Notes : Toutes les réponses doivent être justifiées.

Pour exprimer les algorithmes, un candidat a le choix entre l'utilisation des langages Pascal ou Caml.

Les deux exercices sont indépendants.

Exercice I -

Soit $P = \{p_0, p_1, ..., p_n, ...\}$ un ensemble dénombrable de symboles de proposition.

On considère l'ensemble $A(P)$ des formules propositionnnelles définies sur l'ensemble P comme étant le plus petit ensemble tel que :

- chaque symbole p_i est dans $A(P)$;
- les symboles **0** et **1** sont dans $A(P)$;
- si $A \in A(P)$, alors $\neg A \in A(P)$;
- si $A \in A(P)$ et $B \in A(P)$, alors $(A \wedge B) \in A(P)$, $(A \vee B) \in A(P)$, $(A \Rightarrow B) \in A(P)$
- les formules de $A(P)$ sont obtenues uniquement par application, un nombre fini de fois, des règles précédentes.

Une formule propositionnelle est représentée par un arbre dont les nœuds sont les connecteurs logiques et les feuilles les symboles de proposition, **0** et **1**. On appelle *taille* d'une formule la hauteur de l'arbre associé.

On définit le domaine $B = \{V,F\}$ des valeurs de vérité. Les connecteurs logiques $\neg, \wedge, \vee, \Rightarrow$ ont pour tables de vérité associées :

b_1	b_2	$H^{\neg}(b_1)$	$H^{\wedge}(b_1,b_2)$	$H^{\vee}(b_1,b_2)$	$H^{\Rightarrow}(b_1,b_2)$
F	F	V	F	F	V
F	V	V	F	V	V
V	F	F	F	V	F
V	V	F	V	V	V

On appelle valuation toute fonction $v : P \to B$ qui assigne une valeur de vérité à un symbole de proposition. Cette valuation v s'étend en une unique fonction $\hat{v} : A(P) \to B$ définie par les égalités suivantes :

$$\hat{v}(\mathbf{0}) = F \quad (1)$$

$$\hat{v}(\mathbf{1}) = V \quad (2)$$

$$\forall p \in P \quad \hat{v}(p) = v(p) \quad (3)$$

$$\hat{v}(\neg A) = H^{\neg}(\hat{v}(A)) \quad (4)$$

$$\hat{v}(A \wedge B) = H^{\wedge}(\hat{v}(A), \hat{v}(B)) \quad (5)$$

$$\hat{v}(A \vee B) = H^{\vee}(\hat{v}(A), \hat{v}(B)) \quad (6)$$

$$\hat{v}(A \Rightarrow B) = H^{\Rightarrow}(\hat{v}(A), \hat{v}(B)) \quad (7)$$

Pour une proposition A dépendant de n symboles de proposition distincts $p_1, ..., p_n$, on notera la fonction $H_A : B^n \to B$ qui, pour une valuation v, associe à $(v(p_1), ..., v(p_n))$ la valeur de vérité $\hat{v}(A)$. On dit que H_A est la *table de vérité* de la formule A.

Si A et B sont deux formules de A (P), on dit que A et B sont équivalentes, ce que l'on note $A \equiv B$, si et seulement si quelle que soit la valuation v sur P, $\hat{v}(A) = \hat{v}(B)$

Une formule propositionnelle A est dite *satisfiable* lorsqu'il existe une *valuation* v des propositions, qui rend la formule vraie, c'est-à-dire $\hat{v}(A) = V$. C'est une *tautologie* lorsque toutes les valuations rendent la formule vraie.

On s'intéresse dans cet exercice au problème de la satisfiabilité : étant donnée une formule de la logique des propositions, cette formule est-elle satisfiable ? Est-ce une tautologie ?

On appelle *littéral* un symbole de proposition p (littéral *positif*) ou la négation d'un symbole de proposition $\neg p$ (littéral *négatif*) ou l'une des constantes **0, 1**.

Une *clause* est une disjonction $l_1 \vee ... \vee l_n$ où les l_i sont des littéraux deux à deux distincts. On notera une clause sous la forme $[l_1, ..., l_n]$. Une *clause duale* est une conjonction $l_1 \wedge ... \wedge l_n$ où les l_i sont des littéraux deux à deux distincts. On notera une clause duale sous la forme $< l_1, ..., l_n >$. On dit qu'une formule propositionnelle A est une *forme normale disjonctive* ou *forme clausale duale* si et seulement si $A = D_1 \vee D_2 \vee ... \vee D_n$ où chaque D_i est une clause duale. Par extension de la notation, on écrira $A = [D_1, D_2, ..., D_n]$. Le cas $n = 0$ correspond à $A = \mathbf{0}$.

On dit qu'une formule propositionnelle A est une *forme normale conjonctive* ou *forme clausale* si et seulement si $A = C_1 \wedge C_2 \wedge ... \wedge C_n$ où chaque C_i est une clause. Par extension de la notation, on écrira $A = < C_1, C_2, ..., C_n >$. Le cas $n = 0$ correspond à $A = \mathbf{1}$.

Par extension, si

$$A \equiv \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} C_{i,j} \right)$$

où les $C_{i,j}$ sont des formules propositionnelles, on pourra écrire $D_i = < C_{i,1}, ..., C_{i,m_i} >$ et $A = [D_1, ..., D_n]$.

De même, on étend la notation duale.

On admettra le théorème : "toute formule propositionnelle est équivalente à une forme clausale et est équivalente à une forme clausale duale". Il est intéressant, pour la suite, de savoir qu'on ne connaît pas d'algorithme général de mise sous forme clausale ou sous forme clausale duale d'une formule propositionelle qui soit de complexité non exponentielle.
Par convention, dans les formes clausales duales autres que **1** (resp. clausales autres que **0**), on éliminera les clauses duales contenant **0** ou à la fois p et $\neg p$ (resp. les clauses contenant **1** ou à la fois p et $\neg p$). On dira alors que ces formes sont réduites.

I.A - À quelle condition une formule propositionnelle en forme normale disjonctive réduite est-elle satisfiable ?

On définit $\oplus$ le connecteur logique : $P \oplus Q$ est une abréviation pour $(P \wedge \neg Q) \vee (\neg P \wedge Q)$.

I.B - Montrer que, à équivalence près, $\oplus$ est commutatif et associatif et que $\wedge$ est distributif sur $\oplus$. Montrer que $(\oplus, \wedge, \mathbf{1})$ est fonctionnellement complet, c'est-à-dire que toute fonction booléene à n arguments est la table de vérité d'une formule propositionnelle A n'utilisant que ces connecteurs et la constante **1**.

I.C - En utilisant les questions précédentes, montrer que chaque formule propositionnelle A est équivalente à une autre formule A' qui est de la forme **0, 1** ou $C_1 \oplus C_2 \oplus \ldots \oplus C_n$ où chaque C_i est **1** ou une conjonction de littéraux positifs. Montrer, de plus, que A' peut être choisie de manière à ce que les C_i soient tous distincts (à permutation près) et les littéraux positifs de chaque C_i distincts également (à permutation près). On dira alors que A' est sous forme *ou-exclusive normale réduite*.

On admettra l'unicité, à permutation près, d'une formule sous forme ou-exclusive normale réduite équivalente à une formule donnée A, ce qui signifie que si A' et A'' sont deux telles formules, alors elles sont identiques à l'ordre près c'est-à-dire :

soit $A' = A'' = \mathbf{0}$,

soit $A' = A'' = \mathbf{1}$,

soit $A' = C'_1 \oplus \ldots \oplus C'_p$ $A'' = C''_1 \oplus \ldots \oplus C''_q$

où $p = q$ et chaque C'_i est une permutation d'un C''_j.

On pourra donc parler par la suite de la forme ou-exclusive normale réduite d'une formule propositionnnelle.

I.D - Montrer qu'une formule propositionnelle est une tautologie si et seulement si sa forme ou-exclusive normale réduite est **1**. Que peut-on en conclure quant à la complexité d'un algorithme permettant de convertir une formule propositionnelle sous forme normale ou-exclusive réduite ?

I.E - On appelle *formule de Horn élémentaire* toute disjonction de littéraux comprenant au plus un littéral positif et *formule de Horn* toute conjonction de formules de Horn élémentaires.

Montrer que toute formule de Horn A est équivalente à une conjonction de formules distinctes de la forme : $[p_i]$, ou $[\neg p_1,\ldots,\neg p_n]$ $(n \geq 1)$ ou $[\neg p_1, \ldots, \neg p_n, p_{n+1}]$ $(n \geq 1)$ où tous les p_i représentent des valeurs propositionnelles distinctes. On dit dans ce cas que A est réduite.

I.F - Soit $A = < E_1, \ldots, E_p >$ une formule de Horn réduite.

I.F.1) Montrer que dans chacun des deux cas suivants A est satisfiable :

1^{er} cas : aucune des formules de Horn élémentaire E_i n'est un littéral positif ;

2^{e} cas : toute formule de Horn élémentaire E_i contenant un littéral négatif contient aussi un littéral positif.

I.F.2) Dans le cas où A contient une formule de Horn élémentaire E_i réduite à un littéral positif p et une autre E_j contenant $\neg p$, en notant $D_{i,j}$ la formule de Horn élémentaire obtenue en éliminant $\neg p$ de E_j, montrer que A est satisfiable si et seulement si la formule A' obtenue en remplaçant dans A E_j par $D_{i,j}$ l'est et si $D_{i,j}$ est non vide.

I.F.3) En déduire que la satisfiabilité d'une formule de Horn réduite A peut être décidée en temps polynomial en fonction de la taille de A.

I.G - Soit la formule :

$$A = ((p \wedge q) \vee (r \Rightarrow \neg s)) \Rightarrow ((p \vee (r \Rightarrow \neg s)) \wedge (q \vee (r \Rightarrow \neg s)))$$

En utilisant la méthode de la question précédente appliquée à la formule $\neg A$, montrer que A est une tautologie, c'est à dire que $\neg A$ n'est pas satisfiable.

Exercice II -

On examine dans cet exercice le problème de l'allocation mémoire.

On peut voir la mémoire centrale d'un ordinateur comme un tableau (ou vecteur) `mem`, dont les éléments (qui représentent l'unité élémentaire de mémoire) sont appelés *mots mémoire* (ou *mots*). La taille de ce tableau est en général une puissance de 2, soit 2^N. Les indices de ce tableau (qui sont entiers compris entre 0 et $2^N - 1$) sont appelés *adresses*. Un mot mémoire est de taille suffisante pour stocker une adresse.

Tout programme exécuté sur l'ordinateur effectue des requêtes mémoire : soit en demandant l'allocation d'un bloc mémoire (une suite contigüe de mots) de taille donnée, soit en libérant un bloc mémoire. Ces allocations et libérations sont effectuées en très grand nombre, et de manière totalement imprévisible, car elles diffèrent à chaque exécution du programme, en fonction des données fournies.

Plus précisément, les fonctions d'allocation et de libération auront pour prototype :

en Pascal

```
function alloue (taille: integer): adresse;
procedure libere (ad:adresse; taille:integer);
```

en Caml

```
alloue:int—> adresse= <fun>
libere:adresse—> int—> unit= <fun>
```

Deux problèmes se posent immédiatement :

- trouver une structure de données adéquate pour représenter les blocs libres ;
- quel algorithme utiliser pour trouver un bloc de taille n et le réserver ?

On peut répondre à la première question en créant une liste pour représenter les blocs libres (on verra d'autres possibilités).

En Caml, on utilisera une liste, chaque élément de cette liste contenant la taille du bloc libre en nombre de mots et l'adresse de ce bloc.

En Pascal, on utilisera une liste chaînée, dont chaque maillon contiendra la taille du bloc libre en nombre de mots, l'adresse de ce bloc et un pointeur sur le maillon suivant.

Cette liste de blocs libres peut être triée (par adresses ou tailles) ou non. Initialement, la liste L (pour "libres") contient un seul bloc de taille 2^N et d'adresse 0. Cette liste pourra être considérée comme une variable globale.

Les fonctions (ou procédures) d'allocation devront modifier cette liste.

II.A - En supposant la liste des blocs non triée, écrire :

en Caml :

- la définition du type des éléments de la liste correspondant à la structure de liste nécessaire ;

en Pascal :

- la définition de type correspondant à la structure de liste nécessaire.

Écrire ensuite dans le langage choisi (Caml ou Pascal) :

- la fonction `alloue_premier` qui utilise le premier bloc dans la liste dont la taille est supérieure à la taille demandée ;
- La fonction `alloue_ meilleur` qui utilise un bloc dont la taille se rapproche le plus de la taille demandée ;

On se donne la situation de départ et les requêtes suivantes :

- situation de départ : 2 blocs libres de taille 768 et 512 ;
- requêtes consécutives : 500, 400, 300.

Que donne chacune des deux méthodes ? Laquelle est la plus intéressante ? Montrer par un exemple simple que l'on peut trouver une situation inverse.

II.B - On considère maintenant le problème de la libération d'un bloc alloué : il s'agit de remettre ce bloc dans la liste L, en le fusionnant le cas échéant avec d'éventuels blocs adjacents.

On suppose que la liste des blocs libres est triée par adresse croissante, et que l'algorithme d'allocation est celui donné par la fonction `alloue_premier`. Écrire la fonction `libere` qui va libérer un bloc de mémoire et l'ajouter dans la liste des blocs libres. On prendra garde à fusionner le bloc retourné avec d'éventuels blocs adjacents.

En réalité, la mémoire nécessaire pour gérer la liste L est également située dans la mémoire de l'ordinateur. Au lieu de créer une structure annexe pour la gestion des blocs libres, on stocke l'information nécessaire à cette gestion dans *les blocs eux-mêmes*.

Ainsi, pour allouer (ou spécifier) un bloc libre de taille utile p, on prendra un bloc de taille $p+c$, c étant une constante entière petite devant la taille des blocs réclamés (ce qui ne posera alors aucun problème), les mots supplémentaires servant à stocker de l'information.

Dans les deux questions suivantes, on prendra $c = 2$. Si `adr` est l'adresse d'un bloc de mémoire libre, le mot situé à l'adresse `adr` contient la taille p du bloc libre et le mot situé à l'adresse `adr` $+1$ contiendra l'adresse du bloc libre suivant, le bloc proprement dit étant formé des p mots suivants.

II.C - Modifier les fonctions `alloue_premier` et `libere` pour ne plus utiliser de structure de liste annexe, mais travailler directement dans le tableau `mem`.

Après une utilisation répétée de ces deux fonctions avec une situation de départ où toute la mémoire est libre, une nette tendance se dégage : les blocs de petite taille s'accumulent au début de la liste des blocs libres, si bien qu'il faut chercher assez loin dans la liste dès que l'on veut allouer un bloc de taille p pour p suffisamment grand.

II.D - Suggérer une modification simple de l'algorithme pour la fonction `alloue_premier` de telle sorte que :

- les blocs de petite taille ne s'accumulent pas en un point particulier de la liste ;
- la liste des blocs libres reste triée par adresse croissante dans le but d'utiliser la fonction `libere`.

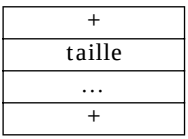
On utilisera une variable globale mémorisant une adresse et on écrira l'algorithme modifié.

Dans la suite, on réserve un peu de place supplémentaire pour ajouter les informations suivantes :

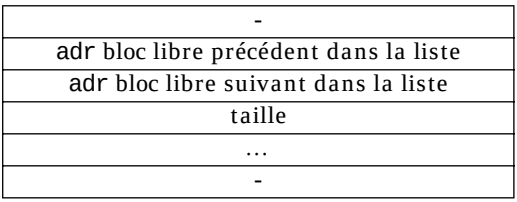
- les limites des blocs sont marquées : un bloc libre sera encadré par des '**-**'et un bloc occupé par des '+' ;
- tout bloc occupé contient également la taille de ce bloc ;
- la liste des blocs libres est *doublement chaînée*.

Cette nouvelle option est représentée par les schémas qui suivent.

On simule, dans le tableau, une structure de liste doublement chaînée pour les blocs libres de la façon suivante :



Bloc occupé



Bloc libre

II.E - Donner un nouvel algorithme plus efficace pour la fonction `libere` utilisant la structure de liste doublement chaînée et les marqueurs de début et de fin de bloc.

On peut envisager une troisième méthode pour allouer des blocs de mémoire. Cette méthode ne permet d'allouer que des blocs dont la taille est une puissance

de 2. Si l'on effectue une requête d'allocation pour un bloc de taille $n \neq 2^k$, alors un bloc de taille 2^k telle que $2^{k-1} < n \leq 2^k$ est alloué.

L'idée de cet algorithme repose sur la construction de listes de blocs de longueur 2^k libres pour chaque $0 \leq k \leq N$. Initialement, un bloc de taille 2^k est demandé, et si aucun bloc de cette taille n'est disponible, un bloc de taille plus importante est coupé en deux. Les deux blocs obtenus sont qualifiés de *jumeaux*. Finalement, un bloc de taille 2^k apparaît.

Chaque bloc mémorise l'information suivante :

- un champ booléen `res` indique si le bloc est réservé ou non ;
- les blocs *libres* possèdent deux liens : un lien `suiv` vers le bloc libre suivant, un lien `prec` vers le bloc libre précédent ;
- un champ `kval` dont la valeur indique que le bloc est de taille 2^{kval}.

D'autre part, on note : `libre`[0], `libre`[1], ..., `libre`[N] en Pascal (resp. `libre.`(0), `libre.`(1), ..., `libre.`(N) en Caml) les têtes de listes de blocs libres de taille $1, 2, 4, ..., 2^N$. Ces listes sont doublement chaînées et triées par adresses croissantes.

II.F - Montrer que si l'on connaît l'adresse d'un bloc, on peut trouver immédiatement l'adresse de son jumeau. En déduire l'intérêt de cette méthode.

II.G - Écrire la fonction `alloue` utilisant cette méthode.

II.H - Écrire la fonction `libere` utilisant cette méthode. Pourquoi la liste des blocs doit-elle être doublement chaînée ?

••• FIN •••