Algorithmique et Programmation C++ 2 — TD 4

1. Send + More = Money

```
#include<iostream.h>



// Variables globales :
const int NV=8;
char C[NV]={'D','E','Y','N','R','O','S','M'};
enum L{D,E,Y,N,R,O,S,M};

// Prototypes des fonctions utilisees :
void afficher(int V[NV]);
bool ok(int V[NV], int n);
void chercher(int V[NV], int & n, int & compteur);



// Procedure MAIN :
int main()
{
    int V[NV];
    int n=0;
    int nSol=0;
    cout << "Recherche arborescente en cours..." << endl;
    chercher(V,n,nSol);
    cout << "Recherche terminee : "<< nSol
         << " solutions trouvees !" << endl;
    return 0;
}


// Fonction recursive de parcours de l'arbre :
void chercher(int V[NV], int & n, int & compteur)
{
    if(n>M)
    {
        afficher(V); // M=7 (cf enum...) :
        compteur++;
    }
    else
    {
        for(int i=0; i<10; i++)
        {
            V[n]=i;
            if(ok(V,n))
            {
                n++;
                chercher(V,n,compteur);
                n--;
            }
        }
    }
}
```

```
// Filtrage progressif des solutions :
bool ok(int V[NV], int n)
{
    int i,j,somme,retenue;
    // Teste d'abord si V[n] est different de
    // toutes les valeurs precedentes :
    for(i=D;i<n;i++)
        for(j=i+1;j<n;j++)
            if(V[i]==V[j]) return false;
    // Si les valeurs des lettres D,E,Y ne sont pas fixees,
    // on ne peut pas conclure :
    if(n<Y) return true;
    // Sinon on verifie de suite si D+E=Y :
    somme=V[D]+V[E];
    if(somme%10!=V[Y]) return false;
    // On calcule la retenue :
    retenue=somme/10;
    // De meme pour continuer les tests il faut que
    // N et R soient fixes (E l'est deja) :
    if(n<R) return true;
    // Pour teste si N+R+retenue=E...
    somme=V[N]+V[R]+retenue;
    if(somme%10!=V[E]) return false;
    // Et ainsi de suite :
    retenue=somme/10;
    if(n<O) return true;
    somme=V[E]+V[O]+retenue;
    if(somme%10!=V[N]) return false;
    retenue=somme/10;
    if(n<M) return true;
    if(V[M]==0) return false;
    somme=V[S]+V[M]+retenue;
    if(somme%10!=V[O]) return false;
    if(somme/10!=V[M]) return false;
    // Remarque : a ce stade, il est clair que nous
    // possedons LA solution en entier :
    return true;
}


// Affichage de la solution :
void afficher(int V[NV])
{
    cout << "   SEND\n"
         << " + MORE\n"
         << " -------\n"
         << "  MONEY\n" << endl;
    cout << "   " << V[S] << V[E] << V[N] << V[D] << endl
         << " + " << V[M] << V[O] << V[R] << V[E] << endl
         << " -------\n"
         << "  " <<V[M]<<V[O]<<V[N]<<V[E]<<V[Y]<<endl;
    cout << "\nAppuyez sur ENTREE pour continuer..." << endl;
    char d[3];
    cin.getline(d,3);
}
```

2. Jeu de pions

```
// Remarque : cet exercice a ete corrige, mais demeure
// incomplet et n'a pas ete teste.


#include<iostream.h>



const int TAILLE=7;
enum ValCase{vide,blanc,noir};

// Classe Pile (liste chainee vue au premier semestre)
class Pile;

// Teste si le plateau est dans l'etat final :
bool gagne(ValCase plateau[]);

// Filtrage :
bool deplacementOk(ValCase plateau[], int c, int d);

// Recherche arborescente des solutions :
void chercher(ValCase plateau[], Pile & solution);

// Deplace les pions ou les remet en place :
void jouer(ValCase plateau[], int c, int d);
void dejouer(ValCase plateau[], int c, int d);
```

```
// Recherche arborescente des solutions :
void chercher(ValCase plateau[], Pile & solution)
{
    if(gagne(plateau))
        // A chaque fois que l'on trouve une solution, on
        // affiche la pile des deplacements, qui contient la
        // liste des deplacements menant a la solution en cours :
        solution.afficher();
    else
    {
        int c;
        for(c=0; c<7; c++)
            // Pour ne pas deplacer un pion vide !
            if(c!=vide)
                for(int d=-2; d<3; d++)
                    if(deplacementOk(plateau(c,d))
                    {
                        jouer(plateau,c,d);
                        // Enregistre le deplacement actuel au sommet
                        // de la pile :
                        solution.empiler(c,d);
                        chercher(plateau, solution);
                        // On remonte dans l'arbre, retire le depla-
                        // -cement pour le remplacer par un autre.
                        solution.depiler();
                        dejouer(plateau,c,d);
                    }
    }
}
```

```
// Filtrage :
bool deplacementOk(ValCase plateau[], int c, int d)
{
    // c : case de depart du deplacement
    // d : ampleur du deplacement

    // Deplacement nul (pour ne pas avoir a le gerer dans le for) :
    if(d==0) return false;
    // Regle 1 : Rejette les deplacements interdits :
    if(plateau[c]==blanc && d<0)
        return false;
    // Idem :
    if(plateau[c]==noir) && d>0)
        return false;
    // Rejette les deplacements sortant du plateau :
    if(c+d<0 || c+d>TAILLE)
        return false;
    // Rejette les deplacements tombant sur une case deja occupee :
    if(plateau[c+d]!=vide)
        return false;
    // A ce stade le deplacement de 1 unite est valide :
    if(d==1 || d==-1)
        return true;
    // Regle 2 : Ici d=2 ou -2 : ceci teste que le pion
    // que l'on veut deplacer n'est pas de la meme couleur
    // que celui qu'on veut sauter :
    if(plateau[c]==plateau[c+d/2])
        return false;
    // A ce niveau, tout est OK :
    return true;
}
```

3. Le cavalier d'Euler

```
#include<iostream.h>
#include<iomanip.h>



// Taille de l'echiquier :
const int N=5;


// Compteur de solutions trouvees :
// Pour N=5, on trouve 1728 solutions distinctes !
int nbSol;


// Prototypes :
bool filtrage(int sol[N][N], int n, int x, int y);
void essayer(int sol[N][N], int n, int x, int y);
void affiche(int sol[N][N]);

inline void jouer(int sol[N][N], int n, int x, int y){sol[x][y]=n;}
inline void dejouer(int sol[N][N], int x, int y) {sol[x][y]=0;}



int main()
{
    int echiquier[N][N];
    int i,j;
    nbSol=0;
    // Initialise l'echiquier a vide avec des 0 :
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            echiquier[i][j]=0;
    cout << "Recherche en cours..." << endl;
    // Lance la recherche en placant des 1 sur chaque case :
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
        {
            jouer(echiquier,1,i,j);
            essayer(echiquier,1,i,j);
            dejouer(echiquier,i,j);
        }
    return 0;
}
```

```
void essayer(int sol[N][N], int n, int x, int y)
{
    if(n==N*N)
        affiche(sol);
    else
    {
        for(int i=-1; i<=1; i++)
            for(int j=-1; j<=1; j++)
                if(i!=0 && j!=0)
                {
                    if(filtrage(sol,n+1,x+2*i,y+j))
                    {
                        jouer(sol,n+1,x+2*i,y+j);
                        essayer(sol,n+1,x+2*i,y+j);
                        dejouer(sol,x+2*i,y+j);
                    }
                    if(filtrage(sol,n+1,x+j,y+2*i))
                    {
                        jouer(sol,n+1,x+j,y+2*i);
                        essayer(sol,n+1,x+j,y+2*i);
                        dejouer(sol,x+j,y+2*i);
                    }
                }
    }
}


bool filtrage(int sol[N][N], int n, int x, int y)
{
    // Le deplacement reste-t-il dans l'echiquier ?
    if(x>=N || x<0) return false;
    if(y>=N || y<0) return false;
    // La case cible est-elle bien vide ?
    if(sol[x][y]!=0) return false;
    // Tout est OK : le deplacement parait valide :
    return true;
}


void affiche(int sol[N][N])
{
    cout << "\n> Solution #" << ++nbSol << " :" << endl;
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
            cout << setw(3) << sol[i][j];
        cout << endl;
    }
    // A reactiver pour voir chaque solution :
    //cout << "Appuyez sur une ENTREE pour continuer.";
    //char a[3];
    //cin.getline(a,3);
    cout << "Recherche en cours..." << endl;
}
```