une fonction (un modèle) à partir d'exemples connus qui à partir des attributs d'un objet potentiellement inconnu indique son étiquette.

2. On obtient l'arbre suivant en prenant comme conventions qu'une feuille est étiquetée par la classe majoritaire des exemples qu'elle représente et que dans le cas où une branche ne recoupe plus aucun exemple, on créé une feuille dont la classe est la classe majoritaire de son noeud père :



   Par exemple, sur la branche gauche, droite, droite, qui correspond à F = NO, T = YES, R = YES, les exemples correspondants sont 9 (+), 12 (-) et 13 (-) donc on étiquette la feuille par -. Sur la branche droite, gauche, gauche, qui correspond à F = YES, T = NO, R = NO, il n'y a aucun exemple associé : on créé une feuille dont l'étiquette est celle majoritaire parmi 4, 7 et 8 qui sont les exemples pour lesquels F = YES et T = NO.

   Remarquons que cet arbre est inutilement complexe : certains branchements semblent inutiles.

3. On devrait trouver 0.47 (arrondi au centième), cf le fichier ID3.ml pour les calculs.

4. Indirectement on l'a déjà fait en question 2 :

   - Si l'ensemble des exemples est vide, on créé une feuille avec une étiquette par défaut ; par exemple, on peut choisir l'étiquette majoritaire des exemples dans le noeud parent.
   - Si l'ensemble des attributs est vide, on ne peut plus séparer les exemples : il faut donc créer une feuille qu'on étiquette par la classe majoritaire des exemples encore présents.
   - Si tous les exemples sont dans la même classe, on créé une feuille étiquetée par cette classe.

Exercice 23 *Formules propositionnelles croissantes (exo 2023)*

1. On fixe $a : E \to \{V, F\}$ une valuation. On étend a par induction comme suit

$$\text{Si } P \text{ et } Q \text{ sont des formules propositionnelles, alors } \begin{array}{l} a(\neg P) = \text{ Négation de } a(P). \\ a(P \wedge Q) = a(P) \text{ et } a(Q), \\ a(P \vee Q) = a(P) \text{ ou } a(Q). \end{array}$$

   Jury : Le jury profite de cette question pour vérifier que le candidat sait faire la différence entre syntaxe et sémantique.

2. Soient P, Q deux formules croissantes. Montrons que $P \wedge Q$ et $P \vee Q$ sont croissantes.

   Soient a, b deux valuations vérifiant $a \leq b$. En utilisant l'abus de notation, on a les égalités suivantes :

$$\begin{array}{ll} a(P \wedge Q) = \min(a(P), a(Q)), & a(P \vee Q) = \max(a(P), a(Q)), \\ b(P \wedge Q) = \min(b(P), b(Q)), & b(P \vee Q) = \max(b(P), b(Q)). \end{array}.$$

   Sachant que $a \leq b$, on a donc $a(P) \leq b(P)$ et $a(Q) \leq b(Q)$. On en déduit :

$$\min(a(P), a(Q)) \leq \min(b(P), b(Q)) \text{ et } \max(a(P), a(Q)) \leq \max(b(P), b(Q))$$

   Autrement dit $a(P \wedge Q) \leq b(P \wedge Q)$ et $a(P \vee Q) \leq b(P \vee Q)$.

$P \wedge Q$ et $P \vee Q$ sont bien croissantes.

Jury : Le candidat est autorisé à utiliser l'abus introduit par l'énoncé (assimiler V à 1 et F à 0) mais le jury se réserve le droit de vérifier que le candidat a bien compris en quoi c'est un abus. Un candidat qui prouve rigoureusement la propriété souhaitée pour un des connecteurs et indique ce qu'il suffirait d'y changer pour l'autre obtient tous les points.

3. On montre par double implication.

   - (Sens réciproque). Soit C une clause conjonctive satisfiable contenant au moins un littéral. Un clause contenant un unique littéral positif est clairement croissant. Par récurrence et par stabilité des formules croissantes par l'opération $\wedge$, on en déduit que tout clause conjonctive contenant uniquement des littéraux positifs est croissante.
   - (Sens direct, par la contraposée). Considérons une clause conjonctive C satisfiable contenant au moins un littéral. Montrons que si celle-ci est croissante, alors tout littéral apparaissant dans C est positif. Supposons que C contient au moins un littéral négatif et quitte à réarranger les termes, on considère que $C = \neg x \wedge C'$ avec C' une clause conjonctive ne contenant pas x (possible car C est satisfiable).

     Considérons une valuation a vérifiant $a(C) = V$. On a alors $a(\neg x) = V$ et $a(C') = V$. Donc $a(x) = F$. On choisit b qui coïncide avec a sur toutes les variables propositionnelles exceptée sur x où on a $b(x) = V$. Par construction, $a \leq b$. Mais $b(C) = F$. Donc C n'est pas une formule croissante.

   On a bien l'équivalence demandée.

   Jury : Il ne faut pas oublier de traiter le caractère nécessaire et suffisant.

4. (a) Si P est logiquement équivalente à une disjonction de clauses conjonctives sans littéral négatif, par stabilité de la croissance par disjonction et les clauses conjonctives sans littéral négatif étant croissante, on en déduit que P est croissante.

   (b) Récriproquement, soit P une formule croissante. On pose :

   $$P' = \bigvee_{C \vDash P} C$$

   où pour deux formules P, Q, on a $P \vDash Q$ si pour toute valuation a , $a(P) = V \implies a(Q) = V$.

   Dans notre cas, C désigne une clause conjonctive croissante inférieure à P (Autrement dit, pour toute valuation a : $a(C) \leq a(P)$). Vérifions que P' est logiquement équivalente à P.

   - soit a une valuation vérifiant $a(P') = V$.Il existe $C \leq P$ une clause telle que $a(C) = V$. On a bien $a(P) = V$.
   - Réciproquement, si a est une valuation vérifiant $a(P) = V$, on pose :

     $$C = \bigwedge_{x \text{ positif } a(x)=V} x$$

     Remarquons que si C était indexé par l'ensemble vide, a enverrait tous les littéraux sur F et dans ce cas par croissance, P serait une tautologie ce qui contredit l'hypothèse de l'énoncé. Les propriétés suivantes sont alors vérifiées :

     – C n'est pas indexé par le vide (sinon, a enverrait tous les littéraux positifs sur F et par croissance, P serait une tautologie)

     – C contient uniquement des littéraux positifs,

     – on a $C \vDash P$ par croissance de P.

Donc C apparaît dans P'. D'où $a(P') = V$.

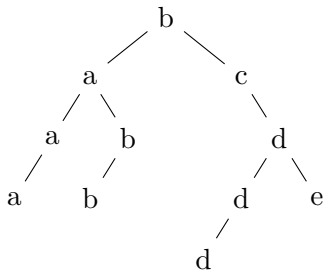
Ainsi, P et P' sont bien logiquement équivalentes.

Jury : Le jury peut donner une indication pour aider le candidat à trouver une formule permettant de répondre à cette question.

Exercice 24 *Arbres binaires de recherche (exo 0)*

1. Un arbre binaire de recherche est un arbre binaire dont les clés sont à valeurs dans un ensemble totalement ordonné. De plus, pour chaque noeud de cet arbre, son étiquette x est strictement supérieure aux étiquettes dans son fils gauche et strictement inférieure aux étiquettes dans son fils droit (des variantes autorisent une inégalité large d'un des deux côtés).

2. On décide de mettre les cas d'égalité à gauche (par exemple) :



3. On procède par induction. On note $P(a)$ le parcours infixe de l'ABR a. Le parcours infixe d'un ABR vide est bien sûr trié. De plus, si a est un ABR de racine x et de fils gauche (respectivement droit) g (respectivement d) alors $P(a) = P(g), x, P(d)$. Par hypothèse inductive, $P(g)$ et $P(d)$ sont triés dans l'ordre croissant. Par définition d'un ABR, x est plus grand que les éléments apparaissant dans $P(g)$ et plus petit que ceux dans $P(d)$: $P(a)$ est donc trié dans l'ordre croissant.

4. On procède récursivement :

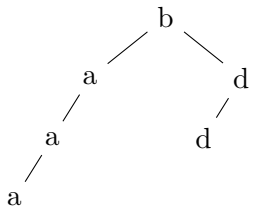
   - Le nombre d'occurrences d'une lettre u dans un arbre vide vaut 0.
   - Si l'ABR n'est pas vide, on compare u à sa racine x. Si $u < x$, on compte le nombre d'occurrences de u à gauche, si $u > x$, on compte le nombre d'occurrences de u à droite et si $u = x$, on compte le nombre d'occurrences de u à gauche et on lui ajoute 1.

   En fait, dès qu'on a trouvé une occurrence de u, les autres sont forcément à gauche et on peut ainsi compter les occurrences de u en ne faisant le parcours que d'une branche de l'arbre. D'où une complexité en $O(h)$ où h est la hauteur de l'arbre.

5. On commence par chercher une occurrence de l'élément s à supprimer en descendant soit à gauche soit à droite selon comment s se compare à la racine courante. Une fois s trouvé :

   - S'il n'a pas de fils, on peut le supprimer directement.
   - S'il en a un, on remplace l'arbre enraciné en s par ce fils.
   - S'il en a deux, on récupère le maximum m du fils gauche, on écrase s avec m puis on supprime m. Cet appel récursif aboutit à l'un des deux cas précédents car par définition le maximum d'un ABR se trouve en descendant systématiquement à droite dont m n'aura pas de fils droit.

   Sur l'arbre de la question 2 on obtient après suppressions :



La complexité d'une suppression est logarithmique en la taille de l'arbre.