

COMPLEXITÉ

Master 1 IL
Groupe 2
2018

Rapport de TP N°6 COMPLEXITÉ : Le problème SAT

BOUDOUR Mehdi / 201500008386/ TP: Le problème SAT



**[ALGORITHMIQUE AVANCÉE
ET COMPLEXITÉ]**

Ce document présent les solutions en 5 étapes : (1) les algorithmes écrits en pseudo-code. (2) le calcul de la complexité au pire des cas. (3) Implémentation de l'algorithme en langage C. (4) capture de l'exécution de l'algorithme. (5) représentation graphique de l'évolution du temps d'exécution en fonction de N. Le programme C complet contenant les détails (affichage, calcul du temps d'exécution,...) d'implémentation est présenté à la fin du document.

I. Algorithme Résolution du problème SAT:

- Une proposition atomique est une variable booléenne, c'est-à-dire prenant ses valeurs dans l'ensemble $BOOL = \{VRAI, FAUX\}$.
- Un littéral est une proposition atomique ou la négation d'une proposition atomique.
- Une proposition atomique est aussi appelée littéral positif ; et la négation d'une proposition atomique est appelé littéral négatif.
- Une clause est une disjonction (somme logique ou (or)) de littéraux.

Etant données m propositions atomiques p_1, \dots, p_m , une instantiation du m -uplet (p_1, \dots, p_m) est un élément de $\{VRAI, FAUX\}$. Une instantiation (e_1, \dots, e_m) de (p_1, \dots, p_m) satisfait une clause c (notée $(e_1, \dots, e_m) c$) si et seulement si l'une des conditions suivantes est satisfaite :

- 1- il existe $i \in \{1, \dots, m\}$ tel que $(e_i = VRAI)$ et $(p_i$ occure ou existe dans $c)$; 2- il existe $i \in \{1, \dots, m\}$ tel que $(e_i = FAUX)$ et $(\neg p_i$ occure ou existe dans $c)$.

Une instantiation satisfait une conjonction (produit logique et (and)) de clauses si et seulement si elle satisfait chacune de ses clauses. Une conjonction de clauses est satisfiable si et seulement s'il existe une instantiation la satisfaisant. Une instantiation satisfaisant une conjonction est dite solution ou modèle de la conjonction.

Le problème SAT est maintenant défini comme suit :

- Entrée : une conjonction C de n clauses construites à l'aide de m propositions atomiques p_1, \dots, p_m ;
- Sortie : la conjonction C est-elle satisfiable ?

STRUCTURE PROPOSÉ : Matrice Binaire représentant une Conjonction.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

$$\Rightarrow M = (\neg a \vee b \vee c \vee \neg d) \wedge (\neg a \vee \neg b \vee c \vee d) \wedge (a \vee \neg b \vee \neg c \vee d)$$

Déclaration: structure Conjonction

```
TYPE ENREGISTREMENT CONJONCTION ;  
    MATRICE:TABLEAU [1..N] [1..M] D'ENTIER;  
    NBCLAUSE, NBLITERRAUX: ENTIER;  
FINENREG;
```

Implémentation : En langage C

```
//Type Matrice
typedef struct Matrice Matrice ;
struct Matrice
{
    long **Pointer;
    long n,m;
};
```

Déclaration: structure Instanciation (liste linéaire chaînée) #1^{ère}_Optimisation

```
TYPE ENREGISTREMENT INSTANCIATION
    VALEUR:ENTIER;
    SUIV: ^INSTANCIATION;
FINENREG;
```

Implémentation : En langage C

```
//Type Instanciation Liste Linéaire Chaînée
typedef struct Instanciation Instanciation ;
struct Instanciation
{
    long valeur;
    Instanciation *suiv;
};
```

Algorithme: *Primitive de Manipulation des Instanciations*

```
FONCTION AJOUTER(E/ I: ^INSTANCIATION, E/ VALEUR:ENTIER) :
^INSTANCIATION
    P: ^INSTANCIATION;
DEBUT
    ALLOUER (P) ;
    P->VALEUR=VALEUR; P->SUIV=I;
    RETOURNER P;
FIN;
```

Diagramme illustrant la complexité O(4) :

Les opérations `ALLOUER (P) ;` et `RETOURNER P;` sont regroupées par une accolade à droite, avec une flèche rouge pointant vers elles depuis la notation **O(4)** dans un rectangle jaune.

Implémentation : En langage C

```
//Primitives de Manipulation des Instanciations
Instanciation *Ajouter(Instanciation *I,long valeur)
{
    Instanciation *P=(Instanciation *)malloc(sizeof(Instanciation
));
```

```

P->valeur=valeur; P->suiv=I;
return P;
}

```

Algorithme : *VerifierClause* : Vérifie si une Clause est vrai pour une instance Donnée

```

FONCTION VERIFIERCLAUSE(E/ CLAUSE :TABLEAU[1..NBLITERRAUX] D'
ENTIER , E/ NBLITERRAUXINSTANCE:ENTIER):BOOLEEN
    J,BIT:ENTIER;
DEBUT
    BIT = INSTANCE;
    POUR J=NBLITERRAUX-1 JUSQU'A 0 PAS -1 FAIRE
        SI (CLAUSE[J] = BIT MOD 2) ALORS
            RETOURNER VRAI;
        FIN SI
        BIT=BIT/2;
    FIN POUR
    RETOURNER FAUX;
FIN;

```

$O(1+nbLiterraux)$

Implémentation : En langage C

```

//Verifier : Vérifie si une Clause est vrai pour une instance
Donnée
long VerifierClause(long *Clause,long nbLiterraux,long instance)
{
    long j,bit = instance;
    for(j=nbLiterraux-1;j>=0;j--)
    {
        if(Clause[j] == bit%2)
        {
            return 1;
        }
        bit=bit/2;
    }
    return 0;
}

```

Algorithme: Solutions : retourne les instanciations pour lesquelles une Clause est vrai parmi les instanciations données en paramètre

```

FONCTION SOLUTIONS(E/ CLAUSE : TABLEAU D' ENTIER,E/
NBLITERRAUX:ENTIER ,E/INSTANCES :
^INSTANCIATION):^INSTANCIATION
    I,BIT,J,TAILLETV:ENTIER;
    SOLUTIONS,P,Q: ^INSTANCIATION;
DEBUT
    SOLUTION = NULL;
    SI (INSTANCES=NULL) ALORS //PAS D'INSTANCES EN ENTREE
    //TESTER TOUTE LA TABLE DE VERITE 2^M
    TAILLETV = PUISSANCE(2,NBLITERRAUX);
    POUR I=0 JUSQU'A TAILLET FAIRE
        SI (VERIFIERCLAUSE(CLAUSE,NBLITERRAUX,I)=VRAI)
            ALORS
                SOLUTIONS = AJOUTER(SOLUTIONS,I);
            FIN SI;
        FIN POUR;
    SINON // INSTANCES != NULL IL YA DES VALEURS A TESTER
        //PARCOURS DE LA LISTE DES INSTANCIATIONS
        P=INSTANCE
        TANT QUE ( P <> NULL;) FAIRE
            SI (VERIFIERCLAUSE(CLAUSE,NBLITERRAUX,P->VALEUR) = VRAI)
                ALORS
                    SOLUTIONS = AJOUTER(SOLUTIONS,P->VALEUR);
                FIN SI;
            Q=P ; P=P->SUIV ; LIBERER(Q);
        FIN TANT QUE;
    FIN SI;
    //RETOURNER LA LISTE DES INSTANCES POUR LESQUELLE CLAUSE EST VRAI
    RETOURNER SOLUTIONS;
FIN;

```

Complexity Analysis:

- Overall Complexity:** $C(\text{Solution}) = O(m \cdot 2^m)$
- First Loop (I=0 to TAILLET):**
 - Inside the loop, the complexity is $O(1 + \text{nbLiterraux})$ for the **VERIFIERCLAUSE** call.
 - The loop itself has a complexity of $O(4)$.
 - Total complexity for this loop: $O(2^{\text{nbLiterraux}} \cdot (5 + \text{nbLiterraux}))$.
- Second Loop (P=INSTANCE to NULL):**
 - Inside the loop, the complexity is $O(1 + \text{nbLiterraux})$ for the **VERIFIERCLAUSE** call.
 - The loop itself has a complexity of $O(4)$.
 - Total complexity for this loop: $O(2^{\text{nbLiterraux}} \cdot (5 + \text{nbLiterraux}))$.

Implémentation : En langage C

```

//Solutions : retourne les instanciations pour lesquelles une
Clause est vrai
//Parmi les instanciations données en paramètre
Instanciation *Solutions(long *Clause,long nbLiterraux
,Instanciation *Instances)
{
    long i,bit, j;
    Instanciation *solutions=NULL ,*P,*Q;
    if(Instances==NULL) //Pas d'instances en Entrée
    {
        //Tester pour toute la Table de Vrité 2^m
        long tailleTV = (long)pow(2,nbLiterraux);
    }
}

```

```

    for(i=0 ; i<tailleTV ; i++)
    {
        if(VerifierClause(Clause,nbLitteraux,i))
        {solutions = Ajouter(solutions,i); }
        Affichage(solutions);
    }

}
else // Instances != NULL il ya des valeurs ⚡ Tester
{ //Parcours de la Liste des instances
  for(P=Instances; P != NULL;)
  {
      if(VerifierClause(Clause,nbLitteraux,P->valeur))
      { solutions = Ajouter(solutions,P->valeur); }

      Q=P ; P=P->suiv ; free(Q); //Libérer la cellule lu de
Instances
  }
}
//retourner la Liste des instances pour laquelle Clause est
VRAI
return solutions;
}

```

Algorithme: *Validation Version 1*

```

FONCTION VALIDATION1 (E/ CONJ :CONJONCTION,INT VALEUR) :BOOLEEN
  I:ENTIER;
  P, INSTANCES:^INSTANCIATION;
DEBUT
  P=NULL;
  INSTANCES = AJOUTER(P,VALEUR);
  SI (INSTANCES=NULL) ALORS
    RETOURNER 0;
  SINON
    POUR I=1 JUSQU'A CONJ.NBCLAUSES FAIRE
      INSTANCES =
      SOLUTIONS(CONJ.MATRICE[I],CONJ.NBLITTERAUX,INSTANCES);
      SI (INSTANCES=NULL) ALORS
        RETOURNER FAUX;
      FIN SI;
    FIN POUR;
  FIN SI;
  RETOURNER VRAI;

```

Complexity analysis annotations:

- $O(4)$ points to the **AJOUTER** call.
- $O(nbLitteraux)$ points to the **SOLUTIONS** call.
- $O(nbClauses * nbLitteraux)$ points to the **POUR** loop.

FIN;

Complexité :

Au pire des cas : (INSTANCES <> NULL)

$$\begin{aligned} C(\text{Version1}) &= \sum_{i=1}^{\text{NbClause}} \text{nbLitteraux} \\ &= (\text{NbClause} - 1 + 1) * \text{nbLitteraux} \\ &= \text{nbClauses} * \text{nbLitteraux} \sim \mathbf{O(n*m)} \text{ Polynomiale} \end{aligned}$$

Au meilleur des cas : (INSTANCES = NULL)

$$C(\text{Version1}) = C(\text{Ajouter}) = 4 \sim \mathbf{\Omega(1)}$$

Implémentation : En langage C


```
long Validation1(long **Conjonction ,long nbClauses, long
nbLitteraux,long valeur)
{
    long i;
    Instanciation *P; P=NULL;
    Instanciation *instances = Ajouter(P,valeur);
    if(instances==NULL) return 0;
    else
        for(i=1;i<nbClauses;i++)
        {
            instances =
Solutions(Conjonction[i],nbLitteraux,instances);
            if(instances==NULL) return 0;
        }
    return 1;
}
```

Exécution :

Cet Algorithme est si rapide qu'il a été difficile de relever des valeurs de temps significatives.

Algorithme : Validation Version 2

```
FONCTION VALIDATION2 (1 (E/ CONJ : CONJONCTION) : ENTIER
    I : ENTIER;
    INSTANCES : ^INSTANCIATION
DEBUT
```



$\mathbf{O(2^{nbLitteraux})}$


```

INSTANCES = SOLUTIONS (CONJ.MATRICE[0], CONJ.NBLITTERAUX, NULL);
SI (INSTANCES=NULL) ALORS
    RETOURNER 0;
SINON
    POUR I=1 JUSQU'A CONJ.NBCLAUSES FAIRE
        INSTANCES = SOLUTIONS (CONJ.MATRICE[I], CONJ.NBLITTERAUX, INSTANCES);
        SI (INSTANCES=NULL) ALORS
            RETOURNER FAUX;
        FIN SI;
    FIN POUR;
FIN SI;
RETOURNER VRAI;
FIN;

```

Complexity analysis for the nested loop:

$$O(\text{nbLitteraux} * 2^{\text{nbLitteraux}})$$

Complexité :

Au pire des cas : (INSTANCES <> NULL)

$$\begin{aligned}
 C(\text{Version2}) &= \sum_{i=1}^{\text{NbClause}} \text{nbLitteraux} * 2^{\text{nbLitteraux}} \\
 &= (\text{NbClause} - 1 + 1) * \text{nbLitteraux} * 2^{\text{nbLitteraux}} \\
 &= \text{nbClauses} * \text{nbLitteraux} * 2^{\text{nbLitteraux}} \\
 &\sim O(n * m * 2^m) \text{ Exponentielle}
 \end{aligned}$$

Au meilleur des cas : (INSTANCES = NULL)

$$C(\text{Version2}) = C(\text{Solutions}) = 2^{\text{nbLitteraux}} \sim \Omega(2^m)$$

Implémentation : En langage C

```

long Validation2(long **Conjonction ,long nbClauses, long
nbLitteraux)
{
    long i;
    Instanciation *instances =
Solutions(Conjonction[0],nbLitteraux,NULL);
    if(instances==NULL) return 0;
    else
        for(i=1;i<nbClauses;i++)
        {
            instances =
Solutions(Conjonction[i],nbLitteraux,instances);
            if(instances==NULL) return 0;
        }
}

```

```
return 1;
```

```
}
```

Exécution :

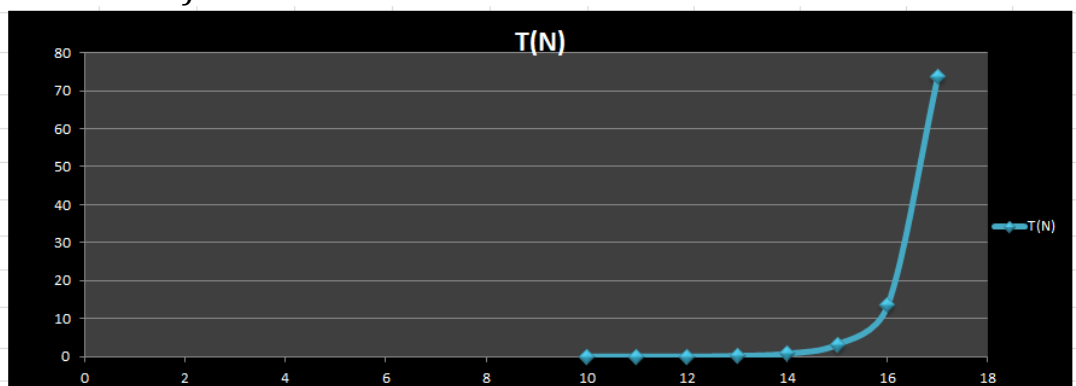
Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction **Validation2**).

```
Execution de Validation2 :  
N = 10.000000    T= 0.004000  
N = 11.000000    T= 0.016000  
N = 12.000000    T= 0.045000  
N = 13.000000    T= 0.185000  
N = 14.000000    T= 0.801000  
N = 15.000000    T= 3.231000  
N = 16.000000    T= 13.429000  
N = 17.000000    T= 73.669000
```

Représentation Graphique :

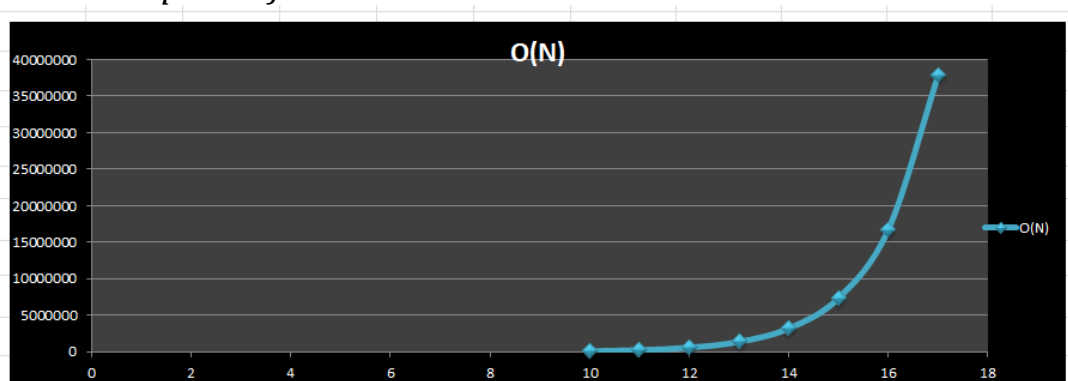
Graphe du temps d'exécution en fonction de N .

Validation2	
$N=M$	$T(N)$
10	0,004
11	0,016
12	0,045
13	0,185
14	0,801
15	3,231
16	13,429
17	73,669



Graphe de la complexité théorique en fonction de N .

Validation2	
$N=M$	$O(N)$
10	102400
11	247808
12	589824
13	1384448
14	3211264
15	7372800
16	16777216
17	37879808



Algorithme : Validation Version 3

```
FONCTION VALIDATION3 (E/ CONJ:CONJONCTION) : ^INSTANCIATION  
I:ENTIER;
```

```

INSTANCES : ^INSTANCIATION
DEBUT
    INSTANCES = SOLUTIONS (CONJ.MATRICE[0], CONJ.NBLITERRAUX, NULL);
    SI (INSTANCES=NULL) ALORS
        RETOURNER NULL;
    SINON
        POUR I=1 JUSQU'A CONJ.NBCLAUSES FAIRE
            INSTANCES =
                SOLUTIONS (CONJ.MATRICE[I], CONJ.NBLITERRAUX, INSTANCES);
                SI (INSTANCES=NULL) ALORS
                    RETOURNER NULL;
                FIN SI;
            FIN POUR ;
        FIN SI;
    RETOURNER INSTANCES;
FIN;

```

Complexity annotations:

- $O(2^{nbLitteraux})$ points to the initial call to SOLUTIONS.
- $O(nbLitteraux * 2^{nbLitteraux})$ points to the recursive call to SOLUTIONS.
- A bracket on the right indicates the total complexity is $O(nbLitteraux * 2^{nbLitteraux})$.

Complexité : De même que la version 2

Au pire des cas : (INSTANCES <> NULL)

$$C(\text{Version3}) = \sim O(n * m * 2^m) \text{ Exponentielle}$$

Au meilleur des cas : (INSTANCES = NULL)

$$C(\text{Version3}) = C(\text{Solutions}) \sim \Omega(2^m)$$

Implémentation : En langage C

```

Instanciation *Validation3(long **Conjonction, long nbClauses, long
nbLitteraux)
{
    long i;
    Instanciation *instances =
    Solutions(Conjonction[0], nbLitteraux, NULL);
    if(instances==NULL) return NULL;
    else
        for(i=1; i<nbClauses; i++)
        {
            instances =
            Solutions(Conjonction[i], nbLitteraux, instances);
            if(instances==NULL) return NULL;
        }
    return instances;
}

```

Exécution :

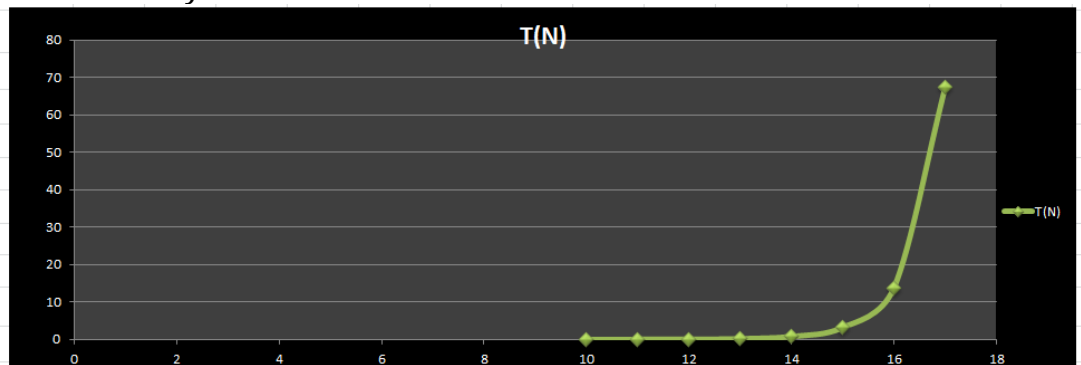
Affichage du temps d'exécution de l'algorithme pour chaque valeur de N (T = le temps d'exécution calculé pour chaque exécution de la fonction **Validation3**).

```
Execution de Validation3 :
N = 10.000000    T= 0.003000
N = 11.000000    T= 0.015000
N = 12.000000    T= 0.046000
N = 13.000000    T= 0.182000
N = 14.000000    T= 0.789000
N = 15.000000    T= 3.206000
N = 16.000000    T= 13.812000
N = 17.000000    T= 67.371000
```

Représentation Graphique :

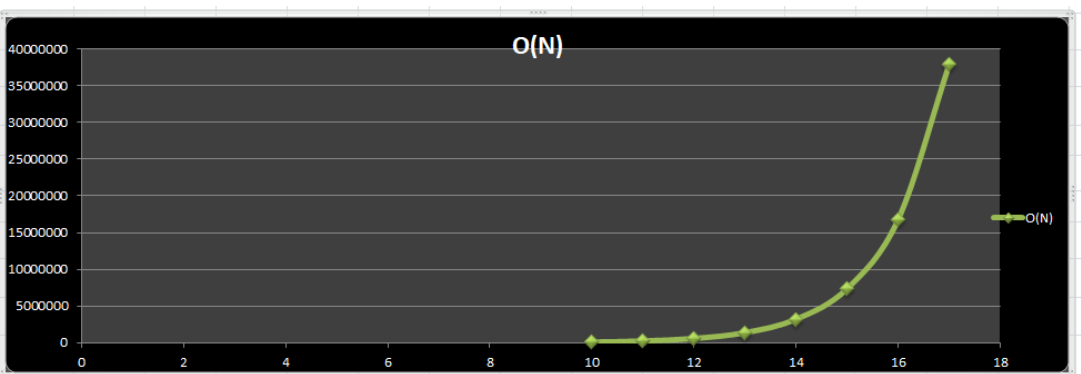
Graphes du temps d'exécution en fonction de N .

Validation3	
$N=M$	$T(N)$
10	0,003
11	0,015
12	0,046
13	0,182
14	0,789
15	3,206
16	13,812
17	67,371



Graphes de la complexité théorique en fonction de N .

Validation3	
$N=M$	$O(N)$
10	102400
11	247808
12	589824
13	1384448
14	3211264
15	7372800
16	16777216
17	37879808



(*)Code Source du Programme complet :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

//Type Matrice
typedef struct Matrice Matrice ;
struct Matrice
{
    long **Pointer;
```

```

    long n,m;
};

//Type Instanciation Liste Linéaire Chainée
typedef struct Instanciation Instanciation ;
struct Instanciation
{
    long valeur;
    Instanciation *suiv;
};

void Affichage(Instanciation *tete)
{
    Instanciation *P;
    long i;
    if(tete==NULL)
    {
        //printf("LA LISTE EST VIDE !");
    }
    else
    {
        P=tete;
        //printf("[");
        while(P!=NULL)
        {
            //printf("%d,",P->valeur);
            P=P->suiv;
        }
        //printf("end]\n");
    }
}

void Affiche(long T[],long N)
{
    long i=0;
    //printf("[");
    for(i=0;i<N-1;i++)
        //printf("%d,",T[i]);
    //printf("%d]\n",T[N-1]);
    ;
}

//Primitives de Manipulation des Instanciations
Instanciation *Ajouter(Instanciation *I,long valeur) //O(4)

```

```

{
    Instanciation *P=(Instanciation *)malloc(sizeof(Instanciation
));
    P->valeur=valeur; P->suiv=I;
    return P;
}

//Verifier : Verifie si une Clause est vrai pour une instance
Donne
long VerifierClause(long *Clause,long nbLitteraux,long instance) //
O(1+nbLitteraux)
{
    //printf("\n\t\tVérification : Clause =
");Affiche(Clause,nbLitteraux);
    //printf("\n\t\tpour instantiation = %d.",instance);

    long j,bit = instance;
    for(j=nbLitteraux-1;j>=0;j--)
    {
        if(Clause[j] == bit%2)
        {
            //printf("\n\t\t\tVRAI pour instantiation =
%d.",instance);
            return 1;
        }
        bit=bit/2;
    }
    //printf("\n\t\t\tFAUSSE pour instantiation = %d.",instance);
    return 0;
}

//Solutions : retourne les instantiations pour lesquelles une
Clause est vrai
//Parmi les instantiations donnees en parametre
Instanciation *Solutions(long *Clause,long nbLitteraux
,Instanciation *Instances) // O(m*2^m)
{
    //printf("\n\tRecherche de Solutions: pour Clause =
");Affiche(Clause,nbLitteraux);
    //printf("\n\tparmi les instantiations = ");
    Affichage(Instances);

    long i,bit, j;
    Instanciation *solutions =NULL ,*P,*Q;
    if(Instances==NULL) //Pas d'instances en Entrée

```

```

{ //Tester pour toute la Table de V?rit? 2^m
    long tailleTV = (long)pow(2,nbLitteraux);
    for(i=0 ; i<tailleTV ; i++) //O(2^nbLitteraux *
(5+nbLitteraux))
    {
        if(VerifierClause(Clause,nbLitteraux,i))//O(1+nbLittera
ux)

        {solutions = Ajouter(solutions,i); }// O(4)
        Affichage(solutions);
    }

}
else // Instances != NULL il ya des valeurs ? Tester
{ //Parcours de la Liste des instantiations
    for(P=Instances; P != NULL;) //au pire cas O(2^nbLitteraux
* (5+nbLitteraux))
    {
        if(VerifierClause(Clause,nbLitteraux,P->valeur))
//O(1+nbLitteraux)
        { solutions = Ajouter(solutions,P->valeur); }// O(4)

        Q=P ; P=P->suiv ; free(Q); //Lib?rer la cellule lu de
Instances
    }
}
//retourner la Liste des instances pour laquelle Clause est
VRAI
    //printf("\n\tLes Solutions trouvees =
");Affichage(solutions);
    return solutions;
}

long Validation1(long **Conjonction ,long nbClauses, long
nbLitteraux,long valeur) //O(n*m) - Omega(1)
{
    long i;
    Instanciation *P; P=NULL;
    Instanciation *instances = Ajouter(P,valeur);
    if(instances==NULL) return 0;
    else
        for(i=1;i<nbClauses;i++)
//O(nbClauses*nbLitteraux*2^nbLitteraux)
        {
            instances =
Solutions(Conjonction[i],nbLitteraux,instances); //O(nbLitteraux)

```

```

        if(instances==NULL) return 0;
    }
    return 1;
}

long Validation2(long **Conjonction ,long nbClauses, long
nbLitteraux) //O(n*m*2^m) - Omega(2^m)
{
    long i;
    Instanciation *instances =
Solutions(Conjonction[0],nbLitteraux,NULL);
    if(instances==NULL) return 0;
    else
        for(i=1;i<nbClauses;i++)
//O(nbClauses*nbLitteraux*2^nbLitteraux)
        {
            instances =
Solutions(Conjonction[i],nbLitteraux,instances);
//O(nbLitteraux*2^nbLitteraux)
            if(instances==NULL) return 0;
        }
    return 1;
}

Instanciation *Validation3(long **Conjonction ,long nbClauses, long
nbLitteraux) //O(n*m*2^m) - Omega(2^m)
{
    long i;
    Instanciation *instances =
Solutions(Conjonction[0],nbLitteraux,NULL);
    if(instances==NULL) return NULL;
    else
        for(i=1;i<nbClauses;i++)
//O(nbClauses*nbLitteraux*2^nbLitteraux)
        {
            instances =
Solutions(Conjonction[i],nbLitteraux,instances);
//O(nbLitteraux*2^nbLitteraux)
            if(instances==NULL) return NULL;
        }
        //printf("\nLa Solution Finale =
");Affichage(instances);
    return instances;
}

```



```

//Matrice Trii;½ Ordre di½croissant
long **Conjonction(long n)
{
    long i,j,**T=(long **)malloc(n*sizeof(long *));
    for(i=0;i<n;i++)
    {
        T[i] = (long *)malloc(n*sizeof(long ));
        for(j=0;j<n;j++)
        {T[i][j]= rand() % 2;}
    }
    return T;
}

double **Calcul_des_Temps(double **tab , long algorithme)
{
    Instanciation *P=NULL;
    long j,verdict,**M;
    for(j=0 ; j<8 ; j++)
    {
        long **Conj = Conjonction((long)tab[0][j]);
        clock_t begin = clock();
        switch(algorithme)
        {
            case 1: verdict =
Validation1(Conj,(long)tab[0][j],(long)tab[0][j],15); break;
            case 2: verdict =
Validation2(Conj,(long)tab[0][j],(long)tab[0][j]); break;
            case 3: P =
Validation3(Conj,(long)tab[0][j],(long)tab[0][j]); break;
        }
        clock_t end = clock();
        tab[1][j] = (double)(end - begin) / CLOCKS_PER_SEC;
        if(P!=NULL)
            tab[2][j] = 1;
        else tab[2][j]=verdict;
    }
    return tab;
}

double **Tableau_de_Valeurs23(void)
{
    long i ;
    double **tab;

```

```

    tab = (double **)malloc(4*sizeof(double *));
    for(i=0 ; i<3 ; i++) tab[i] = (double
*)malloc(8*sizeof(double));
    tab[0][0]=10;
    tab[0][1]=11;
    tab[0][2]=12;
    tab[0][3]=13;
    tab[0][4]=14;
    tab[0][5]=15;
    tab[0][6]=16;
    tab[0][7]=17;
    for(i=0 ; i<8 ; i++)tab[1][i] = 0 ;
    return tab;
}

void Afficher_Tableau_de_Valeurs(double **tab)
{
    long j,verdict;
    for(j=0 ; j<10 ; j++)
    {
        verdict = (int)tab[2][j];
        printf("N = %f \t T= %f \t , Satisfiable = %d
\n",tab[0][j],tab[1][j],tab[2][j]);
    }
}

void AfficherMatrice(long **M,long n,long m)
{
    long i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
            printf("%d\t",M[i][j]);
        printf("\n");
    }
}

int **MatrixToPointer(int *M,int rows ,int cols)
{
    int i,j,**R=(int **)malloc(rows*sizeof(int *));
    for (i = 0; i < rows; i++) {
        *(R+i)=(int *)malloc(cols*sizeof(int ));
        for (j = 0; j < cols; j++) {

```

```

        R[i][j]= *(M + i * cols + j);
    }
}
return R;
}
/*
Matrice LectureConjonction(void)
{
    int n,m,i,j;
    printf("Lecture de la Conjonction:\n");
    printf("Entrez le nombre de clauses : ");
    scanf("%d",&n); getchar();
    printf("Entrez le nombre de litteraux des clauses : ");
    scanf("%d",&m); getchar();
    int **Conjonction=(int **)malloc(n*sizeof(int *));
    for(i=0;i<n;i++) Conjonction[i]=(int *)malloc(m*sizeof(int));
    printf("Remplir la matrice :\n");
    for(i=0;i<n;i++)
    {
        printf("Saisie de la clause %d :\n",(i+1));
        for(j=0;j<m;j++)
        {
            printf("\t%c = ",(j+97));
            scanf("%d",&(Conjonction[i][j]));
            getchar();
        }
    }
    Matrice Mat;
    Mat.m=m;
    Mat.n=n;
    Mat.Pointer=Conjonction;
    return Mat;
}
*/
int main(int argc, char *argv[])
{

    printf("Execution de Validation2 :\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
23(),2));

    printf("Execution de Validation3 :\n");
    Afficher_Tableau_de_Valeurs(Calcul_des_Temps(Tableau_de_Valeurs
23(),3));

```

```

/*  int C[4][4] ={
        {1 ,0 ,1 ,0 },
        {0 ,0 ,0 ,1 },
        {1 ,1 ,0 ,0 },
        {0 ,0 ,1 ,1 }
    };

    //Affichage(Validation(MatrixToPointer(&C[0][0],4,4),4,4));
    //Version 1
    printf("Version 1 :\n");
    Matrice Conj = LectureConjonction();
    printf("\nLa conjontion = \n");
    AfficherMatrice(Conj.Pointer,Conj.n,Conj.m);
    printf("\nEntrez l'instanciation : ");
    int instance ; scanf("%d",&instance); getchar();
    if(Validation1(Conj.Pointer,Conj.n,Conj.m,instance))
printf("\nLinstance %d est Solution.",instance);
    else printf("\nLinstance %d n'est pas Solution.",instance);

    //Version 1
    printf("\nVersion 2 :\n");
    Conj = LectureConjonction();
    printf("\nLa conjontion = \n");
    AfficherMatrice(Conj.Pointer,Conj.n,Conj.m);
    if(Validation2(Conj.Pointer,Conj.n,Conj.m)) printf("\nLa
Conjonction est Satisfiable.");
    else printf("\nLa Conjonction n'est pas Satisfiable.");

    //Version3
    printf("\nVersion 2 :\n");
    Conj = LectureConjonction();
    printf("\nLa conjontion = \n");
    AfficherMatrice(Conj.Pointer,Conj.n,Conj.m);
    Affichage(Validation3(Conj.Pointer,Conj.n,Conj.m));
*/
    getchar();
    return 0;
}

```