

**Expose ALGORITHME AVANCE**

**PRESENTATION D'UN ALGORITHME POUR CREER UN ENSEMBLE DE PRODUIT CARTESIEN**

**Présenté par** : *Mr RAHOVANIRINA Faniriantsoa*

*Mr RANDRENARIZO Tsiry*

*Mr RANDRIAMANANTENA Julio Tolojanahary*

**Mention :** *Informatique*

**Parcours :** *Informatique et* *Programmation*

**Année universitaire :** *2014/2015*

# Introduction :

Jusqu'à maintenant, plusieurs codes sources d'application sont basés sur l'algorithme avancé. Il nous permet d'utiliser un ou plusieurs types d'arbre pour résoudre rapidement certains genres de problèmes.

En mathématiques, il est très important de savoir comment produire des ensembles? Ce genre de produit nous permet de résoudre facilement différents types de problème. Mais entant qu'informaticiens, nous pouvons créer une application qui peut résoudre ce problème du produit cartésien. C'est pour cela que notre thème est basé sur la **"Présentation d'un algorithme pour créer un ensemble de produit cartésien"**.

Nous allons présenter alors un code source en c++ qui peut produire plusieurs ensembles. Mais pour que vous compreniez bien notre raisonnement, notre explication va se diviser en deux parties, la première c'est comment résoudre manuellement un produit cartésien?, dans la deuxième partie nous allons voir la présentation des codes sources d'une application qui répond ce problème du produit cartésien, et c'est dans la dernière partie qu'on va montrer le résultat obtenu après l'exécution de ce programme.

# Comment résoudre manuellement le produit cartésien?

Etant donné deux ensembles E et F; on appelle produit cartésien de E et F l’ensemble : E × F = {(x, y) : x ∈ E et y ∈ F }. Dans ce cas là, Si le problème se pose comme suit : Effectuer le produit cartésien d'un ensemble E par un ensemble F, noté E x F. Cela consiste à distribuer tous les éléments de E par rapport à chaque élément de F.

**Exemples**

**Exemple 1 :**

E = {2, 3} et F = {a, b}

Le produit de E par F donne le résultat suivant :

E x F = {(2, a), (2, b), (3, a), (3, b) }

**Exemple 2 :**

G = {As, Roi}

H = C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image017.gif

Le produit cartésien de G par H est :

G x H = { (As, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image019.gif ), (As, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image021.gif ), (As, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image023.gif ), (As, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image026.gif), (Roi, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image019.gif ), (Roi, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image021.gif), (Roi, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image023.gif ), (Roi, C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image026.gif ) }

Il faut noter bien que Dans tous les cas, le produit de E, comportant k éléments, par F, en comportant h, engendre un ensemble de k.h éléments ou k.h couples (des couples parce que le nombre d'ensembles à produire ici vaut 2 notamment E et F). Dans nos exemples :

E x F doit contenir 2 x 2 = 4 couples et G x H doit avoir 2 x 4 = 8 couples.

Quelques propriétés importantes à retenir pendant la résolution d'un produit cartésien :

* Dans un produit cartésien E1 x E2 x . . . x En, les n-uplets (triplets, quadruplets, quintuplets, sextuplets, heptuplets, octuplets, nonuplet, décuplets . . .) (x1, x2, ..., xn) formés sont ordonnés, et ils sont mentionnés entre parenthèses. Le produit fixe l'ordre des éléments de chaque n-uplets(n-uplets car on a n ensembles à produire). L'élément x1 provient de E1, le deuxième provient de E2, ... et le dernier provient de En.
* Le produit cartésien n'est pas commutatif. C'est à dire : E X F est différent de F X E.
* Le produit cartésien avec un ensemble vide est nul : E x C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image030.gif = C:\Users\Rapa\Downloads\ProduitCartesien\Théorie des ensembles, produit cartésien_files\image030.gif
* Le produit d'un ensemble infini par un ensemble infini est infini. Exemples : N x N = N² ensemble de tous les couples d'entiers positifs ou nuls, R x R = R² ensemble de tous les couples de nombres réels.
* Le produit cartésien de trois ensembles engendre des triplets.  
  Même histoire pour quatre ou plus; c'est à dire que le produit des n ensembles doit engendrer n-uplets d'élément provenant de chaque ensemble.

# Présentation des codes sources d'une application qui répond ce problème de produit cartésien :

Avant toute chose, notre application va contenir deux types de classe. La première qui s'appelle "Ensemble" va contenir toutes les valeurs de chaque ensemble. L'autre qui porte le nom "Arbre" va contenir toutes les ensembles à produire. Cette application contient l'ensemble des fichiers ci-après :

## Contenues du fichier "Ensemble.h" :

#ifndef ENSEMBLE\_H

#define ENSEMBLE\_H

#include <deque>

#include <iostream>

#include <stdio.h>

using namespace std;

class Ensemble

{

public :

Ensemble();

~Ensemble();

void ajouter(int e);

void ajouterAuDebut(int e);

void afficher();

void produireAvec(Ensemble e2);

//void set(int i);

int at(int i);

int size();

protected:

private:

//vecteur ou Tableau des Valeurs (éléments) de l'Ensemble

deque <int> m\_TVE;

};

#endif //ENSEMBLE\_H

## Contenues du fichier " Ensemble.cpp" :

#include "Ensemble.h"

Ensemble::Ensemble()

{

}

void Ensemble::ajouter(int e)

{

//Ajout de l'élément e(paramètre) dans le tableau des valeurs de l'ensemble

m\_TVE.push\_back(e);

}

void Ensemble::ajouterAuDebut(int e)

{

//Ajout de l'élément e au début du tableau des valeurs de l'ensemble

m\_TVE.push\_front(e);

}

void Ensemble::afficher()

{

//Affichage de tous les elements de l'ensemble

for(int c=0;c<(int)m\_TVE.size();c++)

{

if(c<(int)m\_TVE.size()-1)

cout << m\_TVE.at(c) << " , ";

else

cout << m\_TVE.at(c);

}

cout << endl;

}

void Ensemble::produireAvec(Ensemble e2)

{

//distribution de tous les éléments de l'ensemble à prosuire par rapport à //chaque élément de e2

for(int c=0;c<(int) m\_TVE.size();c++)

{

for(int d=0;d<(int) e2.m\_TVE.size();d++)

cout << m\_TVE.at(c) << " , " << e2.m\_TVE.at(d) << "," ;

}

}

int Ensemble::at(int i)

{

//retourne la valeur de l'élément au i-ème position de l'ensemble

return m\_TVE.at(i);

}

int Ensemble::size()

{

//retourne la taille de l'ensemble

return m\_TVE.size();

}

Ensemble::~Ensemble()

{

}

## Contenues du fichier " Arbre.h" :

#ifndef ARBRE\_H

#define ARBRE\_H

#include "Ensemble.h"

#include <vector>

#include <math.h>

class Arbre

{

public :

Arbre();

~Arbre();

void ajouter(Ensemble e);//pour ajouter un ensemble dans l'arbre

void afficher();

void produire();

bool getFilsExiste(int niveau);

int getNbFils(int niveau);

void afficherEtGetFils(int niveau,int numero);

protected:

private:

vector <Ensemble> m\_TE;

vector <Ensemble> m\_TEnsembreFinaux;

vector <int> m\_TFinal;

int m\_numEnsembleActuel;

int m\_nbEnsembleFInaux;

};

#endif //ARBRE\_H

## Contenues du fichier " Arbre.cpp" :

#include "Arbre.h"

Arbre::Arbre()

{

m\_numEnsembleActuel=0;

}

void Arbre::ajouter(Ensemble e)

{

//Ajout de l'ensemble e dans l'arbre

m\_TE.push\_back(e);

}

void Arbre::afficher()

{

//Affichage de tous les ensembles existant dans l'arbre

//C'est la fonction afficher de la classe Ensemble que nous allons appeler ici

for(int c=0;c<(int) m\_TE.size();c++)

{

cout<<"E"<<c<<" = ";

m\_TE.at(c).afficher();

cout<< endl;

}

}

void Arbre::afficherEtGetFils(int niveau,int numero)

{

//m\_numEnsembleActuel nandray valeur 0 tany @ constructeur par defaut de l'Arbre

//Ajouter l'élément qui se trouve au niveau et au numero de l'ensemble m\_TE dans le tableau d'ensemble finaux

//m\_TEnsembreFinaux qui va contenir le résultat du produit cartésien final

m\_TEnsembreFinaux.at(m\_numEnsembleActuel).ajouter((m\_TE.at(niveau).at(numero)));

//Si ce niveau a des fils

if(getFilsExiste(niveau))

{

//Répéter l'ajout au déssus, jusqu'au nombre de fils du niveau actuel

for(int c=0;c<getNbFils(niveau);c++)

afficherEtGetFils(niveau+1,c);

}

else//S'il n'y a pas de fils dans le niveau

{

//======================="pas de fils"=========================

m\_numEnsembleActuel++;

}

/\*En résumer ce fonction va tester l'élément du niveau s'il a des fils. Au cas où ce teste est vrai il va ajouter tous les

premiers fils après son père. Dans ce cas là, il va ajouter un à un aussi tous les derniers fils dans le m\_TEnsembreFinaux

avec des m\_numEnsembleActuel différents. C'est après cela que ce même teste va passer au père suivant des derniers fils et

refaire la même action. Si tous les pères des dérniers fils ont tous déjà testé, le teste va passer au père suivant des

pères des derniers fils et refaire les mêmes actions. Ce teste se teminera au dernier élément du plus petit niveau. \*/

}

void Arbre::produire()

{

//récupération du nombre de combinaison possible

m\_nbEnsembleFInaux=1;

for(int c=0;c<(int)m\_TE.size();c++)

m\_nbEnsembleFInaux\*=m\_TE.at(c).size();

//==============================================

for(int c=0;c<=m\_nbEnsembleFInaux;c++)

{

//On va insérer des ensembles vides dans l'ensemble finaux nommé m\_TEnsembreFinaux

Ensemble et;

m\_TEnsembreFinaux.push\_back(et);

}

for(int c=0;c<(int) m\_TE.at(0).size();c++)

{

afficherEtGetFils(0,c);

}

//=============================================================

//Maintenant on va tester si tous les éléments de chaque niveau sont complets

//Au cas où le teste est vrai, on doit les completer pour atteindre le résultat finale

int occurence=1;

for(int c=1;c<(int)m\_TE.size();c++)

occurence\*=m\_TE.at(c).size();

for(int d=0;d<m\_nbEnsembleFInaux;d+=occurence)

{

for(int c=d+1;c<d+occurence;c++)

{

//Si la taille de l'ensemble du niveau c est différente à celle de son père

if(m\_TEnsembreFinaux.at(c).size()!=m\_TEnsembreFinaux.at(c-1).size())

{

//On doit calculer la différence. Le résultat va être le nombre d'éléments manquants à ce niveau c

int difference= m\_TEnsembreFinaux.at(c-1).size()-m\_TEnsembreFinaux.at(c).size();

//Il faut ajouter maintenant tous ces éléments manquants au niveau c

for(int e=difference-1;e>=0;e--)

{

m\_TEnsembreFinaux.at(c).ajouterAuDebut(m\_TEnsembreFinaux.at(c-1).at(e));

}

}

}

}

//=============================================================

//Affiche chaque element de l'ensemble finaux

for(int c=0;c<=m\_nbEnsembleFInaux;c++)

m\_TEnsembreFinaux.at(c).afficher();

}

int Arbre::getNbFils(int niveau)

{

return m\_TE.at(niveau+1).size();

}

bool Arbre::getFilsExiste(int niveau)

{

bool existe;

existe=false;

if(niveau<(int) m\_TE.size()-1)

existe=true;

return existe;

}

Arbre::~Arbre()

{

}

## Contenues du fichier " main.cpp" :

#include "Ensemble.h"

#include "Arbre.h"

#include <iostream>

using namespace std;

int main()

{

Ensemble \*e;

int nbrEnsblAProduire;

int valElmnt, iNbrElmt, c, d;

cout<<"Saisir le nombre d'ensemble a produire : ";

cin>>nbrEnsblAProduire;

e=new Ensemble[nbrEnsblAProduire];

//On va enregistrer tous les elements de chaque ensemble

for(c=0; c<nbrEnsblAProduire; c++){

cout<<"Saisir le nombre d'elements de l'ensemble E"<<c<<" : ";

cin>>iNbrElmt;

for(d=0; d<iNbrElmt; d++){

cout<<"x"<<d<<" de E"<<c<<" = ";

cin>>valElmnt;

e[c].ajouter(valElmnt);

}

}

//Declaration de l'arbre qui va contenir tous les ensembles dont nous avons //déclaré au dessus

Arbre a0;

//Ajout de tous les ensembles à produire dans l'arbre déjà déclaré

for(c=0; c<nbrEnsblAProduire; c++){

a0.ajouter(e[c]);

}

cout<<"On a donc les "<<nbrEnsblAProduire<<" ensembles a produire suivants : " ;

cout<<endl;

//Affichage de tous les ensembles à produire

a0.afficher();

cout<<"Le produit cartesien ";

for(c=0; c<nbrEnsblAProduire; c++){

if(c<(nbrEnsblAProduire-1)) cout<<"E"<<c<<" X ";

else cout<<"E"<<c<<" ="<<endl;

}

//La fonction produire suivante, de la classe Arbre est la fonction la plus //importante de notre algorithme car elle nous engendre la procédure de //résolution du problème de produit cartésien

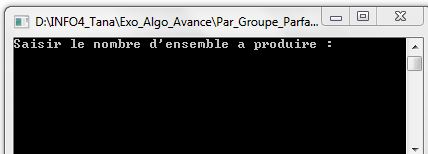
a0.produire();

return 0;

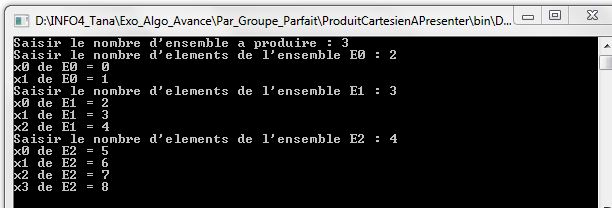
}

# Résultats obtenus après exécution du programme :

En exécutant cette application présentée, elle va demander d'abord à l'utilisateur le nombre d'élément à produire, comme indique la figure ci-après :



Supposons que l'utilisateur a tapé 3, c'est à dire qu'on a trois (03) ensembles à produire (E0, E1 et E2). Après cela, le programme va nous demander le nombre d'éléments de chaque ensemble et ainsi que tous ses contenus, comme la figure suivante l'indique:



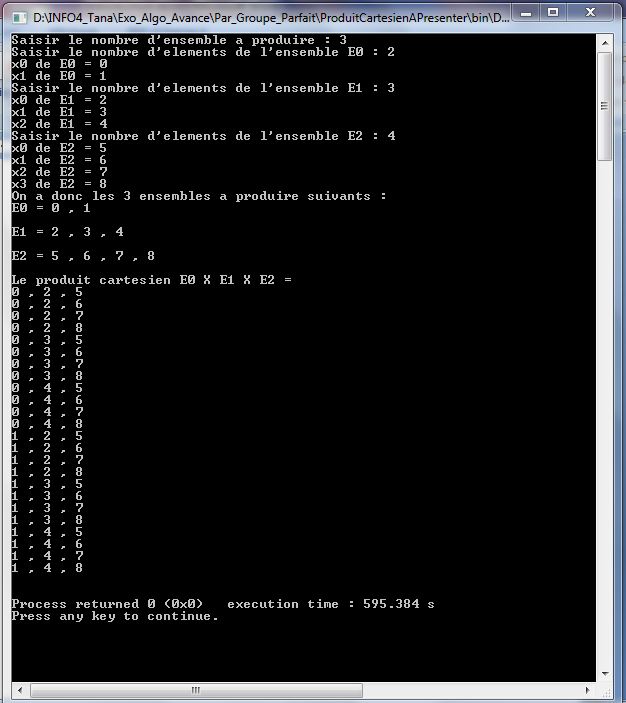
D'après tous ces saisis de l'utilisateur, il demande donc au programme de faire le produit cartésien des trois ensembles :

E0 = { 0, 1 }

E1 = { 2, 3, 4 }

E2 = { 5, 6, 7, 8 }

Après avoir cliqué sur la touche entrée, le programme nous affiche le résultat suivant :



Nous voyons que l'application a obtenu le bon résultat :

E0 x E1 x E2 = {(0, 2, 5), (0, 2, 6), (0, 2, 7), (0, 2, 8), (0, 3, 5), (0, 3, 6), (0, 3, 7), (0, 3, 8),

(0, 4, 5), (0, 4, 6), (0, 4, 7), (0, 4, 8), (1, 2, 5), (1, 2, 6), (1, 2, 7), (1, 2, 8),

(1, 3, 5), (1, 3, 6), (1, 3, 7), (1, 3, 8), (1, 4, 5), (1, 4, 6), (1, 4, 7), (1, 4, 8)}

# Conclusion :